Министерство образования и науки Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

А. А. Изюмов

СПЕЦКУРС. ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебно-методическое пособие

Томск
«Эль Контент»
2013

УДК 004.42(075.8)
ББК 32.973.233-018я73
И 398

Рецензенты:
Комагоров В. П., доцент кафедры оптимизации систем упраления Национального исследовательского Томского политехнического университета;
Безходарнов И. В., исполнительный директор ООО «ТомскСофт».

Изюмов А. А.
И 398 Спецкурс. Технология разработки программного обеспечения : учебно-методическое пособие / А. А. Изюмов. — Томск: Эль Контент, 2013. — 174 с.

ISBN 978-5-4332-0091-3

Рассмотрены общие положения проектирования, моделирования и документирования бизнес-моделей, разрабатываемых на основе унифицированного языка моделирования. Даны понятия ER-моделей, диаграмм классов, диаграмм прецедентов. Приведен пример построения системы управления базой данных, построенной на основе языка MySQL.

УДК 004.42(075.8)
ББК 32.973.233-018я73

ISBN 978-5-4332-0091-3

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Целью данного курса является формирование у обучающихся навыков формализации задачи построения комплекса автоматизации бизнес-процессов. Эта задача подразделяется на следующие основные составляющие:

- построение модели бизнес-процесса в SADT-диаграмме;
- выделение в модели тех участков, автоматизация которых повлечет значительный выигрыш в функционировании всей системы;
- разработка программного обеспечения по автоматизации и создание сопроводительной документации с использованием специализированного программного комплекса в соответствии с UML;
- построение ER-диаграмм и создание базы данных;
- интеграция спроектированной базы данных в комплекс автоматизации.

В ходе курса предполагается сформировать у обучающегося следующие навыки:

- изучение теоретических основ языка UML;
- работа с пакетом UML-моделирования Enerprise Architect;
- закрепление навыков построения ER-диаграмм и изучение основ языка SQL.

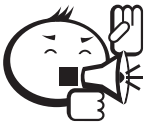
Выполнение лабораторных работ, таким образом, позволяет пройти полный путь от построения модели потребного процесса до получения готового коммерческого приложения, открытого для дальнейшей доработки любым программистом, с полным комплексом документации, оформленным в соответствии с общемировыми стандартами.

Изучение данного курса заканчивается сдачей экзамена. Для получения зачета необходимо выполнение всех лабораторных работ, подтвержденных соответствующими отчетами (в приложение к бумажному отчету необходимо наличие всех моделей и диаграмм в электронном виде).

Соглашения, принятые в книге

Для улучшения восприятия материала в данной книге используются пиктограммы и специальное выделение важной информации.

Эта пиктограмма означает определение или новое понятие.



Эта пиктограмма означает внимание. Здесь выделена важная информация, требующая акцента на ней. Автор здесь может поделиться с читателем опытом, чтобы помочь избежать некоторых ошибок.

Пример

Эта пиктограмма означает пример. В данном блоке автор может привести практический пример для пояснения и разбора основных моментов, отраженных в теоретическом материале.

Контрольные вопросы по главе

Глава 1

ОБЩАЯ ХАРАКТЕРИСТИКА МОДЕЛЕЙ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО АНАЛИЗА И ПРОЕКТИРОВАНИЯ

Язык UML представляет собой общецелевой язык визуального моделирования, который разработан для спецификации, визуализации, проектирования и документирования компонентов программного обеспечения, бизнес-процессов и других систем. Язык UML является достаточно строгим и мощным средством моделирования, которое может быть эффективно использовано для построения концептуальных, логических и графических моделей сложных систем различного целевого назначения. Этот язык вобрал в себя наилучшие качества и опыт методов программной инженерии, которые с успехом использовались на протяжении последних лет при моделировании больших и сложных систем.

С точки зрения методологии ООАП достаточно полная модель сложной системы представляет собой определенное число взаимосвязанных представлений (views), каждое из которых адекватно отражает аспект поведения или структуры системы. При этом наиболее общими представлениями сложной системы принято считать статическое и динамическое, которые, в свою очередь, могут подразделяться на другие более частные.

Принцип иерархического построения моделей сложных систем предписывает рассматривать процесс построения моделей на разных уровнях абстрагирования или детализации в рамках фиксированных представлений.

Уровень представления (layer) *— способ организации и рассмотрения модели на одном уровне абстракции, который представляет горизонтальный срез архитектуры модели, в то время как разбиение представляет ее вертикальный срез.*

При этом исходная или первоначальная модель сложной системы имеет наиболее общее представление и относится к концептуальному уровню. Такая модель, получившая название концептуальной, строится на начальном этапе проектирования и может не содержать многих деталей и аспектов моделируемой системы. Последующие модели конкретизируют концептуальную модель, дополняя ее представлениями логического и физического уровня.

В целом же процесс ООАП можно рассматривать как последовательный переход от разработки наиболее общих моделей и представлений концептуального уровня к более частным и детальным представлениям логического и физического уровня. При этом на каждом этапе ООАП данные модели последовательно дополняются все большим количеством деталей, что позволяет им более адекватно отражать различные аспекты конкретной реализации сложной системы. Общая схема взаимосвязей моделей ООАП представлена на рис. 1.1.

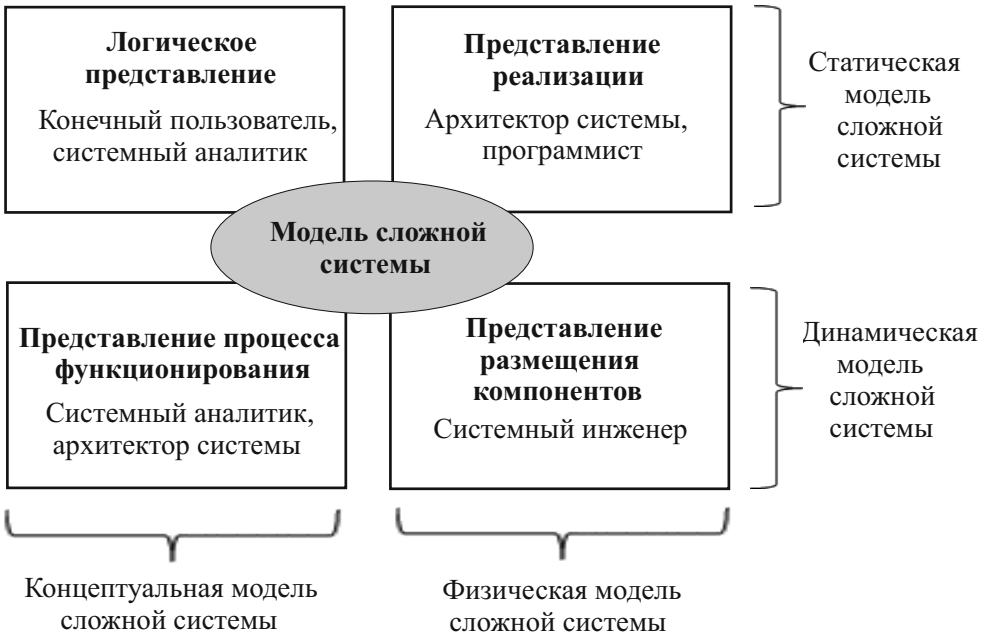


Рис. 1.1 – Общая схема взаимосвязей моделей ООАП

Для описания языка UML используются средства самого языка. К базовым средствам относится пакет, который служит для группировки элементов модели. При этом сами элементы модели, в том числе произвольные сущности, отнесенные к одному пакету, выступают в роли единого целого. При этом все разновидности элементов графической нотации языка UML организованы в пакеты.

1.1 Пакеты в языке UML

Пакет (package) *— общецелевой механизм для организации различных элементов модели в множество, реализующий системный принцип декомпозиции модели сложной системы и допускающий вложенность пакетов друг в друга.*

Пакет — основной способ организации элементов модели в языке UML. Каждый пакет владеет всеми своими элементами, т. е. теми элементами, которые включены в него. Про соответствующие элементы пакета говорят, что они принадлежат пакету или входят в него. При этом каждый элемент может принадлежать только одному пакету. В свою очередь, одни пакеты могут быть вложены в другие.

Подпакет (subpackage) *— пакет, который является составной частью другого пакета.*

По определению все элементы подпакета принадлежат и более общему пакету. Тем самым для элементов модели задается отношение вложенности пакетов, которое представляет собой иерархию.

Для графического изображения пакетов на диаграммах применяется специальный графический символ — большой прямоугольник с небольшим прямоугольником, присоединенным к левой части верхней стороны первого (рис. 1.2, *а, б*). Можно сказать, что визуально символ пакета напоминает пиктограмму папки в популярном графическом интерфейсе. Внутри большого прямоугольника может записываться информация, относящаяся к данному пакету. Если такой информации нет, то внутри большого прямоугольника записывается имя пакета, которое должно быть уникальным в пределах рассматриваемой модели (рис. 1.2, *а*). Если же такая информация имеется, то имя пакета записывается в верхнем маленьком прямоугольнике (рис. 1.2, *б*).

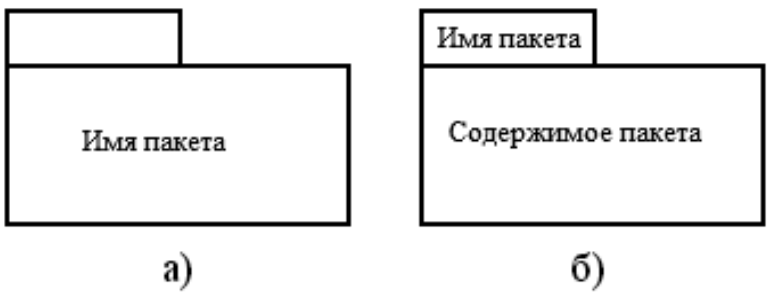


Рис. 1.2 – Графическое изображение пакетов в языке UML

Перед именем пакета может помещаться строка текста, содержащая ключевое слово, заранее определенное в языке UML и называемое стереотипом, например

facade, framework, stub и topLevel. В качестве содержимого пакета могут выступать имена его отдельных элементов и их свойства, такие как видимость элементов за пределами пакета. Одним из типов отношений между пакетами является отношение вложенности или включения пакетов друг в друга. В языке UML это отношение может быть изображено без использования линий простым размещением одного пакета-прямоугольника внутри другого пакета-прямоугольника (рис. 1.3). Так, в данном случае пакет с именем Пакет1 содержит в себе два подпакета: Пакет2 и Пакет3.

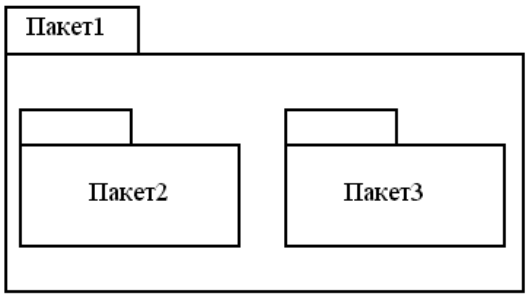


Рис. 1.3 – Графическое изображение вложенности пакетов

Кроме того, в языке UML это же отношение может быть изображено с помощью отрезков линий аналогично графическому представлению дерева. В этом случае наиболее общий пакет или контейнер изображается в верхней части рисунка, а его подпакеты — уровнем ниже. Контейнер соединяется с подпакетами сплошной линией, на конце которой, примыкающей к контейнеру, изображается специальный символ. Он означает, что подпакеты «собственность» или часть контейнера и, кроме этих подпакетов, контейнер не содержит никаких других. Рассмотренный выше пример (рис. 1.3) может быть представлен с помощью явной визуализации отношения включения (рис. 1.4).

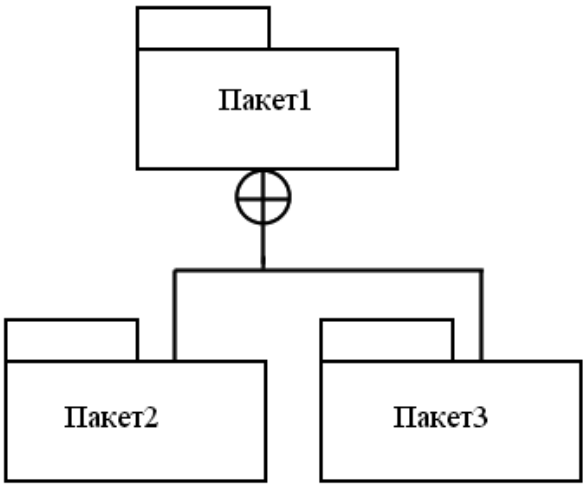


Рис. 1.4 – Графическое изображение вложенности пакетов с помощью явной визуализации отношения включения

Модель является подклассом пакета и представляет собой абстракцию физической системы, которая предназначена для вполне определенной цели. Именно эта цель предопределяет те компоненты, которые должны быть включены в модель, и те, рассмотрение которых не является обязательным. Другими словами, модель отражает релевантные аспекты физической системы, оказывающие непосредственное влияние на достижение поставленной цели. В прикладных задачах цель обычно задается в форме исходных требований к системе, которые, в свою очередь, в языке UML записываются в виде вариантов использования системы. [10]

1.2 Канонические диаграммы языка UML

В рамках языка UML все представления о модели сложной системы фиксируются в виде специальных графических конструкций, получивших название диаграмм.

Диаграмма (diagram) *— графическое представление совокупности элементов модели в форме связного графа, вершинам и ребрам (дугам) которого приписывается определенная семантика.*

Нотация канонических диаграмм *— основное средство разработки моделей на языке UML.*

В нотации языка UML определены следующие виды канонических диаграмм:

- вариантов использования (use case diagram),
- классов (class diagram),
- кооперации (collaboration diagram),
- последовательности (sequence diagram),
- состояний (statechart diagram),
- деятельности (activity diagram),
- компонентов (component diagram),
- развертывания (deployment diagram).

Перечень этих диаграмм и их названия являются каноническими в том смысле, что представляют собой неотъемлемую часть графической нотации языка UML. Более того, процесс ООАП неразрывно связан с процессом построения этих диаграмм. При этом совокупность построенных таким образом диаграмм является самодостаточной в том смысле, что в них содержится вся информация, которая необходима для реализации проекта сложной системы.

Каждая из этих диаграмм детализирует и конкретизирует различные представления о модели сложной системы в терминах языка UML. При этом диаграмма вариантов использования представляет собой наиболее общую концептуальную модель сложной системы, которая является исходной для построения всех осталь-

ных диаграмм. Диаграмма классов, по своей сути, логическая модель, отражающая статические аспекты структурного построения сложной системы.

Диаграммы кооперации и последовательностей представляют собой разновидности логической модели, которые отражают динамические аспекты функционирования сложной системы. Диаграммы состояний и деятельности предназначены для моделирования поведения системы. И, наконец, диаграммы компонентов и развертывания служат для представления физических компонентов сложной системы и поэтому относятся к ее физической модели.

В целом интегрированная модель сложной системы в нотации UML может быть представлена в виде совокупности указанных выше диаграмм (рис. 1.5).

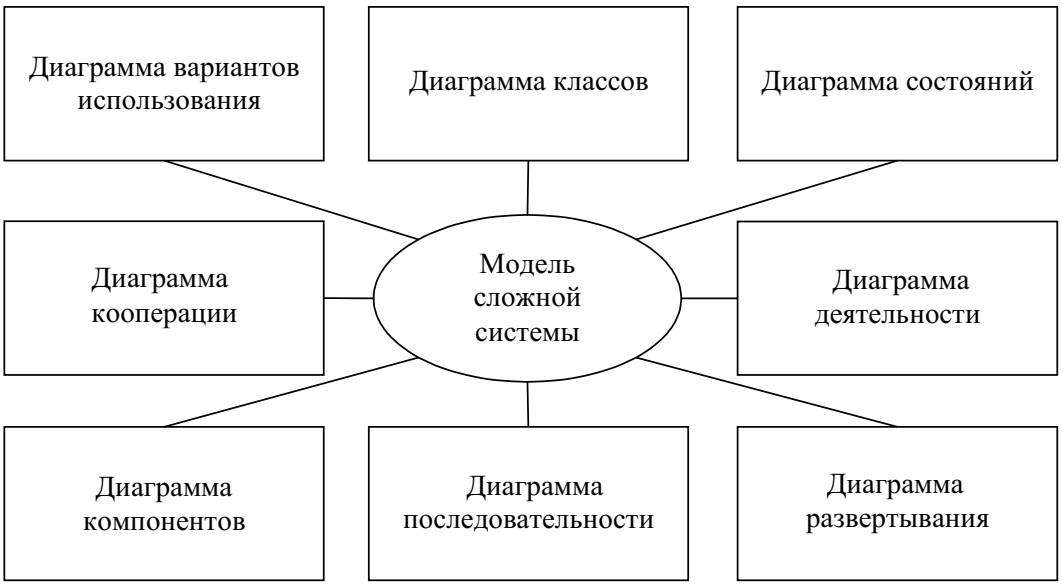


Рис. 1.5 – Интегрированная модель сложной системы в нотации UML

Кроме графических элементов, которые определены для каждой канонической диаграммы, на них может быть изображена текстовая информация, которая расширяет семантику базовых элементов.

Рассмотрим более подробно каждый из основных типов диаграмм.

1.2.1 Class diagram (диаграммы классов)

Этот тип диаграмм позволяет создавать логическое представление системы, на основе которого создается исходный код описанных классов.

Существуют разные точки зрения на построение диаграмм классов в зависимости от целей их применения:

- концептуальная точка зрения — диаграмма классов описывает модель предметной области, в ней присутствуют только классы прикладных объектов;
- точка зрения спецификации — диаграмма классов применяется при проектировании информационных систем;
- точка зрения реализации — диаграмма классов содержит классы, используемые непосредственно в программном коде (при использовании объектно-ориентированных языков программирования).

Значки диаграммы позволяют отображать сложную иерархию систем, взаимосвязи классов (Classes) и интерфейсов (Interfaces). Данный тип диаграмм противоположен по содержанию диаграмме Collaboration, на котором отображаются объекты системы. Существует несколько типов нотаций по созданию диаграмм данного типа. В нотации, предложенной Г. Бучем, которая так и называется Booch, классы изображаются в виде чего-то нечеткого (рис. 1.6), похожего на облако. Таким образом Г. Буч пытается показать, что класс — это лишь шаблон, по которому в дальнейшем будет создан конкретный объект.

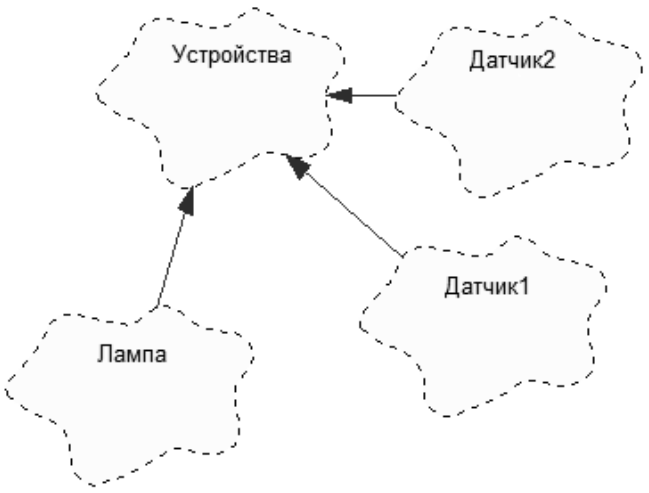


Рис. 1.6 – Диаграмма классов в нотации Гради Буча

Также возможно создавать диаграмму классов в нотации UMT и унифицированной нотации (рис. 1.7).

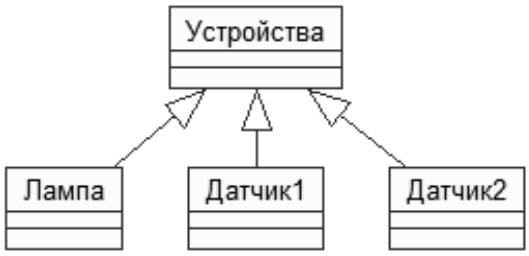


Рис. 1.7 – Диаграмма классов в унифицированной нотации

1.2.2 Component diagram (диаграммы компонентов)

Этот тип диаграмм (рис. 1.8) предназначен для распределения классов и объектов по компонентам при физическом проектировании системы. Часто данный тип диаграмм называют диаграммами модулей.

При проектировании больших систем может оказаться, что система должна быть разложена на несколько сотен или даже тысяч компонентов, и этот тип диаграмм позволяет не потеряться в обилии модулей и их связей. В качестве физических компонент могут выступать файлы, библиотеки, модули, исполняемые файлы, пакеты и т. п.

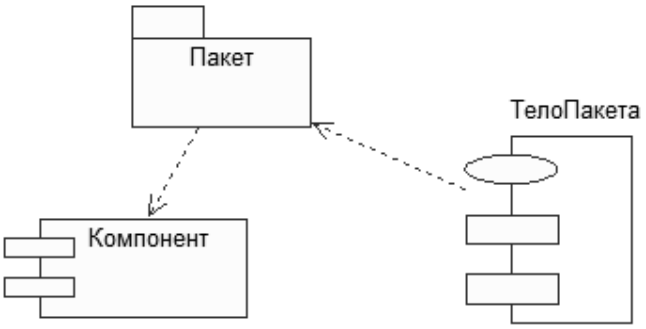


Рис. 1.8 – Диаграмма компонентов

1.2.3 Composite structure diagram (диаграмма композитной/составной структуры)

Статическая структурная диаграмма демонстрирует внутреннюю структуру классов и, по возможности, взаимодействие элементов (частей) внутренней структуры класса.

Подвидом диаграмм композитной структуры являются диаграммы кооперации (Collaboration diagram, введены в UML 2.0), которые показывают роли и взаимодействие классов в рамках кооперации. Кооперации удобны при моделировании шаблонов проектирования.

Диаграммы композитной структуры могут использоваться совместно с диаграммами классов.

1.2.4 Deployment diagram (диаграммы топологии)

Этот вид диаграмм (рис. 1.9) предназначен для анализа аппаратной части системы, то есть «железа», а не программ. В прямом переводе с английского Deployment означает «развертывание», но термин «топология» точнее отражает сущность этого типа диаграмм.

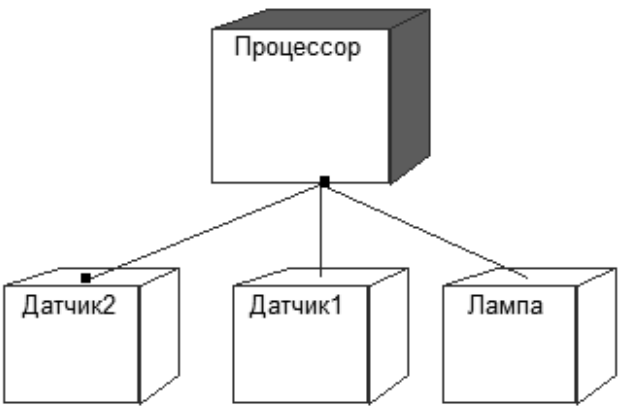


Рис. 1.9 – Диаграмма топологии

Для каждой модели создается только одна такая диаграмма, отображающая процессоры (Processor), устройства (Device) и их соединения.

Служит для моделирования работающих узлов (аппаратных средств, англ. *node*) и артефактов, развёрнутых на них. В UML 2 на узлах разворачиваются артефакты (англ. *artifact*), в то время как в UML 1 на узлах разворачивались компоненты. Между артефактом и логическим элементом (компонентом), который он реализует, устанавливается зависимость манифестации.

1.2.5 Object diagram (диаграмма объектов)

Демонстрирует полный или частичный снимок моделируемой системы в заданный момент времени. На диаграмме объектов отображаются экземпляры классов (объекты) системы с указанием текущих значений их атрибутов и связей между ними.

1.2.6 Use case diagram (диаграммы вариантов использования/прецедентов)

Этот вид диаграмм (рис. 1.10) позволяет создать список операций, которые выполняет система. Часто этот вид диаграмм называют диаграммой функций, потому что на основе набора таких диаграмм создается список требований к системе и определяется множество выполняемых системой функций.

Каждая такая диаграмма или, как ее обычно называют, каждый Use case — это описание сценария поведения, которому следуют действующие лица (Actors).

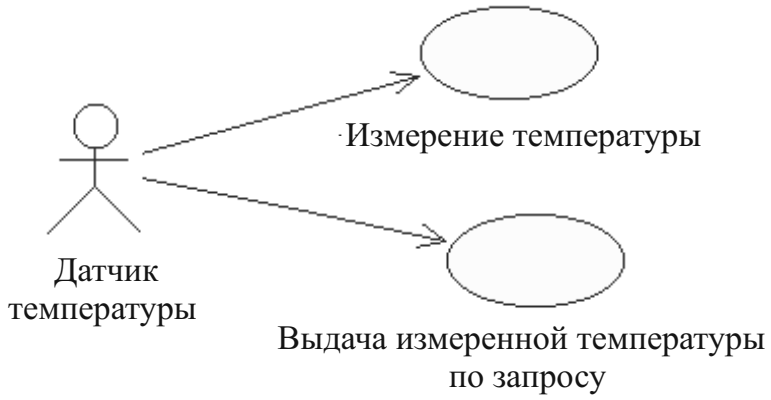


Рис. 1.10 – Диаграмма прецедентов

1.2.7 Statechart diagram (диаграмма состояний)

Диаграмма состояний (рис. 1.11) предназначена для отображения состояний объектов системы, имеющих сложную модель поведения.

Рис. 1.11 – Диаграмма состояний

1.2.8 Activity diagram (диаграммы активности/деятельности)

Это дальнейшее развитие диаграммы состояний (рис. 1.12). Фактически данный тип диаграмм может использоваться и для отражения состояний моделируемого объекта, однако основное назначение Activity diagram в том, чтобы отражать бизнес-процессы объекта, технологические переходы. Этот тип диаграмм позволяет показать не только последовательность процессов, но и ветвление, и даже синхронизацию процессов.

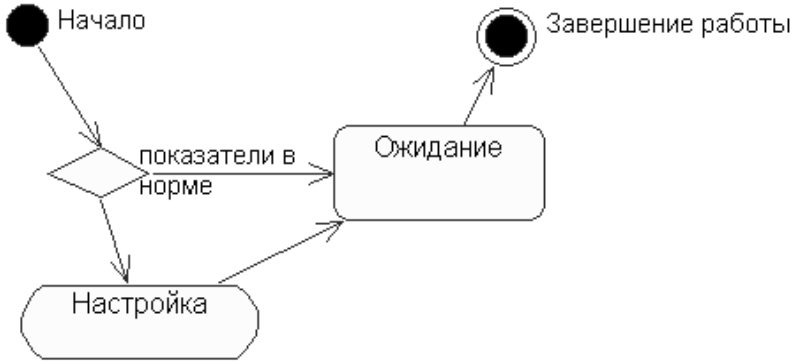


Рис. 1.12 – Диаграмма активности

Этот тип диаграмм позволяет проектировать алгоритмы поведения объектов любой сложности, в том числе может использоваться для составления блок-схем.

1.2.9 Interaction diagram (диаграммы взаимодействия)

Этот тип диаграмм включает в себя диаграммы Sequence diagram (диаграммы последовательностей действий) и Collaboration diagram (диаграммы сотрудничества). Эти диаграммы позволяют с разных точек зрения рассмотреть взаимодействие объектов в создаваемой системе.

1.2.10 Sequence diagram (диаграммы последовательностей действий)

Взаимодействие объектов в системе происходит посредством приема и передачи сообщений объектами-клиентами и обработки этих сообщений объектами-

серверами. При этом в разных ситуациях одни и те же объекты могут выступать и в качестве клиентов, и в качестве серверов.

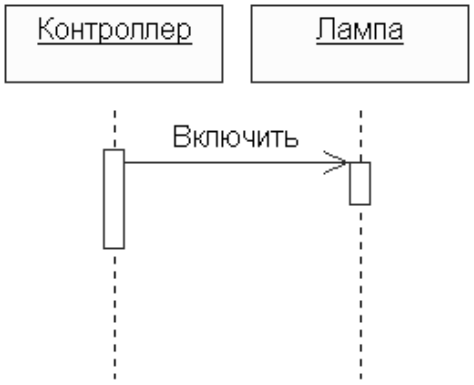


Рис. 1.13 – Диаграмма последовательностей действий

Данный тип диаграмм позволяет отразить последовательность передачи сообщений между объектами (рис. 1.13).

Этот тип диаграммы не акцентирует внимание на конкретном взаимодействии, главный акцент уделяется последовательности приема/передачи сообщений. Для того чтобы окинуть взглядом все взаимосвязи объектов, служит Collaboration diagram.

1.2.11 Collaboration diagram (диаграммы сотрудничества)

Этот тип диаграмм (рис. 1.14) позволяет описать взаимодействия объектов, абстрагируясь от последовательности передачи сообщений. На этом типе диаграмм в компактном виде отражаются все принимаемые и передаваемые сообщения конкретного объекта и типы этих сообщений.

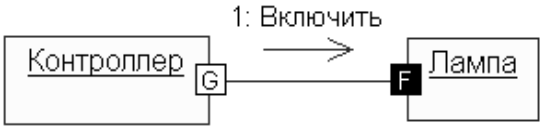


Рис. 1.14 – Диаграмма сотрудничества

По причине того, что диаграммы Sequence и Collaboration являются разными взглядами на одни и те же процессы.

1.2.12 Interaction overview diagram (диаграмма обзора взаимодействия)

Разновидность диаграммы деятельности, включающая фрагменты диаграммы последовательности и конструкции потока управления.

1.2.13 Timing diagram (диаграмма синхронизации)

Альтернативное представление диаграммы последовательности, явным образом показывающее изменения состояния на линии жизни с заданной шкалой времени. Может быть полезна в приложениях реального времени.

1.3 Особенности графического изображения диаграмм языка UML

Большинство перечисленных выше диаграмм являются в своей основе графами специального вида, состоящими из вершин в форме геометрических фигур, которые связаны между собой ребрами или дугами. Поскольку информация, которую содержит в себе граф, носит топологический характер, ни геометрические размеры, ни расположение элементов диаграмм не имеют принципиального значения.

Для диаграмм языка UML существуют три типа визуальных графических обозначений, которые важны с точки зрения заключенной в них информации:

- Геометрические фигуры на плоскости, играющие роль вершин графов соответствующих диаграмм. При этом сами геометрические фигуры выступают в роли графических примитивов языка UML, а форма этих фигур (прямоугольник, эллипс) должна строго соответствовать изображению отдельных элементов языка UML (класс, вариант использования, состояние, деятельность). Графические примитивы языка UML имеют фиксированную семантику, переопределять которую пользователям не допускается. Графические примитивы должны иметь собственные имена, а, возможно, и другой текст, который содержится внутри границ соответствующих геометрических фигур или, как исключение, вблизи этих фигур.
- Графические взаимосвязи, которые представляются различными линиями на плоскости. Взаимосвязи в языке UML обобщают понятие дуг и ребер из теории графов, но имеют менее формальный характер и более развитую семантику.
- Специальные графические символы, изображаемые вблизи от тех или иных визуальных элементов диаграмм и имеющие характер дополнительной спецификации (украшений).

Все диаграммы в языке UML изображаются с использованием фигур на плоскости. Отдельные элементы — с помощью геометрических фигур, которые могут иметь различную высоту и ширину с целью размещения внутри них других конструкций языка UML. Наиболее часто внутри таких символов помещаются строки текста, которые уточняют семантику или фиксируют отдельные свойства соответствующих элементов языка UML. Информация, содержащаяся внутри фигур, имеет значение для конкретной модели проектируемой системы, поскольку регламентирует реализацию соответствующих элементов в программном коде.

Пути представляют собой последовательности из отрезков линий, соединяющих отдельные графические символы. При этом концевые точки отрезков линий должны обязательно соприкасаться с геометрическими фигурами, служащими для обозначения вершин диаграмм, как принято в теории графов. С концептуальной точки зрения путям в языке UML придается особое значение, поскольку это про-

стые топологические сущности. Отдельные части пути или сегменты могут не существовать вне содержащего их пути. Пути всегда соприкасаются с другими графическими символами на обеих границах соответствующих отрезков линий, т. е. пути не могут обрываться на диаграмме линией, которая не соприкасается ни с одним графическим символом. Как отмечалось выше, пути могут иметь в качестве окончания или терминатора специальную графическую фигуру — значок, который изображается на одном из концов линий.

Дополнительные значки или украшения представляют собой графические фигуры фиксированного размера и формы. Они не могут увеличивать свои размеры, чтобы разместить внутри себя дополнительные символы. Значки размещаются как внутри других графических конструкций, так и вне их. Примерами значков могут служить окончания связей элементов диаграмм или графические обозначения кванторов видимости атрибутов и операций классов.

Строки текста служат для представления различных видов информации в грамматической форме. Предполагается, что каждое использование строки текста должно соответствовать синтаксису в нотации языка UML. В отдельных случаях может быть реализован грамматический разбор этой строки, который необходим для получения дополнительной информации о модели. Например, строки текста в различных секциях обозначения класса могут соответствовать атрибутам этого класса или его операциям. На использование строк накладывается условие: требуется, чтобы семантика всех допустимых символов была заранее определена в языке UML или служила предметом его расширения в конкретной модели.

При графическом изображении диаграмм следует придерживаться следующих основных рекомендаций:

- Каждая диаграмма должна служить законченным представлением соответствующего фрагмента моделируемой предметной области. Речь идет о том, что в процессе разработки диаграммы необходимо учесть все сущности, важные с точки зрения контекста данной модели и диаграммы. Отсутствие тех или иных элементов на диаграмме служит признаком неполноты модели и может потребовать ее последующей доработки.
- Все сущности на диаграмме модели должны быть одного уровня представления. Здесь имеется в виду согласованность не только имен одинаковых элементов, но и возможность вложения отдельных диаграмм друг в друга для достижения полноты представлений. В случае достаточно сложных моделей систем желательно придерживаться стратегии последовательного уточнения или детализации отдельных диаграмм.
- Вся информация о сущностях должна быть явно представлена на диаграммах. В языке UML при отсутствии некоторых символов на диаграмме могут быть использованы их значения по умолчанию (например, в случае неявного указания видимости атрибутов и операций классов), тем не менее необходимо стремиться к явному указанию свойств всех элементов диаграмм.
- Диаграммы не должны содержать противоречивой информации. Противоречивость модели может служить причиной серьезных проблем при ее реализации и последующем использовании на практике. Например, наличие замкнутых путей при изображении отношений агрегирования или композиции приводит к ошибкам в программном коде, который будет реализовывать соответствующие классы. Наличие элементов с одинаковыми именами

и различными атрибутами свойств в одном пространстве имен также приводит к неоднозначной интерпретации и может быть источником проблем.

- Каждая диаграмма должна быть самодостаточной для правильной интерпретации всех ее элементов и понимания семантики всех используемых графических символов. Любые пояснительные тексты, которые не являются собственными элементами диаграммы (например, комментариями), не должны приниматься во внимание разработчиками. В то же время общие фрагменты диаграмм могут уточняться или детализироваться на других диаграммах этого же типа, образуя вложенные или подчиненные диаграммы. Таким образом, модель системы на языке UML представляет собой пакет иерархически вложенных диаграмм, детализация которых должна быть достаточной для последующей генерации программного кода, реализующего проект соответствующей системы.
- Количество типов диаграмм для конкретной модели приложения строго не фиксировано. Для простых приложений нет необходимости строить все без исключения типы диаграмм. Некоторые из них могут просто отсутствовать в проекте системы, и это не будет считаться ошибкой разработчика. Например, модель системы может не содержать диаграмму развертывания для приложения, выполняемого локально на компьютере пользователя. Важно понимать, что перечень диаграмм зависит от специфики конкретного проекта системы.
- Любая модель системы должна содержать только те элементы, которые определены в нотации языка UML. Имеется в виду требование начинать разработку проекта, используя только те конструкции, которые уже определены в метамодели UML. Как показывает практика, этих конструкций вполне достаточно для представления большинства типовых проектов программных систем. И только при отсутствии необходимых базовых элементов языка UML следует использовать механизмы их расширения для адекватного представления конкретной модели системы. Не допускается переопределение семантики тех элементов, которые отнесены к базовой нотации метамодели языка UML [11].

Лабораторная работа №1

Цель работы

Целью данной работы является построение Use Case диаграммы выбранной предметной области.

Краткое изложение теоретической части

Интерфейс Sparx Enterprise Atchitect

Данная программа визуально состоит из различных функциональных блоков (рис. 1.15):

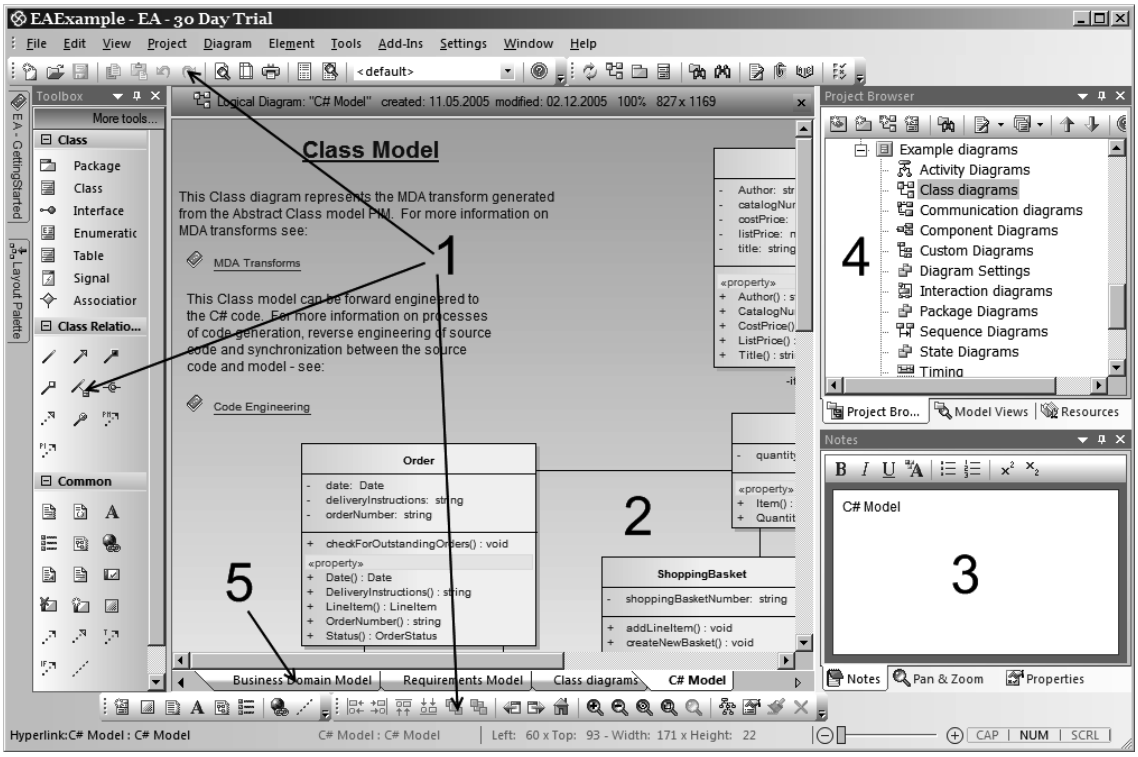


Рис. 1.15 – Интерфейс Sparx Enterprise Atchitect

Визуально выделяются:

1) панели инструментов,
2) окно диаграмм,
3) окно документирования,
4) дерево модели,
5) вкладки диаграмм.

Важным достоинством программы является поддержка стандарта UML 2.1, богатые возможности по визуализации и кодогенерации проектов разных типов.

Для создания модели в Enterprise Architect сначала необходимо выбрать пункт меню <*File/New Project…*>. Далее следует ввести имя проекта, после чего необходимо будет определить заранее те модели, которые требуется создать в данный момент (рис. 1.16–1.17) — диаграммы для моделей (и сами модели иных аспектов функционирования моделируемой системы) можно добавлять и на более поздних этапах проектирования.

Выбранные диаграммы станут доступны в менеджере проекта. Для перехода к нужной диаграмме необходимо выполнить двойной щелчок на нужной в дереве, после чего станет возможным перемещаться между диаграммами через выбор нужной закладки внизу экрана.

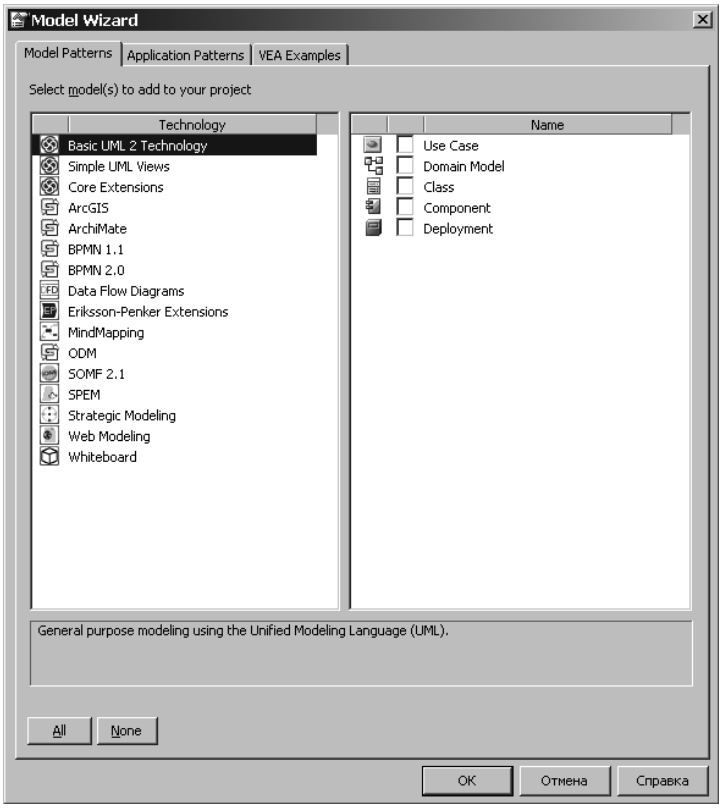


Рис. 1.16 – Выбор диаграмм для модели в Enterprise Atchitect

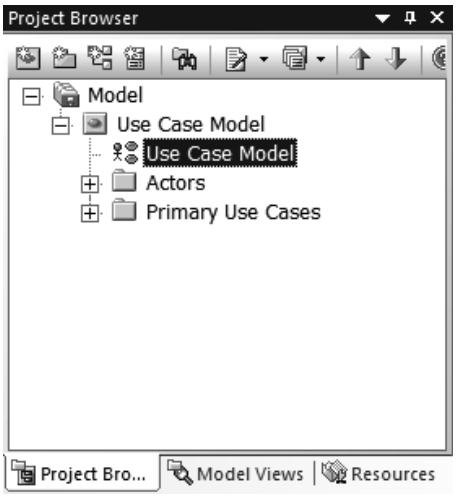


Рис. 1.17 – Диаграммы в Project Browser

В случае, если пользователь решит добавить к модели дополнительные диаграммы, он может сделать это путём нажатия пиктограммы, изображенной на рис. 1.18.

Набор кнопок в панели инструментов зависит от того, какая диаграмма применяется. Например, в случае диаграммы прецедентов, она будет иметь вид как на рис. 1.19-1, в случае диаграммы классов, рис. 1.19-2.

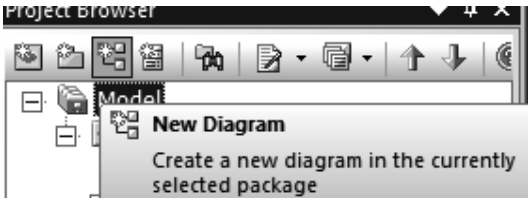


Рис. 1.18 – Добавление диаграммы к модели

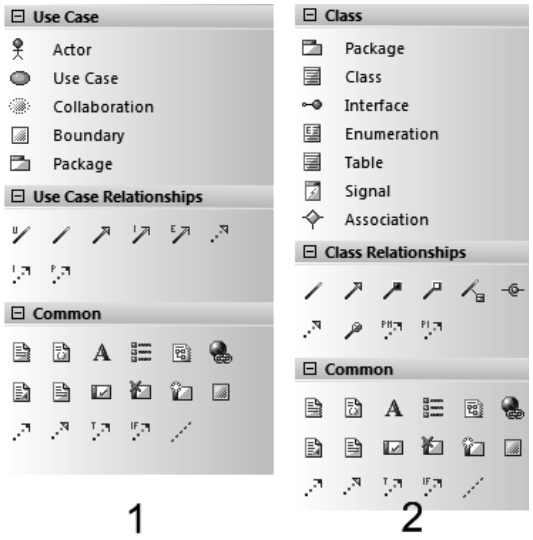


Рис. 1.19 – Внешний вид панели инструментов Enterprise Architect

Описание элементов интерфейса приведено в таблице 1.1. Описаны лишь те элементы, которые пригодятся для выполнения работ данного методического пособия.

Таблица 1.1 – Элементы панели инструментов Enterprise Architect

	Наименование	**Назначение**
	Actor	Новый актер
	Use Case	Новый прецедент
	Collaboration	Создает хранилище, логически объединяющее связанный набор ролей и связей между ними.
	Boundary	Создает визуальную структуру, объединяющую элементы. Функциональной нагрузки не несёт.
	Package	Контейнер, который может содержать в себе как другие контейнеры и диаграммы, так и набор элементов, объединенных какой-либо логикой
	Use	Создает отношение использования
	Associate	Создает отношение ассоциации
	Generalize	Создает отношение наследования

продолжение на следующей странице

Таблица 1.1 — Продолжение

	Наименование	Назначение
	Include	Создает отношение включения
	Extend	Создает отношение расширения
	Realize	Описывает отношение, при котором набор элементов приемника описывается набором элементов передатчика
	Invokes	Данные отношения определены в OML (Open Modeling Language). Описывают отношения зависимости. Invokes показывает, что прецендент *A* в конкретный момент влечет выполнение *B*, в то время, как Precedes показывает, что перед выполнением *D* должен полностью завершиться прецедент *C*.
	Precedes	
	Note	Создает заметку
	Constraint	Создает ограничение
	Text element	Создает текстовый элемент
	Diagram legend	Добавляет к диаграмме легенду
	Diagram notes	Добавляет информацию по диаграмме
	Hyperlink	Добавляет на лист диаграммы гиперссылку
	Document	Позволяет вложить в диаграмму документ
	Artifact	Экспонат — любая физическая информация, используемая или произведенная системой, представленной в диаграмме Развертывания. Экспонаты могут связывать свойства или операции и могут иллюстрироваться примерами или быть связанными с другими Экспонатами. Примеры Экспонатов включают файлы моделей, исходные файлы, таблицы базы данных и т. д.
	Requirement	Описывает требование. Внешнее или внутреннее
	Issue	Описывает дефект
	Change	Описывает изменение
	Boundary	Создает визуальную структуру, объединяющую элементы.

продолжение на следующей странице

Таблица 1.1 — Продолжение

	Наименование	**Назначение**
	Dependency	Создает связь зависимости. Это означает, что элемент или набор элементов связанный отношением зависимости, требует других элементов для своего полноценного описания или выполнения
	Trace	Подвид отношения зависимости. Соединяет элементы или наборы элементов, представляющие те же принципы, но в масштабе моделей
	Information flow	Описывает поток данных (информации), передаваемых от передатчика к приемнику
	Note link	Создает линию, описывающую связь с заметкой
	Package	Сущность, которая представляет собой группу классов и интерфейсов
	Class	Создает класс. Внешний вид класса описывается тремя секциями: • в верхней указывается имя; • в средней описываются свойства, атрибуты, и ссылки; • в нижней части описываются методы (процедуры и функции). Подчеркивание означает статическую операцию.
	Interface	Интерфейсы аналогичны классам, за исключением заголовка с < >
	Enumeration	Создает перечисление
	Table	Создает таблицу — частный вид класса. Отличается наличием свойств с описанием типа базы данных и возможностью указать информацию по колонкам, а также определить триггеры и индексы
	Signal	Определяет сигнал — передачу запроса Send. Объект получения обрабатывает экземпляры класса запроса как определено его свойствами. Данные, которые несёт Send, представлены как атрибуты сигнала. Сигнал определен независимо от классификаторов, обрабатывающих его возникновение.
	Association	*n*-арная ассоциация связывает три и более элемента

продолжение на следующей странице

Таблица 1.1 — Продолжение

	Наименование	Назначение
	Compose	Используется для демонстрации агрегации большого числа мелких частей в более крупный объект. При удалении родительского объекта удаляются и связанные
	Aggregate	Создает отношение включения
	Association class	Создает включенный класс
	Assembly	Создает связь, указывающую на необходимость описания интерфейса между связанными объектами
	Nesting	Создает связь, графически описывающую механизм включения одними элементами других
	Package merge	Создает связь, описывающую объединение элементов
	Package import	Создает связь, описывающую, что один пакет импортирует другой
	Lifeline	Описывает конкретный экземпляр, участвующий во взаимодействии (множественность не подразумевается).
	Boundary	Описывает объект, который имеет некоторую ограниченность. Используется в концептуальной фазе проектирования, чтобы фиксировать пользователей, взаимодействующих с системой
	Control	Создает объект контроль — показывающий, что модель контролирует сущность или актера
	Entity	Создает сущность
	Fragment	Описывает фрагмент
	Endpoint	Описывает точку остановки
	Diagram gate	Графически обозначает точку, через/в которую идёт передача информации для фрагмента
	State	Описывает некое состояние системы — статичное или динамичное
	Message	Описывает сообщение
	Self message	Описывает сообщение объекта для самого себя
	Call	Описывает вызов
	Recursion	Описывает рекурсивный вызов
	Activity	Описывает активность, возникающую как элемент функционирования системы или передачи информации
	Action	Действие описывает базовый процесс или передачу информации, которые возникают в системе

продолжение на следующей странице

Таблица 1.1 — Продолжение

	Наименование	Назначение
	Partition	Используется для логической организации действий
	Object	Частный случай класса в момент его выполнения
	Central buffer node	Описывает узел для управления потоками информации, поступающей из нескольких источников
	Datastore	Описывает хранилище данных
	Decision	Описывает условие — момент, в который возможно изменить поведение системы
	Merge	Узел слияния — узел контроля, который примиряет множественные потоки данных. Не используется, чтобы синхронизировать параллельные потоки: просто выбирает один из многих и идёт по нему.
	Send	Используется, чтобы изобразить действие по посылке сигналов
	Receive	Используется, чтобы изобразить действие по приему сигналов
	Synch	Описывает точку, изображающую ситуацию синхронизации параллельных путей функционирования модели
	Initial	Описывает точку начала
	Final	Описывает точку конца
	Flow final	Описывает конец информационного потока (преднамеренный и нет)
	Exception	Описывает ситуацию исключения
	Fork/join	Создает элемент, являющийся буферным и позволяющий разделить информацию на несколько потоков
	Control flow	Соединяет две активности в диаграмме активности
	Object flow	Соединяет две активности или состояния и показывает факт передачи объектов между ними
	Interrupt flow	Абстрактный класс для направленных подключений между двумя узлами деятельности

Размещение объектов разных типов не вызывает никаких трудностей. Однако следует по максимуму использовать возможности Enterprise Architect по объединению элементов в пакеты и группы. Следует учитывать, что в одной модели может быть несколько диаграмм одного типа, описывающего разные варианты поведения системы, и чем более полно разработчик стремится отразить функционирование модели, тем больше будет этих диаграмм.

При единократном размещении объекта в модели, его можно использовать в разных диаграммах, причём использовать в трёх разных ролях (рис. 1.20):

- В качестве ссылки на объект (физически объект останется на той диаграмме, на которой был размещен первично, а пользователь, изменяя свойства и имя объекта в дочерней диаграмме, будет менять свойства объекта и на родительской).
- В качестве экземпляра существующего класса (при этом размещаемый объект получит собственное имя, однако свойства его будут определяться свойствами объекта, экземпляром класса которого он является).
- В качестве потомка родительского объекта (при этом, наследуя все свойства родительского объекта, объект-потомок может обладать и своими уникальными атрибутами).

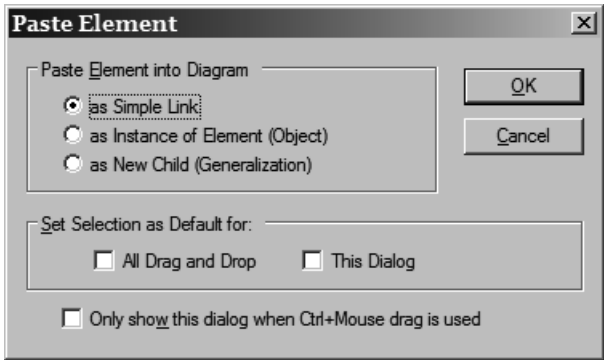


Рис. 1.20 – Размещение элемента в диаграмме

Визуальное моделирование с использованием нотации UML можно представить как процесс поуровневого спуска от наиболее общей и абстрактной концептуальной модели исходной бизнес-системы к логической, а затем и к физической модели соответствующей программной системы. Для достижения этих целей вначале строится модель в форме так называемой диаграммы вариантов использования (use case diagram), которая описывает функциональное назначение системы, или, другими словами, то, что бизнес-система должна делать в процессе своего функционирования.

Диаграмма вариантов использования (use case diagram) *— диаграмма, на которой изображаются отношения между актерами и вариантами использования.*

Диаграмма вариантов использования — это исходное концептуальное представление или концептуальная модель системы в процессе ее проектирования и разработки. Создание диаграммы вариантов использования имеет следующие цели:

- определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы;
- сформулировать общие требования к функциональному поведению проектируемой системы;
- разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей;

- подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

Назначение данной диаграммы состоит в следующем: проектируемая программная система представляется в форме так называемых вариантов использования, с которыми взаимодействуют внешние сущности или актеры. При этом актером или действующим лицом называется любой объект, субъект или система, взаимодействующая с моделируемой бизнес-системой извне. Это может быть человек, техническое устройство, программа или любая другая система, которая служит источником воздействия на моделируемую систему так, как определит разработчик. Вариант использования служит для описания сервисов, которые система предоставляет актеру. Другими словами, каждый вариант использования определяет набор действий, совершаемый системой при диалоге с актером. При этом ничего не говорится о том, каким образом будет реализовано взаимодействие актеров с системой и собственно выполнение вариантов использования.

Рассматривая диаграмму вариантов использования в качестве модели бизнес-системы, можно ассоциировать ее с «черным ящиком». Концептуальный характер этой диаграммы проявляется в том, что подробная детализация диаграммы или включение в нее элементов физического уровня представления на начальном этапе проектирования скорее имеет отрицательный характер, поскольку предопределяет способы реализации поведения системы. Эти аспекты должны быть сознательно скрыты от разработчика на диаграмме вариантов использования.

В самом общем случае, диаграмма вариантов использования представляет собой граф специального вида, который является графической нотацией для представления конкретных вариантов использования, актеров и отношений между этими элементами. При этом отдельные элементы диаграммы заключают в прямоугольник, который обозначает границы проектируемой системы. В то же время отношения, которые могут быть изображены на данном графе, представляют собой только фиксированные типы взаимосвязей между актерами и вариантами использования, которые в совокупности описывают сервисы или функциональные требования к моделируемой системе.

Базовыми элементами диаграммы вариантов использования являются вариант использования и актер.

Вариант использования (use case) *— внешняя спецификация последовательности действий, которые система или другая сущность могут выполнять в процессе взаимодействия с актерами.*

Вариант использования представляет собой спецификацию общих особенностей поведения или функционирования моделируемой системы без рассмотрения внутренней структуры этой системы. Несмотря на то, что каждый вариант использования определяет последовательность действий, которые должны быть выполнены проектируемой системой при взаимодействии ее с соответствующим актером, сами эти действия не изображаются на рассматриваемой диаграмме.

Содержание варианта использования может быть представлено в форме дополнительного пояснительного текста, который раскрывает смысл или семантику

действий при выполнении данного варианта использования. Такой пояснительный текст получил название текста-сценария или просто сценария.

Отдельный вариант использования обозначается на диаграмме эллипсом, внутри которого содержится его краткое имя в форме существительного или глагола) с пояснительными словами. Сам текст имени варианта использования должен начинаться с заглавной буквы.

Имя (name) *— строка текста, которая используется для идентификации любого элемента модели.*

Цель спецификации варианта использования заключается в том, чтобы зафиксировать некоторый аспект или фрагмент поведения проектируемой системы без указания особенностей реализации данной функциональности. В этом смысле каждый вариант использования соответствует отдельному сервису, который предоставляет моделируемая система по запросу актера, т. е. определяет один из способов применения системы. Сервис, который инициализируется по запросу актера, должен представлять собой законченную последовательность действий. Это означает, что после того как система закончит обработку запроса актера, она должна возвратиться в исходное состояние, в котором снова готова к выполнению следующих запросов.

Диаграмма вариантов использования содержит конечное множество вариантов использования, которые в целом должны определять все возможные стороны ожидаемого поведения системы. Для удобства множество вариантов использования может рассматриваться как отдельный пакет. Применение вариантов использования на всех этапах работы над проектом позволяет не только достичь требуемого уровня унификации обозначений для представления функциональности подсистем и системы в целом, но и является мощным средством последовательного уточнения требований к проектируемой системе на основе их итеративного обсуждения со всеми заинтересованными специалистами.

Примерами вариантов использования могут быть следующие действия: *проверка состояния текущего счета клиента, оформление заказа на покупку товара, получение дополнительной информации о кредитоспособности клиента, отображение графической формы на экране монитора и другие действия.*

Актер (actor) *— согласованное множество ролей, которые играют внешние сущности по отношению к вариантам использования при взаимодействии с ними.*

Актер представляет собой любую внешнюю по отношению к моделируемой системе сущность, которая взаимодействует с системой и использует ее функцио-

нальные возможности для достижения определенных целей или решения частных задач. Каждый актер может рассматриваться как некая отдельная роль относительно конкретного варианта использования. Стандартным графическим обозначением актера на диаграммах является фигурка «человечка», под которой записывается имя актера (рис. 1.21).

В некоторых случаях актер может обозначаться в виде прямоугольника класса со стереотипом ≪actor≫ и обычными составляющими элементами класса. Имена актеров должны начинаться с заглавной буквы и следовать рекомендациям использования имен для типов и классов модели. При этом символ отдельного актера связывает соответствующее описание актера с конкретным именем.

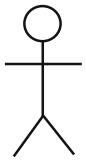


Рис. 1.21 – Графическое обозначение актера

Имя актера должно быть достаточно информативным с точки зрения семантики. Для этой цели подходят наименования должностей в компании (например, *продавец, кассир, менеджер, президент*). Не рекомендуется давать актерам имена собственные или названия моделей конкретных устройств, даже если это с очевидностью следует из контекста проекта. Дело в том, что одно и то же лицо может выступать в нескольких ролях и соответственно обращаться к различным сервисам системы.

Актеры используются для моделирования внешних по отношению к проектируемой системе сущностей, которые взаимодействуют с системой. В качестве актеров могут выступать другие системы, в том числе подсистемы проектируемой системы или ее отдельные классы. Важно понимать, что каждый актер определяет согласованное множество ролей, в которых могут выступать пользователи данной системы в процессе взаимодействия с ней. В каждый момент времени с системой взаимодействует вполне определенный пользователь, при этом он играет или выступает в одной из таких ролей. Наиболее наглядный пример актера — конкретный посетитель web-сайта в Интернете со своими параметрами аутентификации.

Поскольку в общем случае актер всегда находится вне системы, его внутренняя структура никак не определяется. Для актера имеет значение только его внешнее представление, т. е. то, как он воспринимается со стороны системы. Актеры взаимодействуют с системой посредством передачи и приема сообщений от вариантов использования. Сообщение представляет собой запрос актером сервиса от системы и получение этого сервиса. Это взаимодействие может быть выражено посредством ассоциаций между отдельными актерами и вариантами использования. Кроме этого, с актерами могут быть связаны интерфейсы, которые определяют, каким образом другие элементы модели взаимодействуют с этими актерами.

Отношения на диаграмме вариантов использования

Отношение (relationship) *— семантическая связь между отдельными элементами модели.*

Между элементами диаграммы вариантов использования могут существовать различные отношения, которые описывают взаимодействие экземпляров одних актеров и вариантов использования с экземплярами других актеров и вариантов. Один актер может взаимодействовать с несколькими вариантами использования. В этом случае этот актер обращается к нескольким сервисам данной системы. В свою очередь, один вариант использования может взаимодействовать с несколькими актерами, предоставляя для всех них свой сервис.

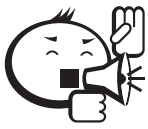
В то же время два варианта использования, определенные в рамках одной моделируемой системы, также могут взаимодействовать друг с другом, однако характер этого взаимодействия будет отличаться от взаимодействия с актерами. Однако в обоих случаях способы взаимодействия элементов модели предполагают обмен сигналами или сообщениями, которые инициируют реализацию функционального поведения моделируемой системы.

В языке UML имеется несколько стандартных видов отношений между актерами и вариантами использования:

- ассоциации (association relationship),
- включения (include relationship),
- расширения (extend relationship),
- обобщения (generalization relationship).

При этом общие свойства вариантов использования могут быть представлены тремя различными способами, а именно — с помощью отношений включения, расширения и обобщения.

Отношение ассоциации — одно из фундаментальных понятий в языке UML и в той или иной степени используется при построении всех графических моделей систем в форме канонических диаграмм. Применительно к диаграммам вариантов использования ассоциация служит для обозначения специфической роли актера при его взаимодействии с отдельным вариантом использования. Другими словами, ассоциация специфицирует семантические особенности взаимодействия актеров и вариантов использования в графической модели системы. На диаграмме вариантов использования, так же как и на других диаграммах, отношение ассоциации обозначается сплошной линией между актером и вариантом использования. Эта линия может иметь некоторые дополнительные обозначения, например имя и кратность.



В контексте диаграммы вариантов использования отношение ассоциации между актером и вариантом использования может указывать на то, что актер инициирует соответствующий вариант использования. Такого актера называют главным. В других случаях подобная ассоциация может указывать на актера, которому предоставляется справочная информация о результатах функционирования моделируемой системы. Таких актеров часто называют второстепенными.

Включение (include) в языке UML *— это разновидность отношения зависимости между базовым вариантом использования и его специальным случаем. При этом отношением зависимости (dependency) является такое отношение между двумя элементами модели, при котором изменение одного элемента (независимого) приводит к изменению другого элемента (зависимого).*

Так, например, отношение включения, направленное от варианта использования «Предоставление кредита в банке» к варианту использования «Проверка платежеспособности клиента», указывает на то, что каждый экземпляр первого варианта использования всегда включает в себя функциональное поведение или выполнение второго варианта использования. В этом смысле поведение второго варианта использования является частью поведения первого варианта использования на данной диаграмме. *Графически данное отношение обозначается как отношение зависимости в форме пунктирной линии со стрелкой, направленной от базового варианта использования к включаемому варианту использования. При этом данная линия помечается стереотипом* ≪*include*≫ (рис. 1.22).

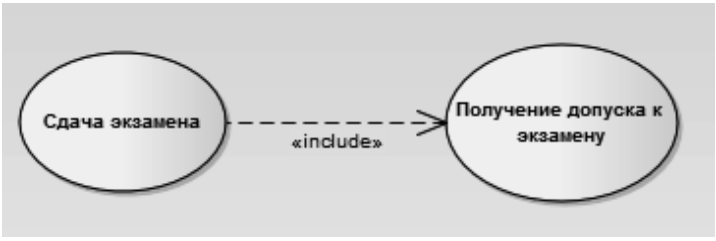


Рис. 1.22 – Пример графического изображения отношения включения между вариантами использования

Семантика этого отношения определяется следующим образом. Процесс выполнения базового варианта использования включает в себя как собственное подмножество последовательность действий, которая определена для включаемого варианта использования. При этом выполнение включаемой последовательности действий происходит всегда при инициировании базового варианта использования. Так, событию «Сдача экзамена» действительно всегда предшествует событие «Получение допуска к экзамену».

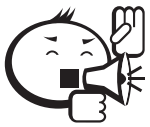
Один вариант использования может входить в несколько других вариантов, а также содержать в себе другие варианты. Включаемый вариант использования является независимым от базового варианта в том смысле, что он предоставляет последнему инкапсулированное поведение, детали реализации которого скрыты от последнего и могут быть легко перераспределены между несколькими включаемыми вариантами использования. Более того, базовый вариант зависит только от результатов выполнения включаемого в него варианта использования, но не от структуры включаемых в него вариантов.

Отношение расширения (extend) *определяет взаимосвязь базового варианта использования с другим вариантом использования, функциональное поведение которого задействуется базовым не всегда, а только при выполнении дополнительных условий.*

В языке UML отношение расширения является зависимостью, направленной к базовому варианту использования и соединенной с ним в так называемой точке расширения. Отношение расширения между вариантами использования обозначается как отношение зависимости в форме пунктирной линии со стрелкой, направленной от того варианта использования, который является расширением для базового варианта использования. Данная линия со стрелкой должна быть помечена стереотипом ≪extend≫ (рис. 1.23).

Рис. 1.23 – Пример графического изображения отношения расширения между вариантами использования

В изображенном фрагменте имеет место отношение расширения между базовым вариантом использования «Получение доп. баллов» и вариантами использования «Сдача всех лабораторных» и «Посещение всех лекций». Это означает, что свойства поведения первого варианта использования в некоторых случаях могут быть дополнены функциональностью второго варианта использования, а иногда и третьего. Для того чтобы эти расширения имели место, должно быть выполнено определенное логическое условие данного отношения расширения.



Отношение расширения позволяет моделировать таким образом, что один из вариантов использования должен присоединять к своему поведению дополнительное поведение, определенное для другого варианта использования. В то же время, данное отношение всегда предполагает проверку условия и ссылку на точку расширения в базовом варианте использования. Точка расширения определяет место в базовом варианте использования, в которое должно быть помещено расширение при выполнении соответствующего логического условия. При этом один из вариантов использования может быть расширением для нескольких базовых вариантов, а также иметь в качестве собственных расширений другие варианты. Базовый вариант использования не зависит от своих расширений.

Семантика отношения расширения определяется следующим образом. Если базовый вариант использования выполняет некоторую последовательность действий, которая определяет его поведение, и при этом имеется точка расширения на экземпляр другого варианта использования, которая является первой из всех точек расширения у базового варианта, то проверяется логическое условие данного отношения. Если это условие выполняется, исходная последовательность действий расширяется посредством включения действий другого варианта использования. Следует заметить, что условие отношения расширения проверяется лишь один раз — при первой ссылке на точку расширения, и если оно выполняется, то все расширяющие варианты использования вставляются в базовый вариант.

Два и более актера могут иметь общие свойства, т. е. взаимодействовать с одним и тем же множеством вариантов использования одинаковым образом. Такая общность свойств и поведения представляется в виде отношения обобщения с другим, возможно, абстрактным актером, который моделирует соответствующую общность ролей.

Графически отношение обобщения обозначается сплошной линией со стрелкой в форме не закрашенного треугольника, которая указывает на родительский вариант использования (рис. 1.24). Эта линия со стрелкой имеет специальное название — стрелка-обобщение.

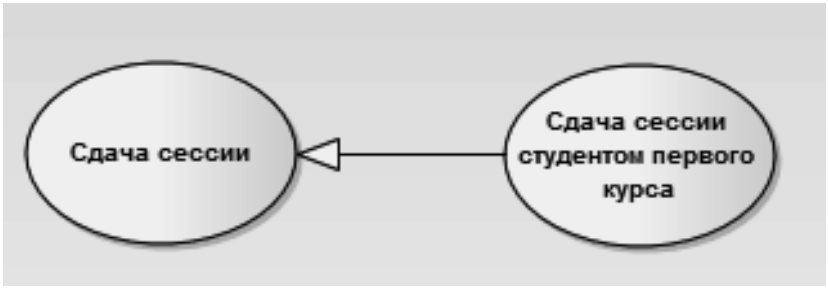


Рис. 1.24 – Пример графического изображения отношения обобщения между вариантами использования

В данном примере отношение обобщения указывает на то, что вариант использования «Сдача сессии студентом первого курса» — специальный случай варианта использования «Сдача сессии». Другими словами, первый вариант использования является специализацией второго варианта использования. При этом вариант использования «Сдача сессии» еще называют предком или родителем по отношению к варианту использования «Сдача сессии студентом первого курса», а последний вариант называют потомком по отношению к первому варианту использования. Следует подчеркнуть, что потомок наследует все свойства поведения своего родителя, а также может обладать дополнительными особенностями поведения.

Отношение обобщения между вариантами использования применяется в том случае, когда необходимо отметить, что дочерние варианты использования обладают всеми особенностями поведения родительских вариантов. При этом дочерние варианты использования участвуют во всех отношениях родительских вариантов. В свою очередь, дочерние варианты могут наделяться новыми свойствами поведения, которые отсутствуют у родительских вариантов использования, а также уточнять или модифицировать наследуемые от них свойства поведения [10].

Рекомендации по разработке диаграмм вариантов использования (пример)

В качестве примера на лабораторных работах будем рассматривать предметную область «Учёба в вузе».

Постановка проблемы: проектируется программное обеспечение для управления учебным процессом. Должны быть реализованы следующие возможности: ведение (редактирование) справочников «Студенты» и связанного с ним «Учебный процесс», формирование отчетов по успеваемости и оплате. Пользователями программной системы являются деканат и администратор БД.

Администратор входит в систему с правами администратора. Имеет возможность редактирования базы данных товаров (создавать и удалять справочники и записи). Следит за работоспособностью системы в целом.

Деканат входит в систему с правами пользователя. Заполняет и редактирует справочники. Формирует отчеты.

Создание диаграммы вариантов использования (Use Case)

Этап 1: При создании диаграммы вариантов использования начинаем с выделения действующих лиц (или актеров), участвующих во взаимодействии с нашей системой. В нашем случае можно выделить несколько типов действующих лиц, необходимых для отражения разных аспектов функционирования системы: *Студент, Деканат, Администратор БД, Родители студента, Преподаватель, Сотрудники.*

Кроме того, если мы хотим подчеркнуть общность между Администратором БД, Деканатом и Преподавателем (они являются сотрудниками одного и того же вуза), можно выделить действующее лицо Сотрудники и наладить связь обобщения между ними. Создание прямой связи (use) между пользователями, типа «Пользователь *A* — Пользователь *B*», в данном случае является ошибкой, так как не несет

в себе информации о взаимодействии пользователя с системой. Дабы не захламлять диаграмму избыточной информацией, правильным будет отнесение всех актеров в отдельную группу (рис. 1.25).

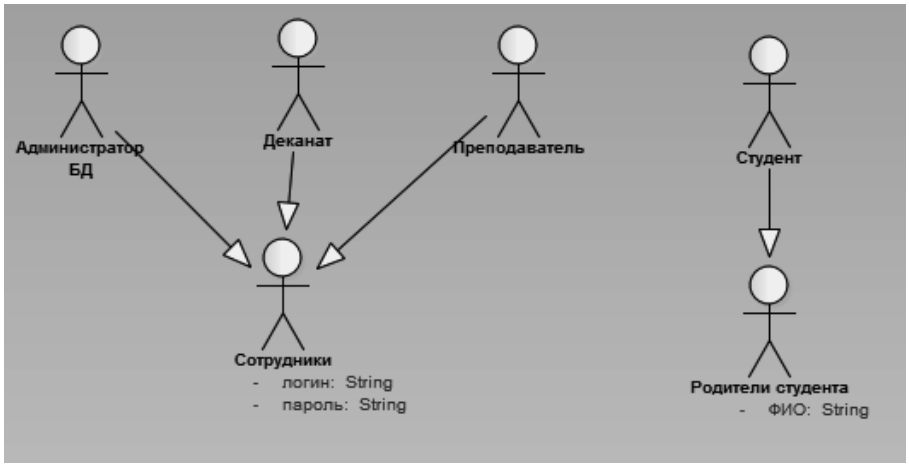


Рис. 1.25 – Содержимое пакета «Актеры»

Этап 2: Следующим шагом является выделение возможностей (функций, вариантов использования), которые программа должна предоставлять данным пользователям. На основании общего описания системы (см. начало главы) можно выделить следующие возможности программы: *«Регистрация оплаты в БД», «Формирование отчета», «Регистрация учебного процесса в БД», «Создание записи», «Удаление записи»* и т. д.

Вариантами использования могут быть только те возможности, которые будут реализованы в нашей системе. Например, такие функции, как *«Администратор следит за работоспособностью системы»* или *«Преподаватель ставит оценку студенту в зачётную книжку»*, происходят без использования функций проектируемой системы, это лишь последствия или внешние действия, имеющие косвенное отношение к системе. Поэтому подобные функции не будут включены в диаграмму вариантов использования (следует оговориться, что данные функции не могут быть включены при описании программного продукта, однако вполне могут использоваться, например, при описании бизнес-процессов, важных для отражения логики функционирования нашей предметной области).

Старайтесь не вдаваться в детали методов реализации, которые будут использоваться при реализации данного программного продукта. Например, такие действия, как *«Осуществить транзакцю для записи справочника в БД»* или *«Проверить логин и пароль»*, относятся уже скорее к технологии создания программы, чем к возможностям. Старайтесь опираться, прежде всего, на внешнее представление программы, поставьте себя на место конечных пользователей программы, которые «не знают», каким образом работает та или иная функция программы, однако вполне могут пользоваться всеми её возможностями.

Вариант использования это всегда какое-то действие. Не допускайте имен, обозначающих некие объекты, например *Отчет, Справочник*. Вместо этого используйте действия над этими объектами *«Формирование отчета», «Редактирование справочника»*.

При определении количества вариантов использования, размещаемых на диаграмме, придерживайтесь правила «золотой середины». Каждый вариант использования должен быть не слишком обобщенной, а вполне конкретной функцией. Например, вариант использования *«Ведение БД»* есть слишком общее понятие, в нашем случае это и *«Редактирование справочников»*, и *«Формирование отчетов»* и т. п. Однако большого количества функций также следует избегать, так как схема может потерять свою наглядность. В этом случае, некоторые варианты использования можно объединить. Например, в нашем случае *«Редактирование справочника»* и *«Заполнение поля справочника ФИО»*, *«Заполнение поля справочника Адрес»* нет смысла разделять в разные варианты использования, так как эти операции тесно взаимосвязаны и выполняются одновременно и в совокупности друг с другом.

Этап 3: Важным шагом является создание связей между Актерами и вариантами использования. Ассоциация *«Пользователь-Вариант использования»* отображает связь использования. Обратная ассоциация *«Вариант использования-пользователь»* показывает, что *«Вариант использования»* при инициализации передает некоторую информацию пользователю.

Не забывайте о том, что каждый пользователь на диаграмме представляет собой «Роль», которую играет тот или иной внешний объект. Причем один и тот же человек может играть несколько ролей в системе. Например, *«Преподаватель Иванов И. И.»* может зайти в систему в качестве представителя деканата и исправить справочники или посмотреть текущую успеваемость студента и т. д., а может войти в качестве администратора и удалить учётную запись студента. В связи с этим ассоциаций между ролью *«Администратор БД»* и вариантами использования *«Редактирование справочников»* — не существует.

Этап 4: Последним этапом создания диаграммы является документирование объектов диаграммы. Документация на активный объект вносится в поле *Documentation*. В табл. 1.2 показан пример документации на объекты нашей диаграммы.

При документировании пользуйтесь следующими соображениями:

- Пользователь — это внешний объект нашей системы. Поясните, что он собой представляет и какую роль играет в нашей системе.
- Вариант использования — это функция системы. Поясните, какие действия выполняет данная функция, какие возможности она реализует.
- Ассоциация — показывает, каким образом данный Пользователь использует данную Функцию. Учитывайте, что разные пользователи могут по-разному использовать один и тот же вариант использования.

Таблица 1.2 – Документация на объекты диаграммы Use Case

Объект	Документация
Действующие лица	
Сотрудники	Работники вуза, обладающие доступом к системе управления учебным процессом
Администратор БД	Управляет доступом пользователей к системе, следит за ее работоспособностью, отвечает за структуру и целостность БД
Деканат	Работает с преподавателями, заполняя по их сведениям БД информацией по текущей успеваемости студента. Работает с родителями, получая от них оплату за обучение студента и регистрацию данной информации в БД
Варианты использования	
Вход в систему	Проверка учетной записи пользователя — логина и пароля при входе в систему. Если пользователь существует и пароль верный, то определяется роль пользователя — «Деканат» или «Администратор БД» и осуществляется загрузка соответствующих меню и прав на изменение метаданных
Редактирование справочников	Добавление, удаление, корректировка элементов справочника «Студент» и «Учебный план», сохранение данных в БД
Формирование отчета	Вывод сформированного отчета на экран с целью просмотра или печати
Ассоциации	
Администратор — Создание учетной записи.	Использует для добавления, удаления, изменения учетных записей пользователей системы
Деканат — Редактирование справочников	Использует для добавления, удаления, изменения элементов справочника
Менеджер — Формирование счет-фактуры	Использует для формирования и просмотра счета по выбранным товарам
Родители студента — Оплата учёбы	Использует для передачи деканату информации о сумме и типе оплаты

Окончательный вид диаграммы прецедентов для проектируемой системы показан на рис. 1.26–1.30.

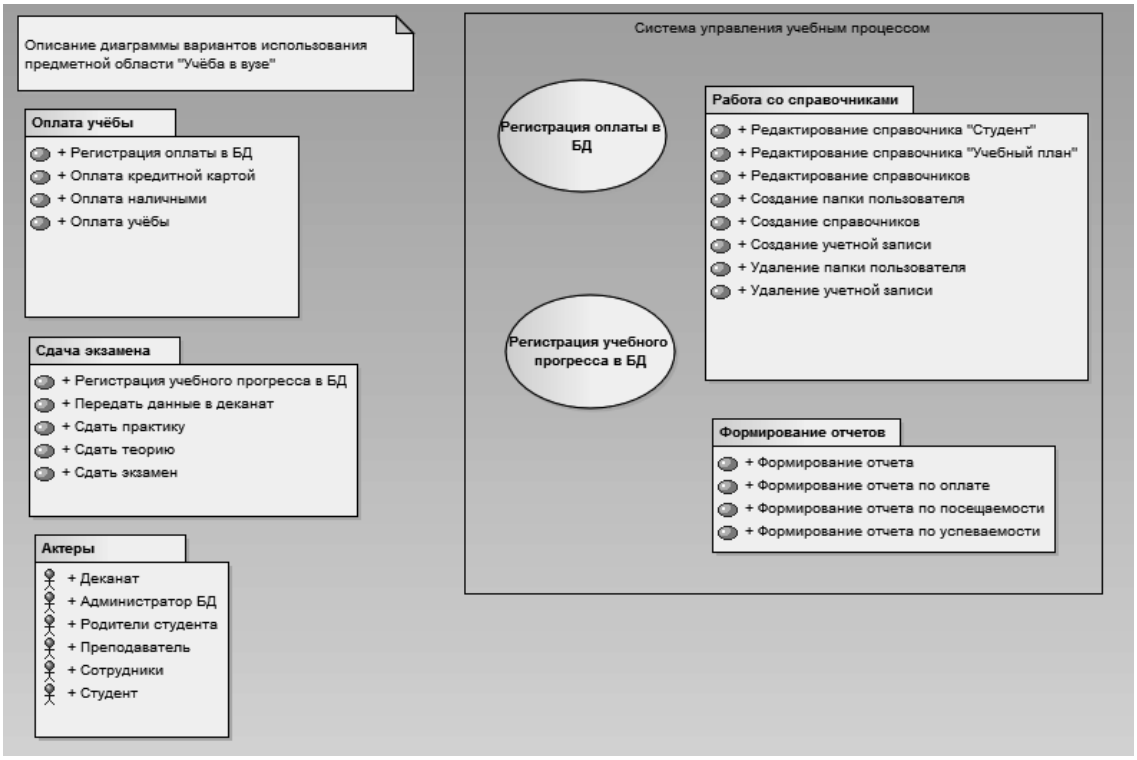


Рис. 1.26 – Диаграмма вариантов использования (общий вид)

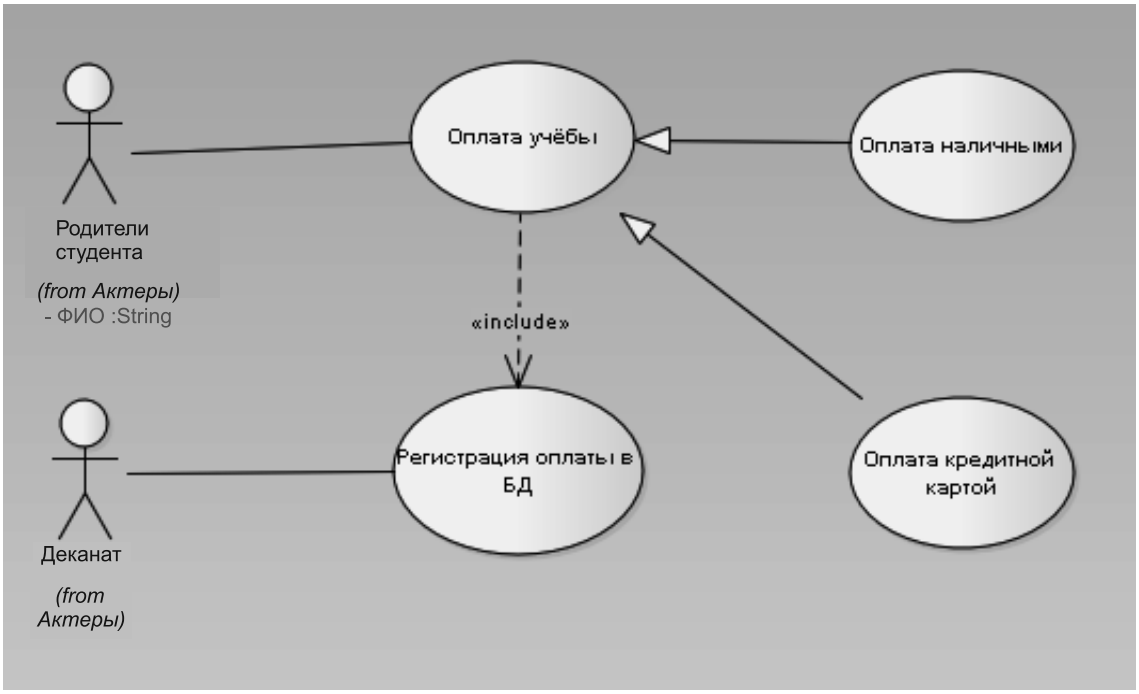


Рис. 1.27 – Диаграмма вариантов использования «Оплата учёбы»

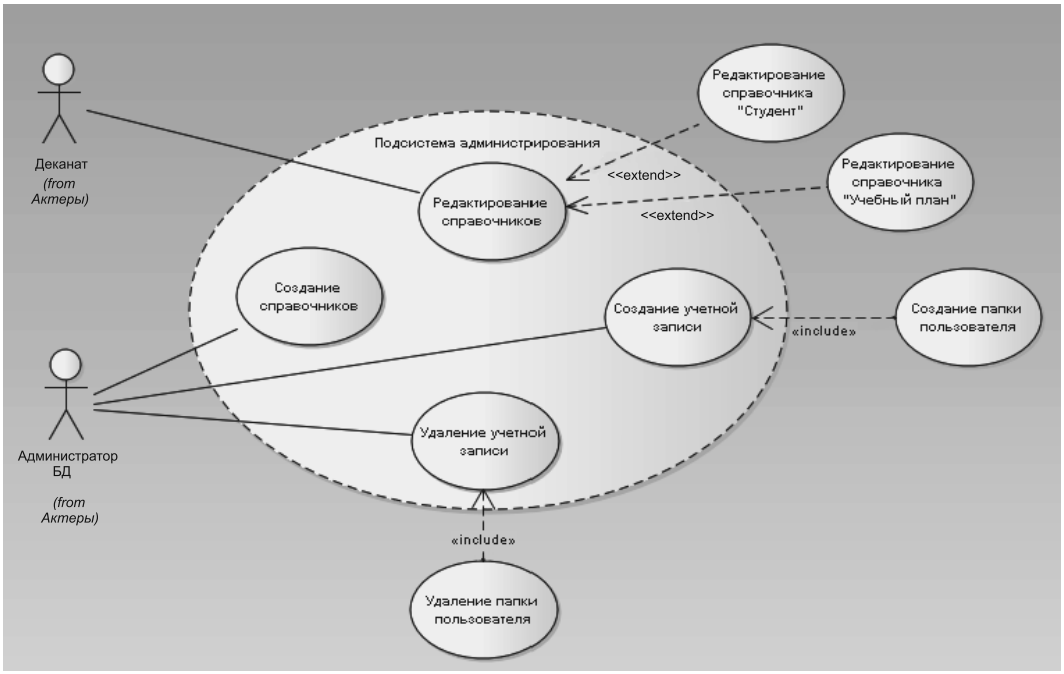


Рис. 1.28 – Диаграмма вариантов использования «Работа со справочниками»

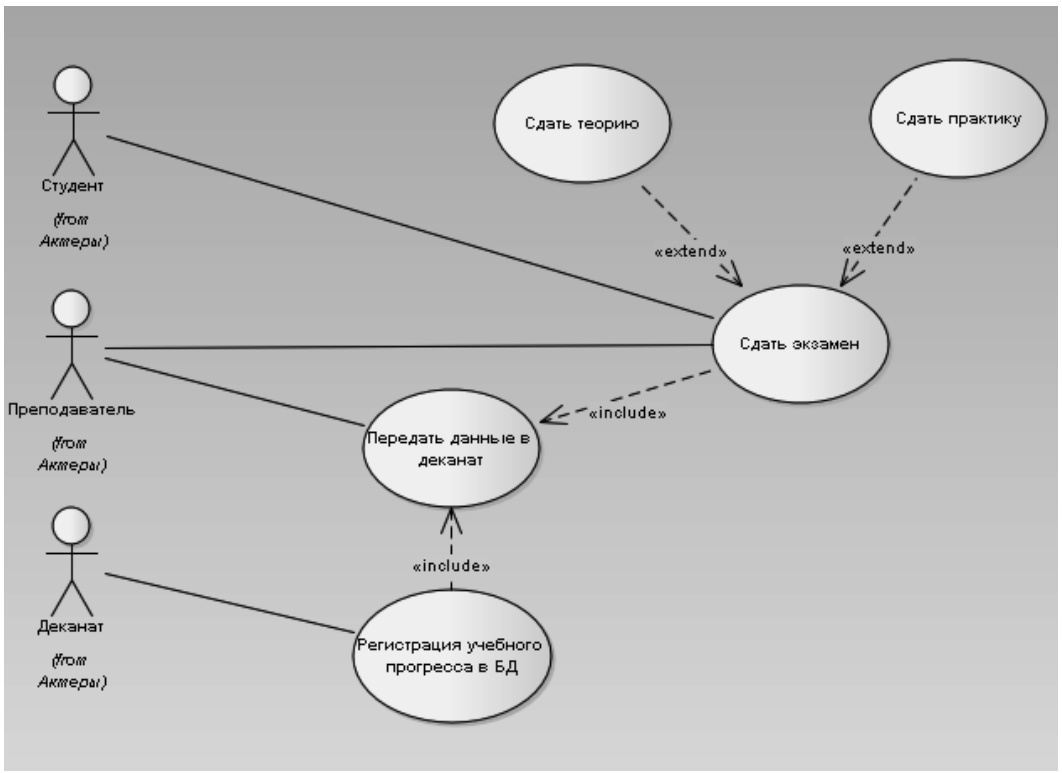


Рис. 1.29 – Диаграмма вариантов использования «Сдача экзамена»

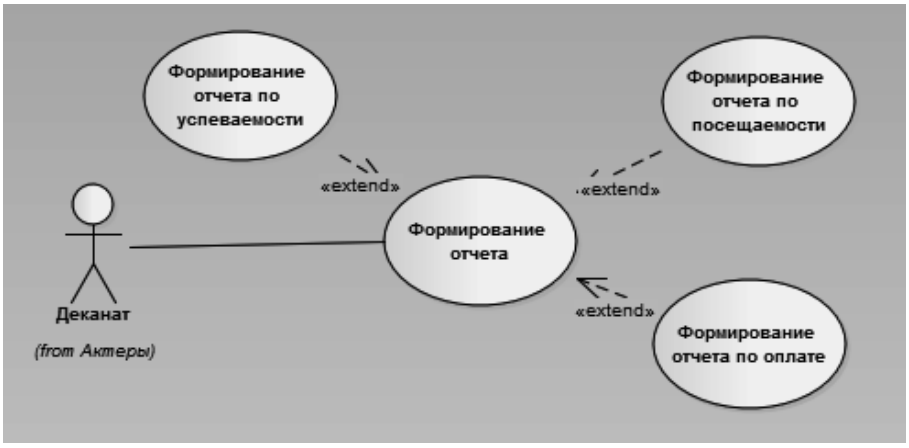


Рис. 1.30 – Диаграмма вариантов использования «Формирование отчётов»

Чтобы не потерять работу, сохраните проект (*File/Save*) или нажмите <*Ctrl+S*>.

Задание

Необходимо создать диаграмму прецедентов в инструментальной среде UML редактора, следуя описанным принципам работы с системой. Объектом автоматизации является процесс, выбранный из Приложения А.

Выбор варианта осуществляется по общим правилам с использованием следующей формулы:

$$V = (N * k) \operatorname{div} 100,$$

где: V — искомый номер варианта (при $V = 0$ выбирается максимальный вариант), N — общее количество вариантов по контрольной работе, k — значение двух последних цифр пароля (число в диапазоне 0..99), div — целочисленное деление (дробная часть отбрасывается).

Документы отчетности сдаются на проверку в электронной форме и включают в себя:

- файл модели;
- текстовый документ, содержащий описание потоков событий прецедентов моделируемой информационной системы.

Допускается документирование потоков событий и основных проектных решений в рамках среды UML редактора с использованием окна документирования. Помните, выходной файл данной модели будет использоваться и в остальных работах, поэтому заранее определите, какие диаграммы будут использоваться в данном проекте.

Контрольные вопросы по главе 1

1) Что такое пакет и подпакет?
2) Перечислите основные типы канонических диаграмм.
3) Перечислите три типа визуальных графических обозначений, которые важны с точки зрения заключенной в них информации.
4) Почему в диаграммах следует избегать противоречивой информации?
5) Какие типы диаграмм при построении модели следует строить в обязательном порядке?

Глава 2

ДИАГРАММА КЛАССОВ

Центральное место в методологии ООАП занимает разработка логической модели системы в виде диаграммы классов. Диаграмма классов отражает, в частности, различные взаимосвязи между отдельными сущностями предметной области, такими как объекты и подсистемы, а также описывает их внутреннюю структуру и типы отношений. На данной диаграмме не указывается информация о временных аспектах функционирования системы. С этой точки зрения диаграмма классов может служить дальнейшим развитием концептуальной модели проектируемой системы.

Диаграмма классов (class diagram) *— диаграмма языка UML, на которой представлена совокупность декларативных или статических элементов модели, таких как классы с атрибутами и операциями, а также связывающие их отношения.*

Диаграмма классов предназначена для представления статической структуры модели системы в терминологии классов объектно-ориентированного программирования. При этом диаграмма классов может содержать интерфейсы, пакеты, отношения и даже отдельные экземпляры классификаторов, такие как объекты и связи. Когда говорят о данной диаграмме, имеют в виду статическую структурную модель проектируемой системы, т. е. графическое представление таких структурных взаимосвязей логической модели системы, которые не зависят от времени.

2.1 Класс

Класс (class) *— абстрактное описание множества однородных объектов, имеющих одинаковые атрибуты, операции и отношения с объектами других классов.*

Графически класс в нотации языка UML изображается в виде прямоугольника, который дополнительно может быть разделен горизонтальными линиями на разделы или секции (рис. 2.1). В этих секциях могут указываться имя класса, атрибуты и операции класса.

Рис. 2.1 – Варианты графического изображения класса на диаграмме классов

На начальных этапах разработки диаграммы отдельные классы могут обозначаться простым прямоугольником, в котором должно быть указано имя соответствующего класса (рис. 2.1, *а*). По мере проработки отдельных компонентов диаграммы описание классов дополняется атрибутами (рис. 2.1, *б*) и операциями (рис. 2.1, *в*). Четвертая секция (рис. 2.1, *г*) не обязательна и служит для размещения дополнительной информации справочного характера, например об исключениях или ограничениях класса, сведениях о разработчике или языке реализации. Предполагается, что окончательный вариант диаграммы содержит наиболее полное описание классов, которые состоят из трех или четырех секций.

2.2 Имя класса

Имя класса должно быть уникальным в пределах пакета, который может содержать одну или несколько диаграмм классов. Имя указывается в самой верхней секции прямоугольника, поэтому она часто называется секцией имени класса. В дополнение к общему правилу именования элементов языка UML имя класса записывается по центру секции имени полужирным шрифтом и должно начинаться с заглавной буквы. Рекомендуется в качестве имен классов использовать существительные, записанные по практическим соображениям без пробелов. Необходимо помнить, что имена классов образуют словарь предметной области при ООАП.

В секции имени класса могут также находиться стереотипы или ссылки на стандартные шаблоны, от которых образован данный класс и, соответственно, от которых он наследует атрибуты и операции. В этой секции может также приводиться информация о разработчике данного класса и статус состояния разработки. Здесь также могут записываться и другие общие свойства этого класса, имеющие отношение к другим классам диаграммы или стандартным элементам языка UML.

Класс может иметь или не иметь экземпляров или объектов. В зависимости от этого в языке UML различают конкретные и абстрактные классы.

Конкретный класс (concrete class) *— класс, на основе которого могут быть непосредственно созданы экземпляры или объекты.*

Рассмотренные выше обозначения относятся к конкретным классам. От них следует отличать абстрактные классы.

Абстрактный класс (abstract class) *— класс, который не имеет экземпляров или объектов.*

Для обозначения имени абстрактного класса используется наклонный шрифт (*курсив*). В языке UML принято общее соглашение о том, что любой текст, относящийся к абстрактному элементу, записывается курсивом. Это имеет принципиальное значение, поскольку является семантическим аспектом описания абстрактных элементов языка UML.

В некоторых случаях необходимо явно указать, к какому пакету относится тот или иной класс. Для этой цели используется специальный символ разделитель — двойное двоеточие — «::». Синтаксис строки имени класса в этом случае будет следующий: <Имя пакета>::< Имя класса >. Другими словами, перед именем класса должно быть явно указано имя пакета, к которому его следует отнести. Например, если определен пакет с именем *Факультет*, то класс *Группа* в этом банке может быть записан в виде: *Факультет::Группа*.

2.3 Атрибуты класса

Атрибут (attribute) *— содержательная характеристика класса, описывающая множество значений, которые могут принимать отдельные объекты этого класса.*

Атрибут класса служит для представления отдельного свойства или признака, который является общим для всех объектов данного класса. Атрибуты класса записываются во второй сверху секции прямоугольника класса. Эту секцию часто называют секцией атрибутов.

В языке UML принята определенная стандартизация записи атрибутов класса, которая подчиняется некоторым синтаксическим правилам. Каждому атрибуту класса соответствует отдельная строка текста, которая состоит из квантора видимости атрибута, имени атрибута, его кратности, типа значений атрибута и, возможно, его исходного значения. Общий формат записи отдельного атрибута класса следующий:

<квантор видимости> <имя атрибута> [кратность] : <тип атрибута> = = <исходное значение> строка-свойство.

Видимость (visibility) *— качественная характеристика описания элементов класса, характеризующая потенциальную возможность других объектов модели оказывать влияние на отдельные аспекты поведения данного класса.*

Видимость в языке UML специфицируется с помощью квантора видимости (visibility), который может принимать одно из 4-х возможных значений и отображаться при помощи специальных символов.

Символ «+» — обозначает атрибут с областью видимости типа общедоступный (public). Атрибут с этой областью видимости доступен или виден из любого другого класса пакета, в котором определена диаграмма.

Символ «#» — обозначает атрибут с областью видимости типа защищенный (protected). Атрибут с этой областью видимости недоступен или невиден для всех классов, за исключением подклассов данного класса.

Символ «–» — обозначает атрибут с областью видимости типа закрытый (private). Атрибут с этой областью видимости недоступен или невиден для всех классов без исключения.

И, наконец, символ «~» — обозначает атрибут с областью видимости типа пакетный (package). Атрибут с этой областью видимости недоступен или невиден для всех классов за пределами пакета, в котором определен класс-владелец данного атрибута.

Квантор видимости может быть опущен. Его отсутствие означает, что видимость атрибута не указывается. Эта ситуация отличается от принятых по умолчанию соглашений в традиционных языках программирования, когда отсутствие квантора видимости трактуется как public или private. Однако вместо условных графических обозначений можно записывать соответствующее ключевое слово: *public, protected, private, package.*

Имя атрибута представляет собой строку текста, которая используется в качестве идентификатора соответствующего атрибута и поэтому должна быть уникальной в пределах данного класса. Имя атрибута — единственный обязательный элемент синтаксического обозначения атрибута, должно начинаться со строчной (малой) буквы и не должно содержать пробелов.

Кратность (multiplicity) *— спецификация области значений допустимой мощности, которой могут обладать соответствующие множества.*

Кратность атрибута характеризует общее количество конкретных атрибутов данного типа, входящих в состав отдельного класса. В общем случае кратность записывается в форме строки текста из цифр в квадратных скобках после имени соответствующего атрибута, при этом цифры разделяются двумя точками: *[нижняя граница..верхняя граница]*, где нижняя и верхняя границы положительные целые числа. Каждая такая пара служит для обозначения отдельного замкнутого интервала целых чисел, у которого нижняя (верхняя) граница равна значению нижней границы (верхней). В качестве верхней границы может использоваться специальный символ «*» (звездочка), который означает произвольное положительное целое число, т. е. неограниченное сверху значение кратности соответствующего атрибута.

Интервалов кратности для отдельного атрибута может быть несколько. В этом случае их совместное использование соответствует теоретико-множественному объединению соответствующих интервалов. Значения кратности из интервала следуют в монотонно возрастающем порядке без пропуска отдельных чисел, лежащих между нижней и верхней границами. При этом придерживаются следующего правила: соответствующие нижние и верхние границы интервалов включаются в значение кратности.

Если в качестве кратности указывается единственное число, то кратность атрибута принимается равной данному числу. Если же указывается единственный знак «*», то это означает, что кратность атрибута может быть произвольным положительным целым числом или нулем. В языке UML кратность широко используется также для задания ролей ассоциаций, составных объектов и значений атрибутов. Если кратность атрибута не указана, то по умолчанию в языке UML принимается ее значение, равное *[1..1]*, т. е. в точности 1.

Тип атрибута представляет собой выражение, семантика которого определяется некоторым типом данных, определенным в пакете Типы данных языка UML или самим разработчиком. В нотации UML тип атрибута иногда определяется в зависимости от языка программирования, который предполагается использовать для реализации данной модели. В простейшем случае тип атрибута указывается строкой текста, имеющей осмысленное значение в пределах пакета или модели, к которым относится рассматриваемый класс.

Исходное значение служит для задания начального значения соответствующего атрибута в момент создания отдельного экземпляра класса. Здесь необходимо придерживаться правила принадлежности значения типу конкретного атрибута. Если исходное значение не указано, то значение соответствующего атрибута не определено на момент создания нового экземпляра класса. С другой стороны, конструктор объекта может переопределять исходное значение в процессе выполнения программы, если в этом возникает необходимость.

При задании атрибутов могут быть использованы дополнительные синтаксические конструкции — это подчеркивание строки атрибута, пояснительный текст в фигурных скобках и косая черта перед именем атрибута. Подчеркивание строки атрибута означает, что соответствующий атрибут общий для всех объектов данного класса, т. е. его значение у всех создаваемых объектов одинаковое (аналог ключевого слова *static* в некоторых языках программирования).

Пояснительный текст в фигурных скобках может означать две различные конструкции. Если в этой строке имеется знак равенства, то вся запись *строка-свойство* служит для указания дополнительных свойств атрибута, которые мо-

гут характеризовать особенности изменения значений атрибута в ходе выполнения программы. Фигурные скобки как раз и обозначают фиксированное значение соответствующего атрибута для класса в целом, которое должны принимать все вновь создаваемые экземпляры класса без исключения. Это значение принимается за исходное значение атрибута, которое не может быть переопределено в последующем. Отсутствие *строки-свойства* по умолчанию трактуется так, что значение соответствующего атрибута может быть изменено в программе.

Знак «/» перед именем атрибута указывает на то, что данный атрибут является производным от некоторого другого атрибута этого же класса.

Производный атрибут (derived element) *— атрибут класса, значение которого для отдельных объектов может быть вычислено посредством значений других атрибутов этого же объекта.*

2.4 Операции класса

Операция (operation) *— это сервис, предоставляемый каждым экземпляром или объектом класса по требованию своих клиентов, в качестве которых могут выступать другие объекты, в том числе и экземпляры данного класса.*

Операции класса записываются в третьей сверху секции прямоугольника класса, которую часто называют секцией операций. Совокупность операций характеризует функциональный аспект поведения всех объектов данного класса. Запись операций класса в языке UML также стандартизована и подчиняется определенным синтаксическим правилам. При этом каждой операции класса соответствует отдельная строка, которая состоит из квантора видимости операции, имени операции, выражения типа возвращаемого операцией значения, и, возможно, строка-свойство данной операции. Общий формат записи отдельной операции класса следующий:

<квантор видимости><имя операции>(список параметров): <выражение типа возвращаемого значения>строка-свойство

Квантор видимости, как и в случае атрибутов класса, может принимать одно из четырех возможных значений и соответственно отображается при помощи специального символа либо ключевого слова.

- Символ «+» обозначает операцию с областью видимости типа общедоступный (public).
- Символ «#» обозначает операцию с областью видимости типа защищенный (protected).
- Символ «–» используется для обозначения операции с областью видимости типа закрытый (private).
- Символ «~» используется для обозначения операции с областью видимости типа пакетный (package).

Квантор видимости для операции может быть опущен. В этом случае его отсутствие просто означает, что видимость операции не указывается. Вместо условных графических обозначений также можно записывать соответствующее ключевое слово: *public, protected, private, package*.

Имя операции представляет собой строку текста, которая используется в качестве идентификатора соответствующей операции и поэтому должна быть уникальной в пределах данного класса. Имя операции — единственный обязательный элемент синтаксического обозначения операции, должно начинаться со строчной (малой) буквы и, как правило, записываться без пробелов.

Список параметров является перечнем разделенных запятой формальных параметров, каждый из которых, в свою очередь, может быть представлен в следующем виде:

<направление параметра> <имя параметра> : <выражение типа> = <значение параметра по умолчанию>.

Параметр (parameter) *— спецификация переменной операции, которая может быть изменена, передана или возвращена.*

Параметр может включать имя, тип, направление и значение по умолчанию. Направление параметра — есть одно из ключевых слов *in, out* или *inout* со значением *in* по умолчанию, в случае если вид параметра не указывается. Имя параметра есть идентификатор соответствующего формального параметра, при записи которого следуют правилам задания имен атрибутов. Выражение типа является спецификацией типа данных для допустимых значений соответствующего формального параметра. Наконец, значение по умолчанию в общем случае представляет собой некоторое конкретное значение для этого формального параметра.

Выражение типа возвращаемого значения также указывает на тип данных значения, которое возвращается объектом после выполнения соответствующей операции. Две точки и выражение типа возвращаемого значения могут быть опущены, если операция не возвращает никакого значения. Для указания нескольких возвращаемых значений данный элемент спецификации операции может быть записан в виде списка отдельных выражений.

Операция с областью действия на весь класс показывается подчеркиванием имени и строки выражения типа. В этом случае под областью действия операции понимаются все объекты этого класса. В этом случае вся строка записи операции подчеркивается.

Строка-свойство служит для указания значений свойств, которые могут быть применены к данной операции. Строка-свойство может отсутствовать, если свойства не специфицированы.

Список формальных параметров и тип возвращаемого значения не обязателен. Квантор видимости атрибутов и операций может быть указан в виде специального значка или символа, которые используются для графического представления моделей в инструментальном средстве. Еще раз следует напомнить, что имена операций, так же как атрибутов и параметров, записываются со строчной (малой) буквы, а их типы параметров — с заглавной (большой) буквы. При этом обязательной частью строки записи операции является наличие имени операции и круглых скобок.

2.5 Интерфейс

Интерфейс (interface) *— именованное множество операций, которые характеризуют поведение отдельного элемента модели.*

Интерфейс в контексте языка UML является специальным случаем класса, у которого имеются операции, но отсутствуют атрибуты. Для обозначения интерфейса используется специальный графический символ окружность или стандартный способ — прямоугольник класса со стереотипом ≪*interface*≫ (рис. 2.2).

В качестве имени интерфейса может использоваться существительное, которое характеризует соответствующую информацию или сервис, например: *«Датчик температуры», «Поле ввода», «Видеокамера»*. С учетом языка реализации модели имя интерфейса, как и имена других классов, рекомендуется записывать на английском и начинать с заглавной буквы «i», например ITemperatureSensor, ISecureInformation.

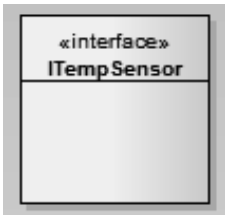


Рис. 2.2 – Пример графического изображения интерфейса на диаграмме классов

Интерфейсы на диаграмме служат для спецификации таких элементов модели, которые видимы извне, но их внутренняя структура остается скрытой от клиентов. Интерфейсы не могут содержать ни атрибутов, ни состояний, ни направленных ассоциаций. Они содержат только операции без указания особенностей их реализации. Формально интерфейс не только отделяет спецификацию операций системы от их реализации, но и определяет общие границы проектируемой системы. В последующем интерфейс может быть уточнен явным указанием тех операций, которые специфицируют отдельный аспект поведения системы.

Кроме внутреннего устройства классов важную роль при разработке проектируемой системы имеют различные отношения между классами, которые также могут быть изображены на диаграмме классов. Совокупность допустимых типов таких отношений строго фиксирована в языке UML и определяется самой семантикой этих отношений. Базовые отношения, изображаемые на диаграммах классов:

- отношение ассоциации (association relationship),
- отношение обобщения (generalization relationship),
- отношение агрегации (aggregation relationship),
- отношение композиции (composition relationship).

Каждое из этих отношений имеет собственное графическое представление, которое отражает семантический характер взаимосвязи между объектами соответствующих классов.

2.6 Отношение ассоциации

Ассоциация (association) *— семантическое отношение между двумя и более классами, которое специфицирует характер связи между соответствующими экземплярами этих классов.*

Отношение ассоциации соответствует наличию произвольного отношения или взаимосвязи между классами. Данное отношение, как уже отмечалось, обозначается сплошной линией со стрелкой или без нее и с дополнительными символами, которые характеризуют специальные свойства ассоциации. Ассоциации рассматривались при изучении элементов диаграммы вариантов использования, применительно к диаграммам классов, тем не менее семантика этого типа отношений значительно шире. В качестве дополнительных специальных символов могут использоваться имя ассоциации, символ навигации, а также имена и кратность классов-ролей ассоциации.

Имя ассоциации — необязательный элемент ее обозначения. Однако если оно задано, то записывается с заглавной буквы рядом с линией ассоциации. Отдельные классы ассоциации могут играть определенную роль в соответствующем отношении, на что явно указывает имя концевых точек ассоциации на диаграмме.

Наиболее простой случай данного отношения — *бинарная ассоциация* (binary association), которая служит для представления произвольного отношения между двумя классами. Она связывает в точности два различных класса и может быть ненаправленным (симметричным) или направленным отношением. Частный случай бинарной ассоциации — *рефлексивная ассоциация*, которая связывает класс с самим собой. Ненаправленная бинарная ассоциация изображается линией без стрелки. Для нее на диаграмме может быть указан порядок чтения классов с использованием значка в форме треугольника рядом с именем данной ассоциации.

В качестве простого примера ненаправленной бинарной ассоциации можно рассмотреть отношение между двумя классами — классом «ВУЗ» и классом «Студент» (рис. 2.3). Они связаны между собой бинарной ассоциацией «Учится», имя которой указано на рисунке рядом с линией ассоциации. Для данного отношения определен следующий порядок чтения следования классов — студент учится в вузе.

Рис. 2.3 – Графическое изображение ненаправленной бинарной ассоциации между классами

Направленная бинарная ассоциация изображается сплошной линией с простой стрелкой на одной из ее концевых точек. Направление этой стрелки указывает на то, какой класс является первым, а какой — вторым.

В качестве простого примера направленной бинарной ассоциации можно рассмотреть отношение между двумя классами — классом «Студент» и классом «Счет» (рис. 2.4). Они связаны между собой бинарной ассоциацией с именем «Имеет», для которой определен порядок следования классов. Это означает, что конкретный объект класса «Студент» всегда должен указываться первым при рассмотрении взаимосвязи с объектом класса «Счет». Другими словами, эти объекты классов образуют кортеж элементов, например <клиент, счет_1, счет_2,. . ., счет_*n*>.

Рис. 2.4 – Графическое изображение направленной бинарной ассоциации между классами

Выше мы говорили исключительно об ассоциациях, связывающих два класса. Однако возможны ситуации, когда необходимо показать взаимосвязь большего количества классов. Так, *тернарная ассоциация* связывает отношением три класса. Ассоциация более высокой арности называется *n-арной ассоциацией*.

***n*-арная ассоциация (*n*-ary association)** — *ассоциация между тремя и большим числом классов.*

Каждый экземпляр такой ассоциации представляет собой упорядоченный набор (кортеж), содержащий *n* экземпляров из соответствующих классов. Такая ассоциация связывает отношением более чем три класса, при этом класс может участвовать в ассоциации более чем один раз. Каждый экземпляр *n*-арной ассоциации представляет собой *n*-арный кортеж, состоящий из объектов соответствующих классов. В этом контексте бинарная ассоциация является частным случаем *n*-арной ассоциации, когда значение $n = 2$, но имеет собственное обозначение. Бинарная ассоциация — это специальный случай *n*-арной ассоциации.

Графически *n*-арная ассоциация обозначается ромбом, от которого ведут линии к символам классов данной ассоциации. Сам же ромб соединяется с символами классов сплошными линиями. Обычно линии проводятся от вершин ромба или от середины его сторон. Имя *n*-арной ассоциации записывается рядом с ромбом соответствующей ассоциации. Однако порядок классов в *n*-арной ассоциации, в отличие от порядка множеств в отношении, на диаграмме не фиксируется.

В качестве примера тернарной ассоциации можно рассмотреть отношение между тремя классами: «Студент», «ВУЗ» и «Проект ГПО». Данная ассоциация указывает на наличие отношения между этими тремя классами, которое может представлять информацию о проектах ГПО, реализуемых в вузе, и о студентах, участвующих в выполнении отдельных проектов (рис. 2.5).

Класс может быть присоединен к линии ассоциации пунктирной линией. Это означает, что данный класс обеспечивает поддержку свойств соответствующей *n*-арной ассоциации, а сама *n*-арная ассоциация имеет атрибуты, операции и/или ассоциации. Другими словами, такая ассоциация является классом с соответству-

ющим обозначением в виде прямоугольника и самостоятельным элементом языка UML — ассоциативным классом.

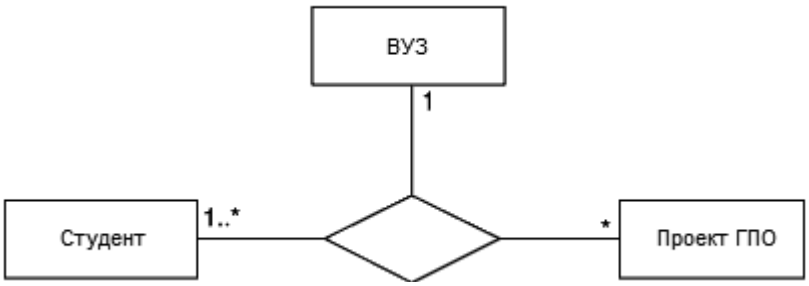


Рис. 2.5 – Графическое изображение тернарной ассоциации между тремя классами

Класс ассоциация (association class) *— модельный элемент, который одновременно является ассоциацией и классом. Ассоциативный класс может рассматриваться как ассоциация, которая обладает свойствами класса, или как класс, имеющий также свойства ассоциации.*

Как уже упоминалось, отдельный класс в ассоциации может играть определенную роль в данной ассоциации. Эта роль может быть явно специфицирована на диаграмме классов. С этой целью в языке UML вводится в рассмотрение специальный элемент — концевая точка ассоциации или конец ассоциации, который графически соответствует точке соединения линии ассоциации с отдельным классом.

Конец ассоциации (association end) *— концевая точка ассоциации, которая связывает ассоциацию с классификатором.*

Конец ассоциации — часть ассоциации, но не класса. Каждая ассоциация может иметь два или больше концов ассоциации. Наиболее важные свойства ассоциации указываются на диаграмме рядом с этими элементами ассоциации и должны перемещаться вместе с ними. Одним из таких дополнительных обозначений является имя роли отдельного класса, входящего в ассоциацию.

Роль (role) *— имеющее имя специфическое поведение некоторой сущности, рассматриваемой в определенном контексте. Роль может быть статической или динамической.*

Имя роли представляет собой строку текста рядом с концом ассоциации для соответствующего класса. Она указывает на специфическую роль, которую играет класс, являющийся концом рассматриваемой ассоциации. Имя роли не обязательный элемент обозначений и может отсутствовать на диаграмме.

Следующий элемент обозначений — кратность ассоциации. Кратность относится к концам ассоциации и обозначается в виде интервала целых чисел, аналогично кратности атрибутов и операций классов, но без прямых скобок. Этот интервал записывается рядом с концом соответствующей ассоциации и означает потенциальное число отдельных экземпляров класса, которые могут иметь место, когда остальные экземпляры или объекты классов фиксированы.

Так, для примера (рис. 2.5) кратность «1» для класса «ВУЗ» означает, что каждый студент может учиться только в одном вузе. Кратность «1..*» для класса «Студент» означает, что в каждом вузе могут обучаться несколько студентов, общее число которых заранее неизвестно и ничем не ограничено. Вместо кратности «1..*» нельзя записать только символ «*», поскольку последний означает кратность «0..*». Для данного примера это означало бы, что отдельные вузы могут совсем не иметь студентов.

Имя ассоциации рассматривается в качестве отдельного атрибута у соответствующих классов ассоциаций и может быть указано на диаграмме явным способом в определенной секции прямоугольника класса.

Ассоциация является наиболее общей формой отношения в языке UML. Все другие типы рассматриваемых отношений можно считать частным случаем данного отношения. Однако важность выделения специфических семантических свойств и дополнительных характеристик для других типов отношений обусловливают необходимость их самостоятельного изучения при построении диаграмм классов. Поскольку эти отношения имеют специальные обозначения и относятся к базовым понятиям языка UML, следует перейти к их последовательному рассмотрению.

2.7 Отношение обобщения

Отношение обобщения является обычным таксономическим отношением или отношением классификации между более общим элементом (родителем или предком) и более частным или специальным элементом (дочерним или потомком).

Обобщение (generalization) *— таксономическое отношение между более общим понятием и менее общим понятием.*

Менее общий элемент модели должен быть согласован с более общим элементом и может содержать дополнительную информацию. Данное отношение используется для представления иерархических взаимосвязей между различными элементами языка UML, такими как пакеты, классы, варианты использования.

Применительно к диаграмме классов данное отношение описывает иерархическое строение классов и наследование их свойств и поведения.

Наследование (inheritance) *— специальный концептуальный механизм, посредством которого более специальные элементы включают в себя структуру и поведение более общих элементов.*

Согласно одному из главных принципов методологии ООАП — наследованию, класс-потомок обладает всеми свойствами и поведением класса-предка, а также имеет собственные свойства и поведение, которые могут отсутствовать у класса-предка.

Родитель, предок (parent) *— в отношении обобщения более общий элемент. Потомок (child) — специализация одного из элементов отношения обобщения, называемого в этом случае родителем.*

На диаграммах отношение обобщения обозначается сплошной линией с треугольной стрелкой на одном из концов (рис. 2.6). Стрелка указывает на более общий класс (класс-предок или суперкласс), а ее начало — на более специальный класс (класс-потомок или подкласс).

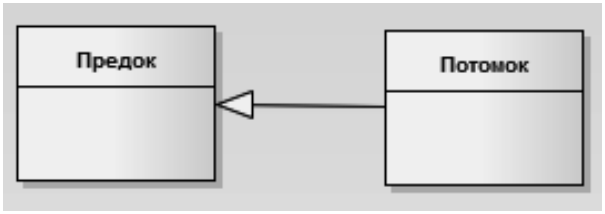


Рис. 2.6 – Графическое изображение отношения обобщения в языке UML

От одного класса-предка одновременно могут наследовать несколько классов-потомков, что отражает таксономический характер данного отношения. В этом случае на диаграмме классов для подобного отношения обобщения указывается несколько линий со стрелками.

Например, класс «ЭВМ» (курсив обозначает абстрактный класс) может выступать в качестве суперкласса для подклассов, соответствующих конкретным вычислительным машинам, таким как: «Лэптоп», «Сервер», «Рабочая станция» и другим. Это может быть представлено графически в форме диаграммы классов следующего вида (рис. 2.7).

С целью упрощения обозначений на диаграмме классов и уменьшения числа стрелок-треугольников и совокупности линий, обозначающих одно и то же отношение обобщения, может быть просто изображена стрелка. В этом случае отдельные линии сходятся к стрелке, которая имеет с этими линиями единственную точку пересечения (рис. 2.8).

Графическое изображение отношения обобщения по форме соответствует графу специального вида, а именно — иерархическому дереву. Как нетрудно заметить, в этом случае класс-предок является корнем дерева, а классы-потомки — его листьями. Отличие заключается в возможности указания на диаграмме классов дополнительной семантической информации, которая может отражать различные теоретико-множественные характеристики данного отношения. При этом класс-предок на диаграмме может занимать произвольное положение относительно своих классов-потомков, определяемое лишь соображениями удобства.

В дополнение к простой стрелке обобщения может быть присоединена строка текста, указывающая на специальные свойства этого отношения в форме огра-

ничения. Этот текст будет относиться ко всем линиям обобщения, которые идут к классам-потомкам. Поскольку отмеченное свойство касается всех подклассов данного отношения, именно поэтому спецификация этого свойства осуществляется в форме ограничения, которое должно быть записано в фигурных скобках.

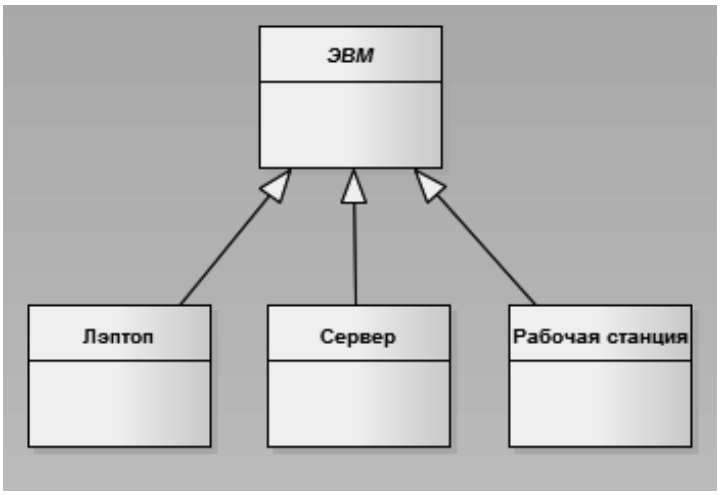


Рис. 2.7 – Пример графического изображения отношения обобщения для нескольких классов-потомков

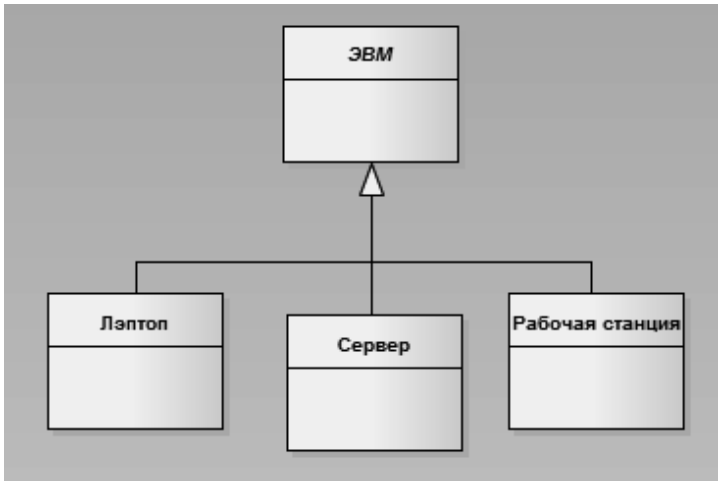


Рис. 2.8 – Альтернативный вариант графического изображения отношения обобщения классов для случая объединения отдельных линий

В качестве ограничений могут быть использованы следующие ключевые слова языка UML:

- *{complete}* — означает, что в данном отношении обобщения специфицированы все классы-потомки и других классов-потомков у данного класса-предка быть не может;
- *{incomplete}* — означает случай, противоположный первому. А именно предполагается, что на диаграмме указаны не все классы-потомки. В последующем возможно разработчик восполнит их перечень, не изменяя уже построенную диаграмму;

- *{disjoint}* — означает, что классы-потомки не могут содержать объектов, одновременно являющихся экземплярами двух или более классов;
- *{overlapping}* — случай, противоположный предыдущему. А именно предполагается, что отдельные экземпляры классов-потомков могут принадлежать одновременно нескольким классам.

С учетом дополнительного использования стандартного ограничения диаграмма классов (рис. 2.8) может быть уточнена (рис. 2.9).

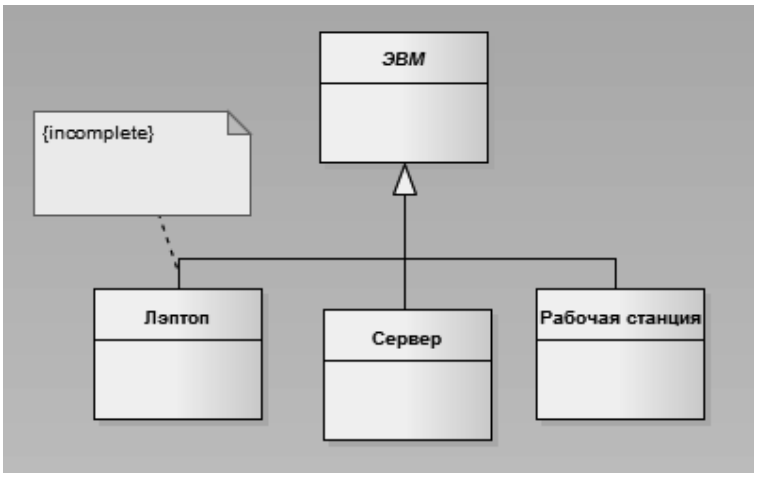


Рис. 2.9 – Вариант уточненного графического изображения отношения обобщения классов с использованием строки-ограничения

2.8 Отношение агрегации

Агрегация (aggregation) *— специальная форма ассоциации, которая служит для представления отношения типа «часть-целое» между агрегатом (целое) и его составной частью.*

Отношение агрегации имеет место между несколькими классами в том случае, если один из классов представляет собой сущность, которая включает в себя в качестве составных частей другие сущности. Данное отношение имеет фундаментальное значение для описания структуры сложных систем, поскольку применяется для представления системных взаимосвязей типа «часть-целое». Раскрывая внутреннюю структуру системы, отношение агрегации показывает, из каких элементов состоит система и как они связаны между собой.

С точки зрения модели отдельные части системы могут выступать в виде как элементов, так и подсистем, которые, в свою очередь, тоже могут состоять из подсистем или элементов. Таким образом, данное отношение по своей сути описывает декомпозицию или разбиение сложной системы на более простые составные части, которые также могут быть подвергнуты декомпозиции, если в этом возникнет необходимость.

Очевидно, что рассматриваемое в таком аспекте деление системы на составные части представляет собой иерархию, но принципиально отличную от той, которая порождается отношением обобщения. Отличие заключается в том, что части системы никак не обязаны наследовать ее свойства и поведение, поскольку являются самостоятельными сущностями. Более того, части целого обладают собственными атрибутами и операциями, которые существенно отличаются от атрибутов и операций целого.

Графически отношение агрегации изображается сплошной линией, один из концов которой представляет собой не закрашенный внутри ромб. Этот ромб указывает на тот класс, который представляет собой «целое» или класс-контейнер. Остальные классы являются его «частями» (рис. 2.10).

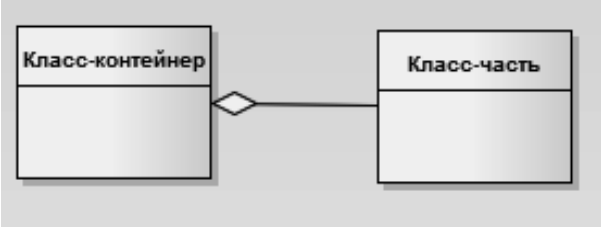


Рис. 2.10 – Графическое изображение отношения агрегации в языке UML

В качестве примера отношения агрегации можно рассмотреть взаимосвязь типа «часть-целое», которая имеет место между классом «Системный блок» и его составными частями: «Центральный процессор», «Материнская плата», «Оперативная память», «Жесткий диск» и «Видеокарта». Используя обозначения языка UML, компонентный состав системного блока можно представить в виде соответствующей диаграммы классов (рис. 2.11), которая в данном случае иллюстрирует отношение агрегации.

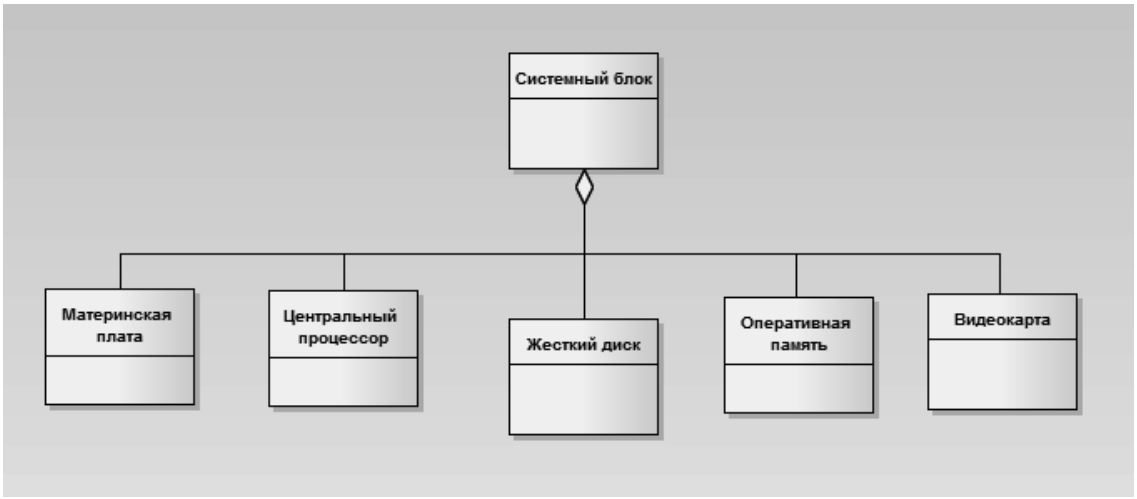


Рис. 2.11 – Диаграмма классов для иллюстрации отношения агрегации на примере системного блока ПК

2.9 Отношение композиции

Композиция (composition) *— разновидность отношения агрегации, при которой составные части целого имеют такое же время жизни, что и само целое. Эти части уничтожаются вместе с уничтожением целого.*

Отношение композиции — частный случай отношения агрегации. Это отношение служит для спецификации более сильной формы отношения «часть-целое», при которой составляющие части тесно взаимосвязаны с целым. Особенность этой взаимосвязи заключается в том, что части не могут выступать в отрыве от целого, т. е. с уничтожением целого уничтожаются и все его составные части.

Композит (composite) *— класс, который связан отношением композиции с одним или большим числом классов.*

Графически отношение композиции изображается сплошной линией, один из концов которой представляет собой закрашенный внутри ромб. Этот ромб указывает на тот класс, который представляет собой класс-композит. Остальные классы являются его «частями» (рис. 2.12).

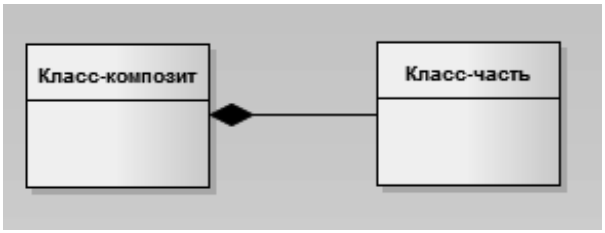


Рис. 2.12 – Графическое изображение отношения композиции в языке UML

Возможно, самым наглядным примером этого отношения является живая клетка в биологии, в отрыве от которой не могут существовать ее составные части. Другой практический пример — окно графического интерфейса программы, которое может состоять из строки заголовка, полос прокрутки, главного меню, рабочей области и строки состояния. Нетрудно заключить, что подобное окно представляет собой класс, а его составные элементы также являются отдельными классами. Последнее обстоятельство характерно для отношения композиции, поскольку отражает различные способы представления данного отношения.

Для отношений композиции и агрегации могут использоваться дополнительные обозначения, применяемые для отношения ассоциации. А именно могут указываться кратности отдельных классов, которые в общем случае не обязательны. Применительно к описанному выше примеру класс Окно программы является

классом-композитом, а взаимосвязи составляющих его частей могут быть изображены следующей диаграммой классов (рис. 2.13) [10].

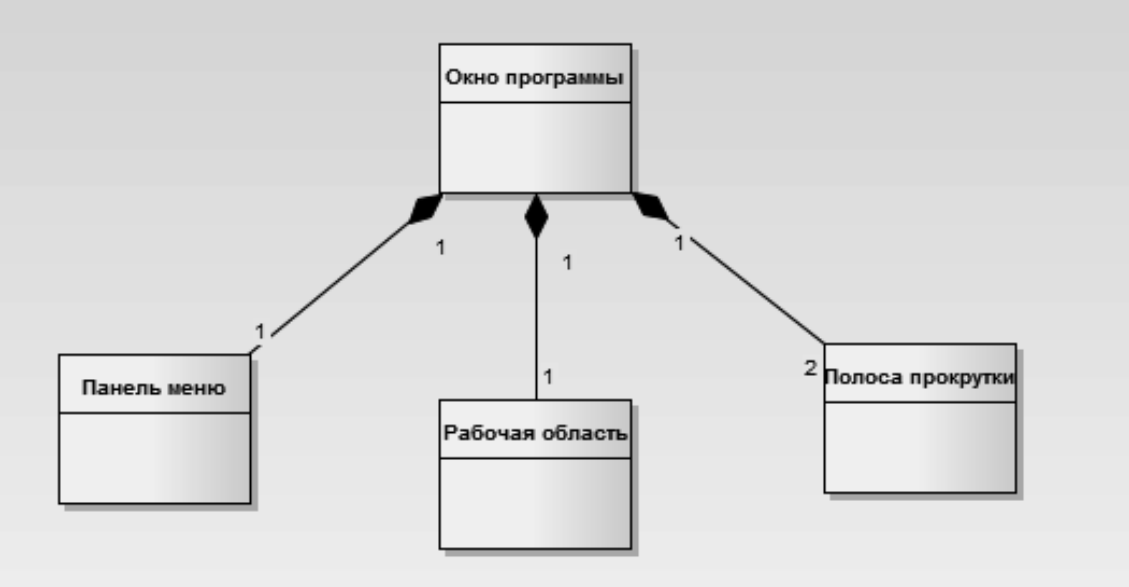


Рис. 2.13 – Диаграмма классов для иллюстрации отношения композиции на примере класса-композита Окно программы

Лабораторная работа №2

Цель работы

Целью данной работы является построение диаграммы классов.

Краткое изложение теоретической части

В UML диаграмма классов является типом диаграммы статической структуры. Она описывает структуру системы, показывая её классы, их атрибуты и операторы, а также взаимосвязи этих классов.

Взаимосвязь — это особый тип логических отношений между сущностями, показанных на диаграммах классов и объектов. В UML представлены следующие виды отношений:

1). *Ассоциация* — показывает, что объекты одной сущности (класса) связаны с объектами другой сущности. Существует пять различных типов ассоциации. Наиболее же распространёнными являются двунаправленная (бинарная) и однонаправленная. Например, классы «Рейс» и «Самолёт» связаны двунаправленной ассоциацией, а классы «Человек» и «Кофейный автомат» связаны однонаправленной. Двойные ассоциации (с двумя концами) представляются линией, соединяющей два классовых блока. Ассоциации в большей степени имеют более двух концов и представляются линиями, один конец которых идет к классовому блоку, а другой

к общему ромбику. В представлении однонаправленной ассоциации добавляется стрелка, указывающая на направление ассоциации. Ассоциации могут быть именованными, и тогда на концах представляющей её линии будут подписаны роли, принадлежности, индикаторы, мультипликаторы, видимости или другие свойства.

Агрегация — это разновидность ассоциации, при отношении между целым и его частями. Как тип ассоциации, агрегация может быть именованной. Агрегация не может включать сразу несколько классов. Агрегация встречается, когда один класс является коллекцией или контейнером других. Причём по умолчанию, агрегацией называют агрегацию по ссылке, т. е. когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет. Графически агрегация представляется пустым ромбиком на блоке класса и линией, идущей от этого ромбика к содержащемуся классу.

Композиция — более строгий вариант агрегации. Известна так же как агрегация по значению. Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено. Графически представляется как и агрегация, но с закрашенным ромбиком.

Различие между композицией и агрегацией заключается в следующем: целое композиции должно иметь мультипликатор кратности 0..1 или 1, что показывает, что часть является частью только одного целого, в агрегации же может быть любой мультипликатор. Приведём наглядный пример. Двигатель является частью машины, следовательно, здесь подходит композиция. В то же время, когда они представлены в базе данных, данная модель двигателя может быть у разных моделей машин, поэтому следует использовать агрегацию.

2). *Обобщение* (Generalization) показывает, что один из двух связанных классов (подтип) является более частной формой другого (надтипа), который называется обобщением первого. На практике это означает что любой экземпляр подтипа является также экземпляром надтипа. Например: животные — супертип млекопитающих, которые в свою очередь супертип приматов и так далее. Эта взаимосвязь легче всего описывается фразой «А — это Б» (приматы — это млекопитающие, млекопитающие — это животные). Графически обобщение представляется линией с пустым треугольником у супертипа. Обобщение также известно как наследование или «is a» взаимосвязь.

3). *Реализация* — отношение между двумя элементами модели, в котором один элемент (клиент) реализует поведение, заданное другим (поставщиком). Графически реализация представляется так же как и обощение, но с пунктирной линией.

4). *Зависимость* — это отношение использования, при котором изменение в спецификации одного влечёт за собой изменение другого, причем обратное не обязательно. Графически представляется пунктирной стрелкой, идущей от зависимого элемента к тому, от которого он зависит. Существует несколько именованных вариантов. Зависимость может быть между экземплярами, классами или экземпляром и классом.

5). *Уточнение отношений*. Уточнение имеет отношение к уровню детализации. Один пакет уточняет другой, если в нем содержатся те же самые элементы, но в более подробном представлении. Например, при написании книги вы наверняка начнете с формулировки предложения, в котором кратко будет представлено

содержание каждой главы. Предположим, что резюме к каждой главе в качестве отдельного элемента входит в пакет «Предложение». Допустим также, что «Завершённая книга» — это пакет, элементами которого являются законченные главы. В этом контексте пакет «Завершённая книга» является уточнением пакета «Предложение» [12].

Мощность отношения (мулитипликатор или кратность) означает число связей между каждым экземпляром класса (объектом) в начале линии с экземпляром класса в ее конце. Различают следующие типичные случаи (таб. 2.1):

Таблица 2.1 – Виды мощности отношения

Нотация	Название	Пример
0..1	Ноль или один	У студента может быть действительный пропуск в общежитие, а может не быть
1	Только один	У студента только одна мать
0 или *	Ноль или более	В течение учёбы студент может сдать много экзаменов, а может ни одного
1..*	Один или более	Студент спит минимум в одном месте

Работа с диаграммой классов в Enterprise Architect

В Enterprise Architect вы можете добавить диаграмму классов, стандартным способом — через Project Browser.

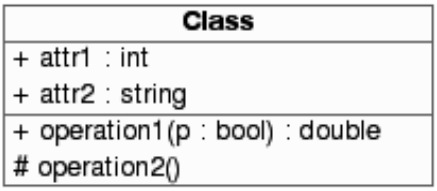


Рис. 2.14 – Класс в UML

Описание элементов управления в Enterprise Architect приведено в лабораторной работе №1. Функциональный смысл действий, которые пользователь может проделать — тот же, отличие заключается в расположении свойств по вкладкам. Для краткости, разберем наиболее применительные в диаграммах классов инструменты (таб. 2.2).

Пояснение к терминологии, характерной для объектно-ориентированного программирования (ООП) и встречающейся в описании элементов управления, используемых для создания диаграммы классов, приведено в Глоссарии.

Таблица 2.2 – Элементы управления диаграммы классов

	Имя	Назначение
	Class	Данный инструмент позволяет создать новый класс в диаграмме и модели. Класс — это установки структуры и шаблона поведения для некоторого множества реальных объектов, которые в дальнейшем будут определены в программе на основе данного шаблона. Класс — это некоторая абстракция реального мира. Когда эта абстракция принимает конкретное воплощение, она называется объектом. Класс в UML изображается как прямоугольник, разделенный на 3 части (рис. 2.14). В верхней части записывается название класса, в середине — атрибуты, в нижней части — операции
	Interface	Значок *Interface* позволяет создать интерфейсный объект, который указывает на видимые извне операции класса или компонента. Обычно интерфейс создается только для некоторых строго определенных классов или компонентов и предназначен скорее для логического отображения системы, но может присутствовать как на диаграмме классов, так и на диаграмме компонентов

Диаграмма классов является основой проектируемой системы и является, пожалуй, наиболее важной схемой проекта. Однако создание диаграммы классов — это, прежде всего, интуитивный процесс и успех его выполнения зависит во многом от интеллекта и опыта проектировщика. Не следует также ожидать, что, пройдя все этапы создания диаграммы, вы сразу же получите диаграмму классов, учитывающую все детали функционирования системы, позволяющую реализовать все возможности и т. д. Процесс описания классов системы является чаще всего итеративным процессом. Полученные диаграммы постоянно дорабатываются — зачастую оказывается, что некоторые классы следует добавить, а другие следует убрать из системы, меняется набор и параметры атрибутов классов, изменяются методы классов, связи и т. п.

Enterprise Architect позволяет устанавливать значительное количество свойств класса, которые, в том числе, влияют на генерацию кода класса, поэтому, чтобы лучше ориентироваться в дальнейших действиях, разберем вкладки окна спецификаций.

Вкладка General

Это окно позволяет задать главные свойства класса, такие как его имя, тип, определить стереотип класса и доступ к нему, когда класс находится в контейнере. Так же как и во всех других диаграммах, здесь можно задать документацию к классу.

Name	Предназначено для задания имени класса
Type	Предназначено для задания типа класса. В нашем случае — это «класс», но можно выбрать значение «параметризированный класс», «инстанцированный класс» и др.
Stereotype	Задает стереотип класса
Export Control (Scope/Visibility)	Предназначен для определения доступа к классу, когда он расположен в контейнере. При этом *+Public* определяет, что элемент виден вне контейнера, в котором он определен, и его можно импортировать в другие части создаваемой модели; *#Protected* — элемент доступен только для вложенных классов, классов с типом friends и собственно внутри класса; *–Private* обозначает защищенный элемент класса; *~Implementation* — элемент виден только в том контейнере, в котором определен

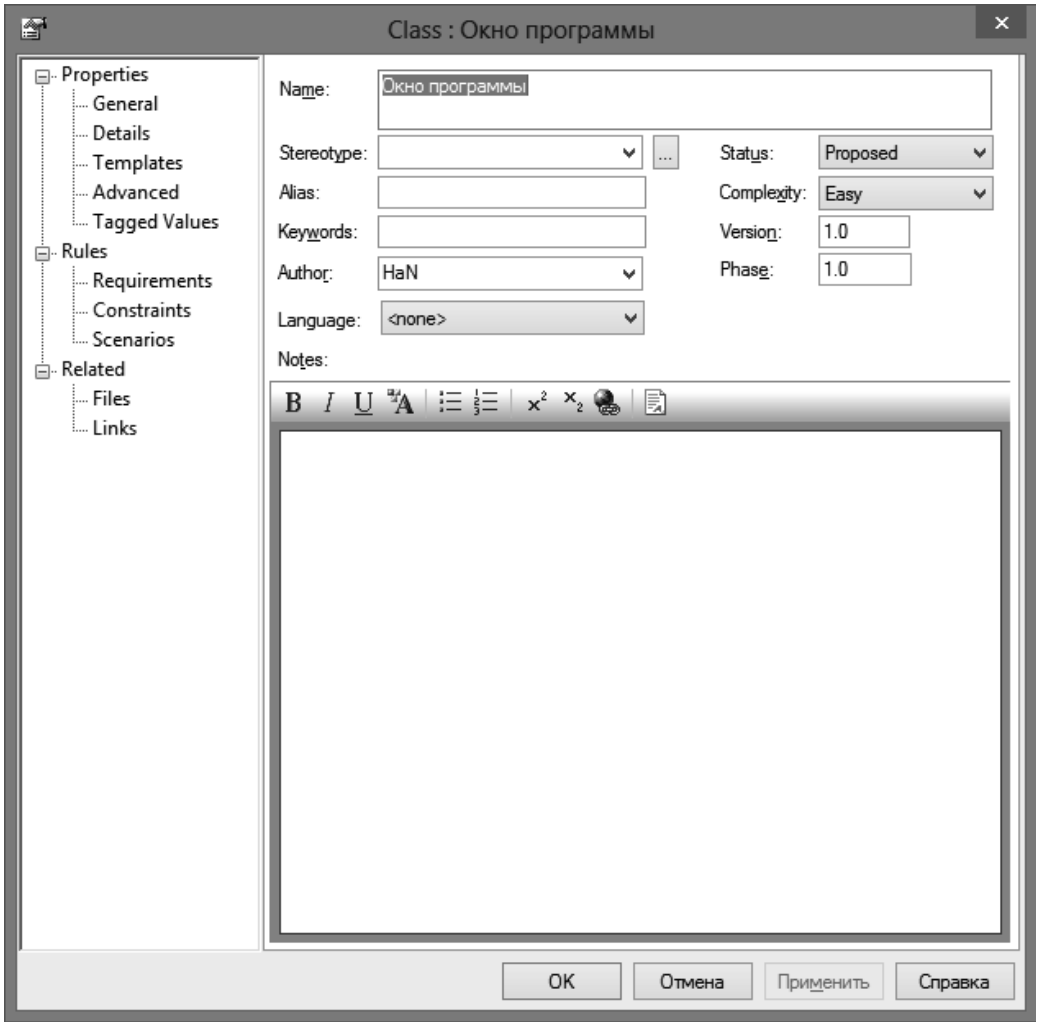


Рис. 2.15 – Окно свойств класса в Enterprise Architect

В окне свойств Enterprise Architect дополнительно можно определить текущее состояние разработки (Status), сложность (Complexity), псевдоним (Alias), язык реализации класса (Language), время жизни (Persistence) (рис. 2.15).

Вкладка Details (детализация)

Позволяет указывать дополнительные установки класса, такие как ожидаемое количество создаваемых объектов класса, ожидаемый расход оперативной памяти и т. д.

Cardinality (или Multiplicity)	Позволяет задать ожидаемое количество объектов, которые будут созданы на основе данного класса
Space	Показывает количество оперативной памяти, необходимой для создания объекта данного класса
Persistence	Определяет время жизни объекта класса. Если установлен флажок Persistent, то объект должен быть доступен в течение всей работы программы или для доступа других потоков или процессов
Concurrency	Обозначает поведение элемента в многопотоковой среде
Abstract	Обозначает, что класс является абстрактным

Вкладка Attributes (атрибуты)

Данная вкладка (в Enterprise Architect подэлемент вкладки Details) позволяет добавлять, удалять, редактировать атрибуты класса.

На данной вкладке представлен список атрибутов класса, который можно редактировать при помощи контекстного меню. Здесь пользователь может изменить название атрибута (*Name*), его тип (*Type*) и стереотип (*Stereotype*), задать начальное значение (*Initial value*) и тип доступа к атрибуту (*Export Control*).

Дополнительная вкладка *Detail* спецификаций атрибутов класса позволяет задать тип хранения атрибута в классе:

- *By Value* — по значению;
- *By Reference* — по ссылке;
- *Unspecified* — не указано.

Также пользователь может указать, что атрибут является *Static* (статическим) или *Derived* (производным).

Также можно определить видимость атрибута, а в Enterprise Architect дополнительно задать начальные/граничные значения, определить ограничения и прочие свойства, характерные для классов ООП.

Вкладка Operations (операции)

Вкладка *Operations* позволяет добавлять, удалять, редактировать операции класса.

На этой вкладке представлен список операций класса, который можно редактировать при помощи контекстного меню. Вкладка *General* спецификаций операции аналогична вкладке *General* атрибутов.

Вкладка *Detail* спецификаций операций позволяет устанавливать дополнительные свойства операции.

Arguments	Позволяет устанавливать список аргументов для операции с их типами и значениями по умолчанию
Exceptions	Позволяет задавать список исключений, которые могут быть вызваны операцией. Здесь можно ввести имя одного или нескольких классов, обрабатывающих исключительные состояния
Size	Позволяет задать размер памяти, требуемой для выполнения операции
Time	Позволяет задать время выполнения операции
Concurrency	Отражает для многопотоковой программы тип выполнения операции

Вкладки Parameters, Behavior, Advanced позволяют задавать дополнительные описания процессов подготовки и завершения операции, а также описание алгоритма операции.

Связи

При выставлении взаимоотношений между классами становится доступным функционал по определению параметров связей.

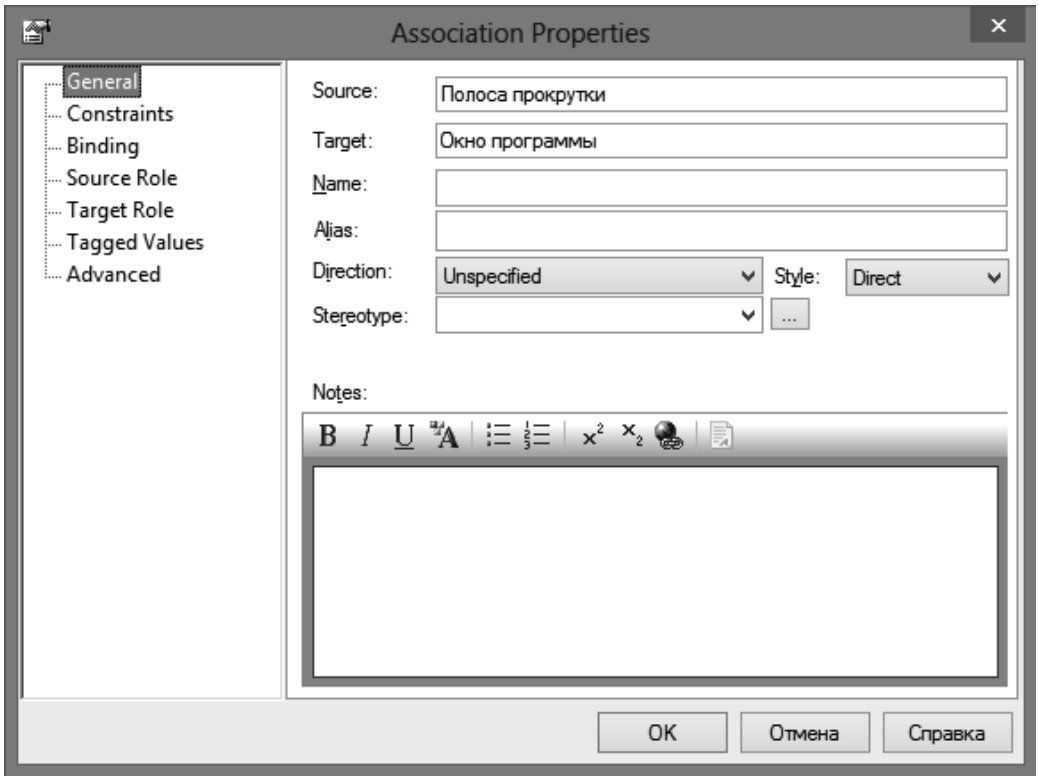


Рис. 2.16 – Окно свойств связи

На вкладке General возможно определить основные свойства, указать имя, стиль и направленность. Определить, будет ли связь идти прямо, либо же выглядеть в виде дерева.

Классы редко бывают изолированы, чаще всего они вступают в отношения друг с другом. Эти отношения показываются при помощи различного вида связей. Типы связей влияют на получаемый при генерации на основе диаграмм исходный код.

В диаграмме классов используются следующие виды связей:

- Unidirectional association (однонаправленная ассоциация);
- Dependency (зависимость);
- Association class (ассоциированный класс);
- Generalization (наследование);
- Realization (реализация).

Unidirectional Association ( )

Одна из важных и сложных типов связи, которая используется в диаграмме классов. Данная связь показывает, что объекты одного класса взаимодействуют с объектами другого класса. Ассоциации имеют направление, показывая отношение одного класса к другому. Каждая сущность, вовлечённая в данную связь, выполняет определённую роль, роли могут быть назначены.

При нажатии правой кнопки мыши на связи активизируется контекстное меню, которое предоставляет быстрый доступ к установкам связи. Однако спецификации связи позволяют проделать то же самое при помощи вкладок диалоговых окон (рис. 2.16).

Вкладка Detail

Вкладка содержит информацию о дополнительных свойствах ассоциации.

Name direction	Указывает имя связанного класса
Constraints	Указывает выражение некоторого семантического условия, которое должно быть выполнено, в то время как система находится в устойчивом состоянии

Вкладки Source Role и Target Role

Показывает информацию об имени, стереотипе, родительском классе и другую основную информацию о связи.

Multiplicity	Показывает, сколько ожидается создать объектов данного класса, может быть задано числом или буквой «*n*». Значение «*n*» указывает, что количество не лимитировано. Можно задать различные варианты ожидаемого количества или диапазон. Причем указанное число отображается рядом со стрелкой связи
Access	Определяет область видимости связи
Aggregation	Показывает, что один класс не просто использует, а содержит другой. Для того чтобы показать, что один класс входит в другой, необходимо установить этот флажок во вкладке Source/Target Role, соответствующей классу-агрегату. При этом стрелка связи на диаграмме приобретает ромб с обратной стороны стрелки
Navigability	Определяет, является ли связь направленной или нет

Остальные настройки представляют интерес прежде всего для разработчиков ПО и определяют специфические для ООП понятия.

Dependency or instantiates (зависимость или реализация)

Этот тип связи позволяет показать, что один класс использует объекты другого (). Изменение в одной структуре может потребовать изменений в другой. Использование может осуществляться при передаче параметров или вызове операций класса. Во многих случаях другие типы зависимостей уже подразумевают какую-либо зависимость, но если вы хотите описать зависимости более детально — вы можете использовать dependency, чтобы описать связь между элементами. Это подразумевает, что зависимость слабая и не пригодна для использования в associative relation. Графически этот вид связи отражается пунктирной стрелкой.

Association Class (ассоциированный класс)

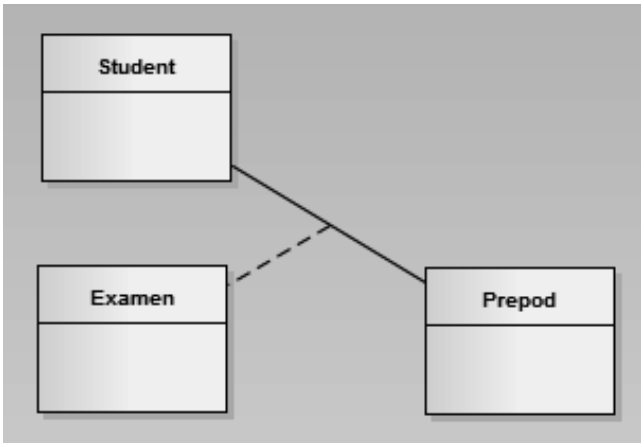


Рис. 2.17 – Пример ассоциированной связи

Используйте данный тип связи для отображения свойства ассоциации ( ). Свойства сохраняются в классе и соединяются связью *Association* (рис. 2.17). Следует отметить, что данного элемента управления может и не быть на стандартной панели инструментов. В таком случае используется меню по расширенному доступу к инструментам управления — кнопка More tools.

Пунктирная линия в примере показывает нам, что сущность Student относится к сущности Prepod и при этом существует сущность Examen, т. е. отношение между Student и Prepod зависит от существования Examen.

Generalization (обобщение)

Данный тип связи позволяет указать, что один класс является родительским по отношению к другому, при этом будет создана связь наследования класса ( ). Пример такой связи показан на рис. 2.18.

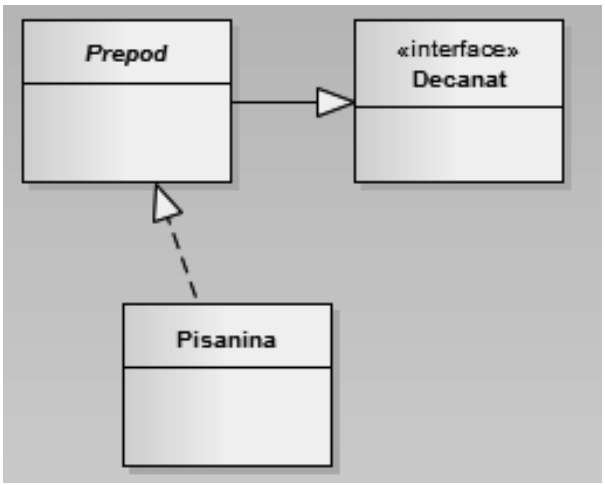


Рис. 2.18 – Пример связи обобщения

Абстрактный класс Prepod наследуется от интерфейса Decanat, Pisanina реализует функциональность Prepod [10].

Рекомендации по построению диаграмм классов (пример)

Создание диаграмм классов является во многом не формализуемым процессом, однако можно выделить несколько основных этапов:

Определение классов

При идентификации классов нашей системы воспользоваться следующей методологией, которая состоит из двух шагов:

- Перечисление кандидатов в классы, найденных в постановке проблемы.
- Исключение ненужных и некорректных классов.

При выявлении кандидатов в классы выделяются все понятия (прежде всего, существительные), имеющие отношение к системе. Из общего описания проекта и диаграммы вариантов использования (см. лаб. 1) выделяют список кандидатов

в классы. Из списка кандидатов в классы исключают «плохие классы» согласно следующим критериям:

- *Избыточные классы.* Если два класса отражают одну и ту же информацию, то сохраняется наиболее информативное имя. Например, хотя мы выделили отдельный класс *Администратор БД*, его функциональность полностью описывается родительским классом *Сотрудники*.
- *Нерелевантные классы*. Если класс не используется в решении проблемы, он должен быть исключен. Это определяется здравым смыслом, поскольку в другом контексте класс может быть важен. Например, в системе продажи театральных билетов должности покупателей безразличны, а должности персонала театра могут быть важны.

  В нашем проекте, например, класс *Родители студента* имеет лишь косвенное отношение к проектируемой системе.
- *Неясные классы*. Класс должен быть характерным, специальным. Некоторые пробные классы могут иметь плохо определенные границы или быть слишком широкими. Например, классы *Регистрация оплаты в БД*, *Регистрация процесса в БД* — являются неясными классами.
- *Атрибуты*. Имена, которые изначально описывают другие объекты, должны быть утверждены как атрибуты. Например, имя, возраст, вес и адрес — обычно атрибуты. В нашем случае, например *Логин* и *Пароль* — атрибуты класса *Учетная запись пользователя*.
- *Операции*. Если имя описывает операцию, которая применяется к объектам, а не манипулирует ими с собственными правами, то это не класс. Например, телефонный звонок является последовательностью действий абонента и телефонной сети.

  В нашем случае *Печать* отчета является операцией класса *Отчет*.
- *Конструкции реализации*. Конструкции, являющиеся подсистемами, функциональными блоками или устройствами, используемыми в системе, должны быть исключены из модели. Они могут быть нужны позднее, на последующих стадиях проектирования, но не сейчас. Например, *Компьютер, Принтер, Редактор текста* и *Подсистема аутентификации* — конструкции реализации для большинства приложений.

Оставшиеся классы размещаем на диаграмме. Отметим, что поскольку класс Сотрудники уже определён в модели, нет необходимости создавать его повторно — достаточно на диаграмму классов поместить ссылку на этот объект.

Определение связей между классами

Любая зависимость между классами или ссылка одного класса к другому является ассоциацией. Атрибуты не должны быть ссылками на класс при моделировании, вместо них используются ассоциации.

Ассоциации (таб. 2.3) часто соответствуют глаголам и глагольным фразам. Они включают физическое размещение (следующий, часть чего-либо, содержит), прямые действия (едет), коммуникацию (говорит кому-то), обладание (имеет) или удовлетворение некоторых условий (работает на, управляет).

Таблица 2.3 – Ассоциации

Классы	Тип связи	Множественность	Описание ассоциации
Сотрудники — Учетная запись пользователя	Unidirectional Association	1..n	Авторизируется
Сотрудники — Редактирование справочников	Unidirectional Association	1..n	Заполняет
Сотрудник — Отчет	Unidirectional Association	—	Формирует
Отчет — Настройка принтера	Aggregation	1	Использует
Справочник Студент — Редактирование справочников	Generalization	—	—
Справочник Учебный план — Редактирование справочников	Generalization	—	—

Определение атрибутов классов

Атрибуты являются свойствами объектов, таких, как *имя, вес, скорость* или *цвет*. При выделении атрибутов, следует учитывать следующие определения:

1). Атрибут должен быть характерен непосредственно для данного класса. Например, атрибут *Ориентация страницы* будет характерен для класса *Свойства печати*, а не для класса *Сотрудники*. Несмотря на то, что именно *Сотрудник* будет инициировать вызов отчета, а для сформированного отчета настраивать ориентацию страницы.

2). Атрибут должен отображать, одну и только одну характеристику класса. Если выделенный атрибут являлся сложным объектом либо содержит некоторую совокупность характеристик, выделите атрибут в отдельный класс либо разбейте его на несколько простых атрибутов. Например, атрибут *Реквизиты студента* является сложным составным атрибутом. Следует разделить данный атрибут на простые атрибуты: *ФИО, Адрес, Телефон* и т. д.

3). При описании классов, входящих в состав схемы наследования, для каждого атрибута необходимо определить, будет ли он общим для классов-наследников или специфичным для определенного класса. В первом случае атрибут включается в базовый класс, во втором — в соответствующий класс-наследник.

4). В базе данных каждая запись имеет свой уникальный код (ключ), который может использоваться при написании кода программы для идентификации

объекта. Однако при проектировании диаграммы классов идентификаторы объектов не следует выделять в атрибуты. Например, свойство *Код студента* не будет отображаться на диаграмме классов, однако наверняка будет использоваться в коде программы, участвуя в запросах.

5). При определении спецификаций атрибутов (*Public, Protected, Private*) пользуйтесь следующими правилами:

Public-атрибут доступен содержащему его классу и классам, имеющим связь с данным классом. *Private*-атрибут доступен только содержащему его классу. Если атрибут является закрытым (*Private*), однако необходимо воздействовать на него извне, будет присутствовать открытая (*Public*) функция, изменяющая или высчитывающая значение этого атрибута. *Protected*-атрибут определятся в базовом классе и доступен базовому классу и его наследникам.

Выделение операторов (методов) классов

1). Метод класса должен быть свойственным тому классу, где он определен. Например, метод *«Распечатать отчет»* будет являться оператором класса *Отчет*, а не *Сотрудник*, несмотря на то, что именно *Сотрудник* инициирует выполнение данного метода. Доступ *Сотрудника* к методу *«Распечатать отчет»* обеспечивается ассоциацией *Сотрудник–Отчет*.

2). Доступ к методу извне содержащего его класса может быть осуществлен только в том случае, если метод описан с открытыми (*Public*) правами доступа. Закрытые (*Private*) методы также обычно инициируются открытыми методами. Так, например, закрытый метод *«Делегировать права»* класса *Проверить пользователя* инициируется открытым методом *«Соединиться»* того же класса.

3). Операторы не должны быть слишком сложными, состоящими из множества различных функций. В этом случае необходимо разбить оператор на более простые составляющие. Например, оператор *«Редактировать запись»* разбивается на составляющие его простые функции: *«Добавить», «Удалить», «Сохранить», «Открыть»* и т. д.

4). Исключение из правила принадлежности метода классу составляют методы по созданию и удалению объектов класса. Подобные методы не могут быть включены в класс, так как класс не может вызвать метод создания собственного объекта. Создание объекта класса может произойти лишь извне, то есть из другого класса. Так, например, метод по созданию объекта класса *Товар*, может быть включен в класс *Прайс-лист*, но никак не в класс *Товар*.

5). Соблюдайте принцип полноты методов класса. Если, например, в классе присутствует метод *«Добавить запись»*, то должен быть определен метод *«Удалить запись»*.

Результатом работы будет построенная диаграмма классов, внешний вид которой приведен на рис. 2.19.

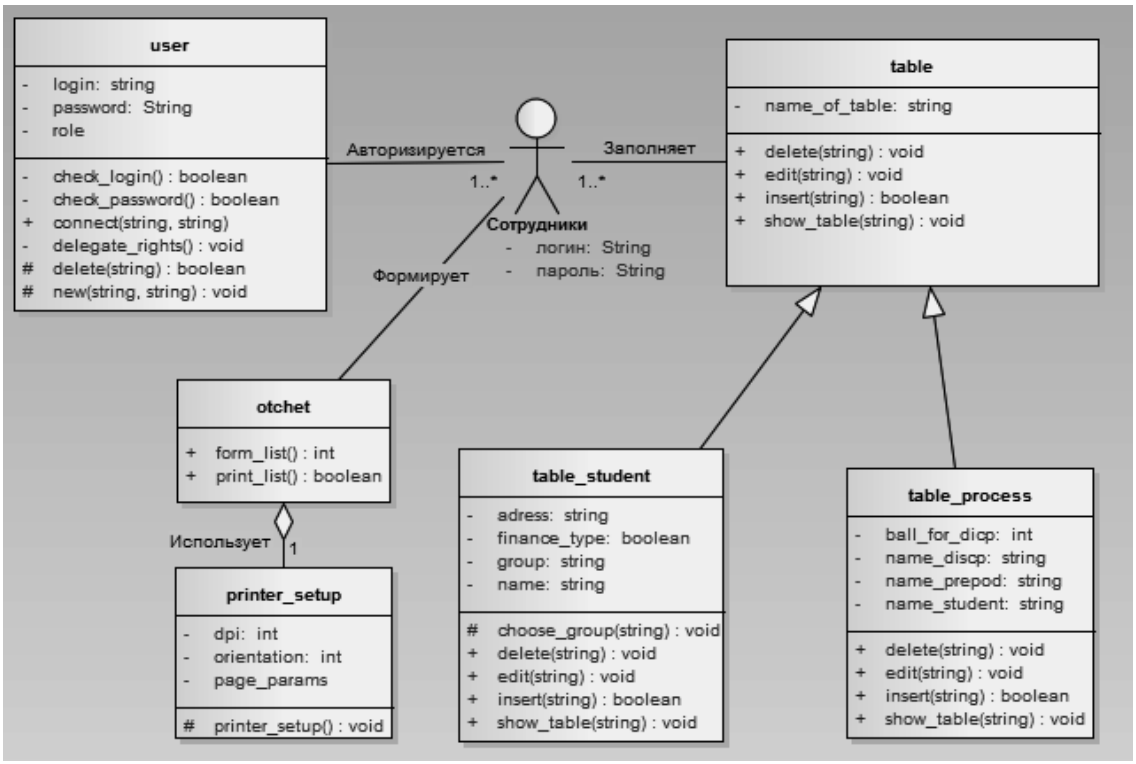


Рис. 2.19 – Готовая диаграмма классов для проекта «Учёба в вузе»

Задание

Необходимо создать диаграмму, следуя описанным принципам работы с системой. Объектом автоматизации является тот же процесс, что и в предыдущей работе. Результат — дополнительная диаграмма в уже существующем файле проекта.

Контрольные вопросы по главе 2

1) Дайте понятие диаграммы классов и класса.
2) В чем отличие конкретного класса от абстрактного?
3) Какая область видимости соответствует следующему определению? «Атрибут с этой областью видимости недоступен или не виден для всех классов, за исключением подклассов данного класса»?
4) В чем особенность интерфейса по отношению к обычному классу?
5) Может ли быть композитным отношение, удовлетворяющее следующему условию: «Данное отношение представляет собой сущность, которая включает в себя в качестве составных частей другие сущности»?

Глава 3

ДИАГРАММЫ КООПЕРАЦИИ И ПОСЛЕДОВАТЕЛЬНОСТИ

Диаграмма кооперации предназначена для описания поведения системы на уровне отдельных объектов, которые обмениваются между собой сообщениями, чтобы достичь нужной цели или реализовать некоторый вариант использования. С точки зрения аналитика или архитектора системы в проекте важно представить структурные связи отдельных объектов между собой. Такое представление структуры модели как совокупности взаимодействующих объектов и обеспечивает диаграмма кооперации.

Кооперация (collaboration/communication) *— спецификация множества объектов отдельных классов, совместно взаимодействующих с целью реализации отдельных вариантов использования в общем контексте моделируемой системы.*

Понятие кооперации — одно из фундаментальных в языке UML. Цель самой кооперации состоит в том, чтобы специфицировать особенности реализации отдельных вариантов использования или наиболее значимых операций в системе. Кооперация определяет структуру поведения системы в терминах взаимодействия участников этой кооперации.

На диаграмме кооперации размещаются объекты, представляющие собой экземпляры классов, связи между ними, которые, в свою очередь, являются экземплярами ассоциаций и сообщения. Связи дополняются стрелками сообщений, при этом показываются только те объекты, которые участвуют в реализации моделируемой кооперации. Далее, как и на диаграмме классов, показываются структурные отношения между объектами в виде различных соединительных линий. Связи могут дополняться именами ролей, которые играют объекты в данной взаимосвязи. И, наконец, изображаются динамические взаимосвязи — потоки сообщений в фор-

ме стрелок с указанием направления рядом с соединительными линиями между объектами, при этом задаются имена сообщений и их порядковые номера в общей последовательности сообщений.

Одна и та же совокупность объектов может участвовать в реализации различных кооперации. В зависимости от рассматриваемой кооперации могут изменяться как связи между отдельными объектами, так и поток сообщений между ними. Именно это отличает диаграмму кооперации от диаграммы классов, на которой должны быть указаны все без исключения классы, их атрибуты и операции, а также все ассоциации и другие структурные отношения между элементами модели.

3.1 Объекты и их графическое изображение

Объект (object) *— сущность с хорошо определенными границами и индивидуальностью, которая инкапсулирует состояние и поведение.*

В контексте языка UML любой объект является экземпляром класса, описанного в модели и представленного на диаграмме классов. Объект создается на этапе реализации модели или выполнения программы. Он имеет собственное имя и конкретные значения атрибутов. Следует рассмотреть особенности семантики и графической нотации объектов, из которых строятся диаграммы.

Для диаграмм кооперации полное имя объекта в целом представляет собой строку текста, разделенную двоеточием и записанную в формате:

<Собственное имя объекта >/<Имя роли класса>:<Имя класса >.

Имя роли класса указывается в том случае, когда соответствующий класс отсутствует в модели или разработчику необходимо акцентировать внимание на особенности его использования в рассматриваемом контексте моделирования взаимодействия. Имя класса — это имя одного из классов, представленного на диаграмме классов. Важно отметить, что вся запись имени объекта подчеркивается, что является визуальным признаком объектов на различных диаграммах языка UML.

Если указано собственное имя объекта, то оно должно начинаться со строчной буквы. В то же время имя объекта, имя роли с символом «/» или имя класса могут отсутствовать. Однако двоеточие всегда должно стоять перед именем класса, а косая черта — перед именем роли.

Таким образом, на диаграммах кооперации могут встретиться следующие варианты возможных записей полного имени объекта:

- $o : C$ — объект с собственным именем o, экземпляр класса C;
- $: C$ — анонимный объект, экземпляр класса C;
- o: (или просто o) — объект-сирота с собственным именем o;
- $o / R : C$ — объект с собственным именем o, экземпляр класса C, играющий роль R;
- $/ R : C$ — анонимный объект, экземпляр класса C, играющий роль R;

- *o* / *R* — объект-сирота с собственным именем *o*, играющий роль *R*;
- / *R* — анонимный объект и одновременно объект-сирота, играющий роль *R*.

Если собственное имя объекта отсутствует, то такой объект принято называть *анонимным*. Однако в этом случае обязательно ставится двоеточие перед именем соответствующего класса. Отсутствовать может и имя класса — такой объект называется *сиротой*. Для него записывается только собственное имя объекта, двоеточие не ставится, имя класса не указывается. Если для объектов указываются атрибуты, то в большинстве случаев они принимают конкретные значения. Для отдельных объектов могут быть дополнительно указаны роли, которые они играют в кооперации.

В контексте языка UML все объекты делятся на две категории: пассивные и активные. Пассивный объект оперирует только данными и не может инициировать деятельность по управлению другими объектами. Однако пассивные объекты могут посылать сигналы в процессе выполнения запросов, которые они обрабатывают. На диаграмме кооперации пассивные объекты изображаются обычным образом без дополнительных стереотипов.

Активный объект (active object) имеет собственный процесс управления и может инициировать деятельность по управлению другими объектами.

3.2 Связи на диаграмме кооперации

Связь (link) *— любое семантическое отношение между некоторой совокупностью объектов.*

Связь как элемент языка UML является экземпляром или примером произвольной ассоциации и может иметь место между двумя и более объектами. Бинарная связь на диаграмме кооперации изображается отрезком сплошной линии, соединяющей два прямоугольника объектов. На концах этой линии дополнительно могут быть явно указаны имена ролей соответствующей ассоциации (рис. 3.1).

Связи не имеют собственных имен, поскольку идентичны как экземпляры некоторой ассоциации. Другими словами, все связи на диаграмме кооперации могут быть только анонимными и при необходимости записываются без двоеточия перед именем ассоциации. Однако чаще всего имена связей на диаграммах кооперации не указываются. Для связей не указывается также и кратность концевых точек.

Как было отмечено выше, особенности моделирования взаимодействия в контексте языка UML заключаются в том, чтобы специфицировать коммуникацию между множеством взаимодействующих объектов. Каждое взаимодействие описывается совокупностью сообщений, которыми участвующие в нем объекты обмениваются между собой.

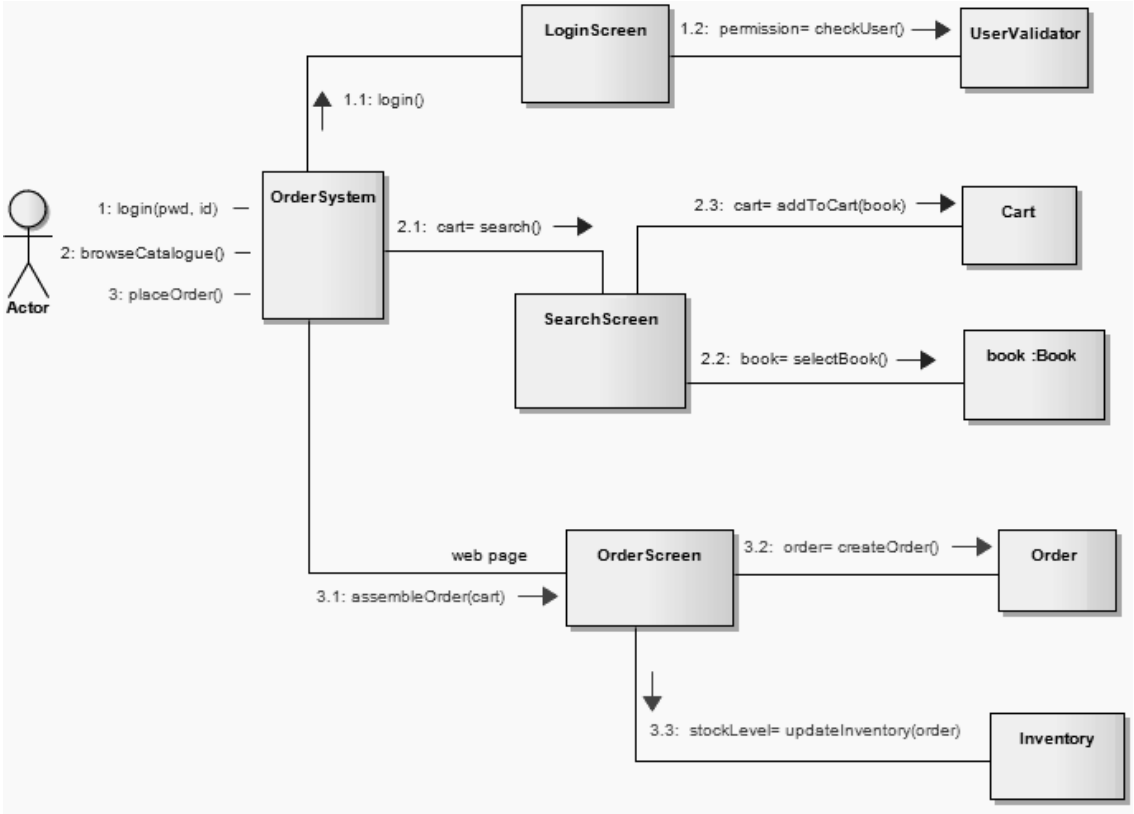


Рис. 3.1 – Пример диаграммы коммуникации «Покупка книги в интернет-магазине»

3.3 Сообщения и их графическое изображение

Сообщение (message) *— спецификация передачи информации от одного элемента модели к другому с ожиданием выполнения определенных действий со стороны принимающего элемента.*

При этом первый объект предполагает, что после получения сообщения вторым объектом последует выполнение некоторого действия. На диаграмме кооперации сообщение является причиной или стимулом начала выполнения операций, отправки сигналов, создания и уничтожения отдельных объектов. Связь обеспечивает канал для направленной передачи сообщений между объектами от объекта-источника к объекту-получателю.

В этом смысле сообщение представляет собой законченный фрагмент информации, который отправляется одним объектом другому. При этом прием сообщения может инициировать выполнение определенных действий, направленных на решение отдельной задачи тем объектом, которому это сообщение отправлено. Сообщения не только передают информацию, но и требуют или предполагают от

принимающего объекта выполнения ожидаемых действий. Сообщения могут инициировать выполнение операций объектом соответствующего класса, а параметры этих операций передаются вместе с сообщением.

В таком контексте каждое сообщение имеет направление от объекта, который инициирует и отправляет сообщение, к объекту, который его получает. Иногда отправителя сообщения называют клиентом, а получателя — сервером. При этом сообщение от клиента имеет форму запроса некоторого сервиса, а реакция сервера на запрос после получения сообщения может быть связана с выполнением определенных действий или передачи клиенту необходимой информации тоже в форме сообщения.

Сообщения в языке UML также специфицируют роли, которые играют объекты — отправитель и получатель сообщения. Сообщения на диаграмме кооперации изображаются дополнительными стрелками рядом с соответствующей связью или ролью ассоциации. Направление стрелки указывает на получателя сообщения. Внешний вид стрелки сообщения имеет определенный смысл. На диаграммах кооперации может использоваться один из трех типов стрелок для обозначения сообщений (рис. 3.2).

Рис. 3.2 – Графическое изображение различных типов сообщений на диаграмме кооперации

- Сплошная линия с треугольной стрелкой (рис. 3.2, *а*) обозначает вызов процедуры (операции) или передачу потока управления. Сообщения этого типа могут быть использованы параллельно активными объектами, когда один из них передает сообщение этого типа и ожидает, пока не закончится некоторая последовательность действий, выполняемая вторым объектом. Обычно все такие сообщения синхронны, т. е. инициируются по завершении деятельности или при выполнении определенного условия.
- Сплошная линия с *V*-образной стрелкой (рис. 3.2, *б*) обозначает асинхронное сообщение в простом потоке управления. В этом случае клиент передает асинхронное сообщение и продолжает выполнять свою деятельность, не ожидая ответа от сервера.
- Пунктирная линия с *V*-образной стрелкой (рис. 3.2, *в*) обозначает возврат из вызова процедуры. Стрелки этого типа зачастую отсутствуют на диаграммах кооперации, поскольку неявно предполагается их существование после окончания процесса выполнения операции или деятельности.

В языке UML определены следующие стереотипы сообщений:

- ≪call≫ (вызвать) — сообщение, требующее вызова операции или процедуры объекта-получателя. Если сообщение с этим стереотипом рефлексивное, то оно инициирует локальный вызов операции у пославшего это сообщение объекта;

- ≪return≫ (возвратить) — сообщение, возвращающее значение выполненной операции или процедуры вызвавшему ее объекту. Значение результата может инициировать ветвление потока управления;
- ≪create≫ (создать) — сообщение, требующее создания другого объекта для выполнения определенных действий. Созданный объект может стать активным (ему передается поток управления), а может остаться пассивным;
- ≪destroy≫ (уничтожить) — сообщение с явным требованием уничтожить соответствующий объект. Посылается в том случае, когда необходимо прекратить нежелательные действия со стороны существующего в системе объекта, либо когда объект больше не нужен и должен освободить задействованные им системные ресурсы;
- ≪send≫ (послать) — обозначает посылку другому объекту сигнала, который асинхронно инициируется одним объектом и принимается (перехватывается) другим. Отличие сигнала от сообщения заключается в том, что сигнал должен быть явно описан в том классе, объект которого инициирует его передачу.

3.4 Рекомендации по построению диаграмм кооперации

Построение диаграммы кооперации можно начинать сразу после построения диаграммы классов. При разработке диаграмм кооперации вначале изображаются объекты и связи между ними. Далее на диаграмму кооперации необходимо нанести все сообщения, указав их порядок и другие семантические особенности. Диаграмма кооперации может содержать только те объекты и связи, которые уже определены на построенной ранее диаграмме классов. В противном случае, если возникает необходимость включения в диаграмму кооперации объектов, которые создаются на основе отсутствующих классов, то диаграмма классов должна быть модифицирована посредством включения в нее явного описания этих классов.

Процесс построения диаграммы кооперации должен быть согласован с процессами построения диаграммы классов и диаграммы последовательности. В первом случае, как уже отмечалось, необходимо следить за использованием только тех объектов, для которых определены порождающие их классы. Во втором случае необходимо согласовывать последовательности передаваемых сообщений. Речь идет о том, что не допускается различный порядок следования сообщений для моделирования одного и того же взаимодействия на диаграмме кооперации и диаграмме последовательности.

Необходимо помнить, что диаграмма кооперации, с одной стороны, обеспечивает концептуально согласованный переход от статической модели диаграммы классов к динамическим моделям поведения, представляемым диаграммами последовательности, состояний и деятельности. С другой стороны, диаграмма этого типа предопределяет особенности реализации модели на диаграммах компонентов и развертывания.

3.5 Объекты и их изображение на диаграмме последовательности

Диаграмма последовательности (sequence diagram) — *диаграмма, на которой показаны взаимодействия объектов, упорядоченные по времени их проявления.*

Особенности взаимодействия элементов моделируемой системы могут быть представлены на диаграммах кооперации и последовательности. Диаграммы кооперации используются для спецификации динамики поведения систем, хотя время в явном виде в них отсутствует. Однако временной аспект поведения может иметь существенное значение при моделировании синхронных процессов, описывающих взаимодействие объектов. Именно для этой цели в языке UML используются диаграммы последовательности.

На диаграмме последовательности неявно присутствует ось времени, что позволяет визуализировать временные отношения между передаваемыми сообщениями. С помощью диаграммы последовательности можно представить взаимодействие элементов модели как своеобразный временной график «жизни» всей совокупности объектов, связанных между собой для реализации варианта использования программной системы, достижения бизнес-цели или выполнения какой-либо задачи.

На диаграмме последовательности также изображаются объекты, которые непосредственно участвуют во взаимодействии, при этом никакие статические связи с другими объектами не визуализируются. Для диаграммы последовательности ключевым моментом является именно динамика взаимодействия объектов во времени. При этом диаграмма последовательности имеет как бы два измерения. Одно — слева направо в виде вертикальных линий, каждая из которых изображает линию жизни отдельного объекта, участвующего во взаимодействии. Второе измерение диаграммы последовательности — вертикальная временная ось, направленная сверху вниз.

Каждый объект графически изображается в форме прямоугольника и располагается в верхней части своей линии жизни (рис. 3.3). Внутри прямоугольника записываются собственное имя объекта со строчной буквы и имя класса, разделенные двоеточием. Для объектов диаграммы последовательности остаются справедливыми правила именования, рассмотренные ранее применительно к диаграммам кооперации. Если на диаграмме последовательности отсутствует собственное имя объекта, то при этом должно быть указано имя класса. Такой объект считается *анонимным*. Может отсутствовать и имя класса, но при этом должно быть указано собственное имя объекта. Такой объект считается *сиротой*. Роль классов в именах объектов на диаграммах последовательности, как правило, не указывается.

Крайним слева на диаграмме изображается объект — инициатор моделируемого процесса взаимодействия (объект «Студент» на рис. 3.3). Правее — другой объект, который непосредственно взаимодействует с первым. Таким образом, порядок

расположения объектов на диаграмме последовательности определяется исключительно соображениями удобства визуализации их взаимодействия друг с другом.

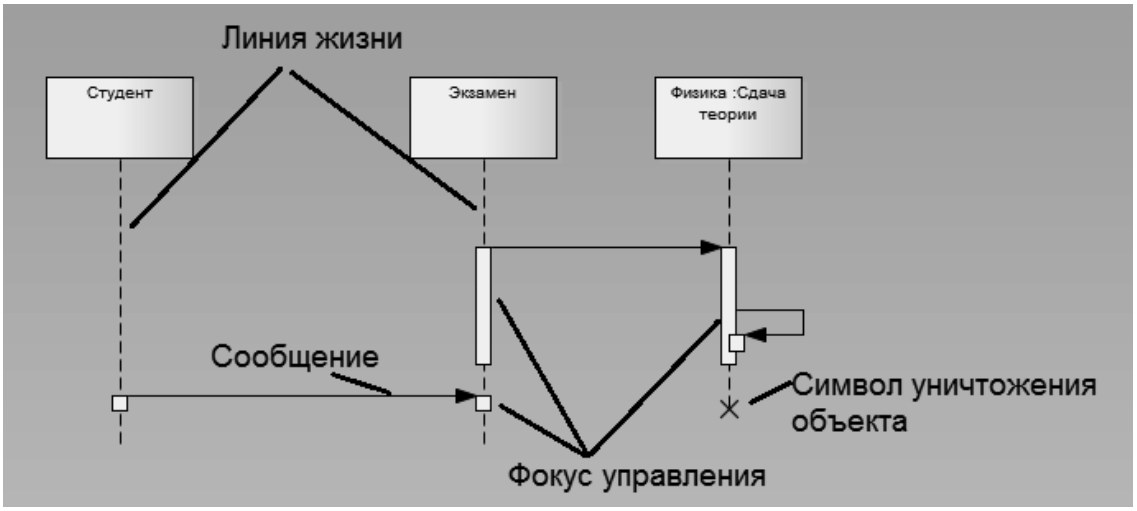


Рис. 3.3 – Графические элементы диаграммы последовательности

Начальному моменту времени соответствует самая верхняя часть диаграммы. При этом процесс взаимодействия объектов реализуется посредством сообщений, которые посылаются одними объектами другим. Сообщения изображаются в виде горизонтальных стрелок с именем сообщения и образуют определенный порядок относительно времени своей инициализации. Другими словами, сообщения, расположенные на диаграмме последовательности выше, передаются раньше тех, которые расположены ниже. При этом масштаб на оси времени не указывается, поскольку диаграмма последовательности моделирует лишь временную упорядоченность взаимодействий типа «раньше-позже».

Линия жизни объекта (object lifeline) *— вертикальная линия на диаграмме последовательности, которая представляет существование объекта в течение определенного периода времени.*

Линия жизни объекта изображается пунктирной вертикальной линией, ассоциированной с единственным объектом на диаграмме последовательности. Линия жизни служит для обозначения периода времени, в течение которого объект существует в системе и, следовательно, может потенциально участвовать во всех ее взаимодействиях. Если объект существует в системе постоянно, то и его линия жизни должна продолжаться по всей рабочей области диаграммы последовательности от самой верхней ее части до самой нижней (объекты «Студент» и «Экзамен» рисунка 3.3).

Отдельные объекты, закончив выполнение своих операций, могут быть уничтожены, чтобы освободить занимаемые ими ресурсы. Для таких объектов линия жизни обрывается в момент его уничтожения. Для обозначения момента уничтожения объекта в языке UML применяется специальный символ в форме латинской

буквы «*X*». На рис. 3.4 этот символ используется для уничтожения анонимного объекта «Class1», образованного от Object1. Ниже этого символа пунктирная линия не изображается, поскольку соответствующего объекта в системе уже нет, и этот объект должен быть исключен из всех последующих взаимодействий.

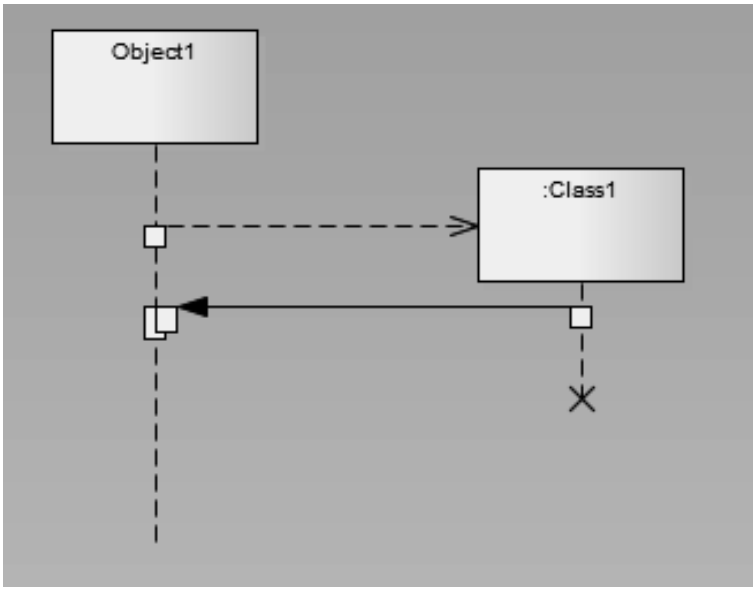


Рис. 3.4 – Графическое изображение линий жизни и фокусов управления объектов

Вовсе не обязательно создавать все объекты в начальный момент времени. Отдельные объекты в системе могут создаваться по мере необходимости, существенно экономя ресурсы системы и повышая ее производительность. В этом случае прямоугольник такого объекта изображается не в верхней части диаграммы последовательности, а в той, которая соответствует моменту создания объекта. При этом прямоугольник объекта вертикально располагается в том месте диаграммы, которое по оси времени совпадает с моментом его возникновения в системе. Объект создается со своей линией жизни, а возможно, и с фокусом управления.

В процессе функционирования объектно-ориентированных систем одни объекты могут находиться в активном состоянии, непосредственно выполняя определенные действия, или в состоянии пассивного ожидания сообщений от других объектов. Фокус управления — символ, применяемый для того, чтобы явно выделить подобную активность объектов на диаграммах последовательности.

Фокус управления (focus of control) *— специальный символ на диаграмме последовательности, указывающий период времени, в течение которого объект выполняет некоторое действие, находясь в активном состоянии.*

Фокус управления изображается в форме вытянутого узкого прямоугольника, верхняя сторона которого обозначает начало получения фокуса управления объекта (начало активности), а ее нижняя сторона — окончание фокуса управления

(окончание активности). Этот прямоугольник располагается ниже обозначения соответствующего объекта и может заменять его линию, если на всем ее протяжении он активен.

Периоды активности объекта могут чередоваться с периодами его пассивности или ожидания. В этом случае у такого объекта фокусы управления изменяют свое изображение на линию жизни, и наоборот (объект «Экзамен» на рис. 3.3). Важно понимать, что получить фокус управления может только объект, у которого в этот момент имеется линия жизни. Если же объект был уничтожен, то вновь возникнуть в системе он уже не может. Вместо него может быть создан лишь экземпляр этого же класса, который, строго говоря, будет другим объектом.

В отдельных случаях инициатором взаимодействия в системе может быть актер или внешний пользователь. При этом актер изображается на диаграмме последовательности самым первым объектом слева со своим фокусом управления (рис. 3.5). Наиболее часто актер и его фокус управления будут существовать в системе постоянно, отмечая характерную для пользователя активность в инициировании взаимодействий с системой. Актер может иметь собственное имя либо оставаться анонимным.

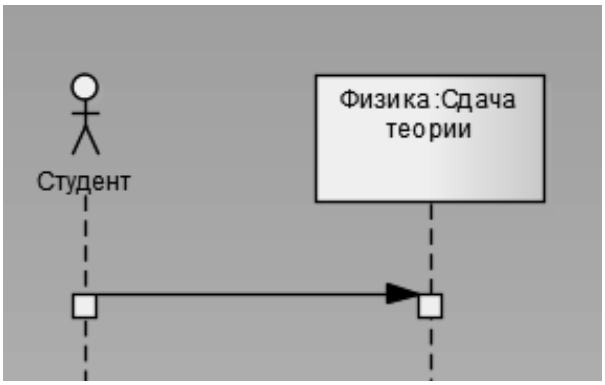


Рис. 3.5 – Пример изображения актера на диаграмме последовательности

В отдельных случаях объект может посылать сообщения самому себе, инициируя так называемые *рефлексивные* сообщения. Если в результате рефлексивного сообщения создается новый подпроцесс или нить управления, то говорят о рекурсивном или вложенном фокусе управления. На диаграмме последовательности рекурсия обозначается небольшим прямоугольником, присоединенным к правой стороне фокуса управления того объекта, для которого изображается данное рекурсивное взаимодействие (сообщение у объекта *«Физика:Сдача теории»* на рис. 3.3).

3.6 Сообщения на диаграмме последовательности

Сообщения как элементы языка UML уже рассматривались ранее при изучении диаграммы кооперации. Стрелки сообщений изображаются аналогично рассмотренным ранее, но применительно к диаграммам последовательности сообщения имеют дополнительные семантические особенности. При этом на диаграмме последовательности все сообщения упорядочены по времени своей передачи в моделируемой системе, хотя номера у них могут не указываться.

На диаграммах последовательности могут присутствовать три разновидности сообщений, каждое из которых имеет свое графическое изображение (рис. 3.2).

Первая разновидность сообщения (рис. 3.2, *а*) наиболее распространена и используется для вызова процедур, выполнения операций или обозначения отдельных вложенных потоков управления. Начало этой стрелки, как правило, соприкасается с фокусом управления того объекта-клиента, который инициирует это сообщение. Конец стрелки соприкасается с линией жизни того объекта, который принимает это сообщение и выполняет в ответ определенные действия. При этом принимающий объект может получить фокус управления, становясь в этом случае активным. Передающий объект может потерять фокус управления или остаться активным.

Вторая разновидность сообщения (рис. 3.2, *б*) используется для обозначения простого асинхронного сообщения, которое передается в произвольный момент времени. Передача такого сообщения обычно не сопровождается получением фокуса управления объектом-получателем.

Третья разновидность сообщения (рис. 3.2, *в*) используется для возврата из вызова процедуры. Примером может служить простое сообщение о завершении вычислений без предоставления результата расчетов объекту-клиенту. В процедурных потоках управления эта стрелка может быть опущена, поскольку ее наличие неявно предполагается в конце активизации объекта. В то же время считается, что каждый вызов процедуры имеет свою пару — возврат вызова. Для непроцедурных потоков управления, включая параллельные и асинхронные сообщения, стрелка возврата должна указываться явным образом.

Обычно сообщения изображаются горизонтальными стрелками, соединяющими линии жизни или фокусы управления двух объектов на диаграмме последовательности. При этом неявно предполагается, что время передачи сообщения достаточно мало по сравнению с процессами выполнения действий объектами. Считается также, что за время передачи сообщения с соответствующими объектами не может произойти никаких событий. Другими словами, состояния объектов не изменяются. Если же это предположение не может быть признано справедливым, то стрелка сообщения изображается под наклоном, так, чтобы конец стрелки располагался ниже ее начала.

Каждое сообщение на диаграмме последовательности ассоциируется с определенной операцией, которая должна быть выполнена принявшим его объектом. При этом операция может иметь аргументы или параметры, значения которых влияют на получение различных результатов. Соответствующие параметры операции будет иметь и вызывающее это действие сообщение. Более того, значения параметров отдельных сообщений могут содержать условные выражения, образуя ветвление или альтернативные пути основного потока управления.

3.7 Ветвление потока управления

Одна из особенностей диаграммы последовательности — возможность визуализировать простое ветвление процесса. Для изображения ветвления используются две или более стрелки, выходящие из одной точки фокуса управления объекта. При этом рядом с каждой из них должно быть явно указано соответствующее условие ветви в форме булевского выражения (рис. 3.6).

Количество ветвей может быть произвольным, однако наличие ветвлений может существенно усложнить интерпретацию диаграммы последовательности. Предложение-условие должно быть явно указано для каждой ветви и записывается в форме обычного текста, псевдокода или выражения языка программирования. Это выражение всегда должно возвращать некоторое булевское выражение. Запись этих условий должна исключать одновременную передачу альтернативных сообщений по двум и более ветвям. В противном случае на диаграмме последовательности может возникнуть конфликт ветвления [10].

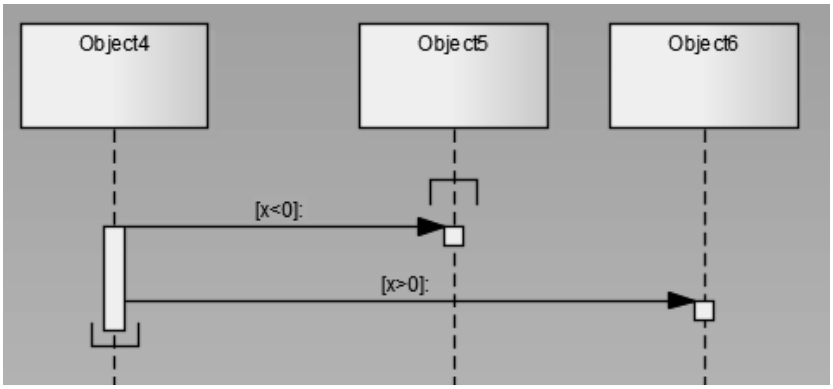


Рис. 3.6 – Графическое изображение бинарного ветвления потока управления на диаграмме последовательности

Лабораторная работа №3

Цель работы

Целью данной работы является построение диаграммы последовательности.

Краткое изложение теоретической части

На диаграмме последовательности (*Sequence Diagram*) изображаются только те объекты, которые непосредственно участвуют во взаимодействии. Ключевым моментом для диаграмм последовательности является динамика взаимодействия объектов во времени.

В UML диаграмма последовательности имеет как бы два измерения. Первое слева направо в виде вертикальных линий, каждая из которых изображает линию жизни отдельного объекта, участвующего во взаимодействии. Крайним слева на диаграмме изображается объект, который является инициатором взаимодействия. Правее изображается другой объект, который непосредственно взаимодействует с первым. Таким образом, все объекты на диаграмме последовательности образуют некоторый порядок, определяемый очередностью или степенью активности объектов при взаимодействии друг с другом. Графически каждый объект изображается прямоугольником и располагается в верхней части своей линии жизни. Внутри прямоугольника записываются имя объекта и имя класса, разделенные двоеточием.

Вторым измерением диаграммы последовательности является вертикальная временная ось, направленная сверху вниз. Начальному моменту времени соответствует самая верхняя часть диаграммы. Взаимодействия объектов реализуются посредством сообщений, которые посылаются одними объектами другим в рамках одного прецедента (use case). Сообщения изображаются в виде горизонтальных стрелок с именем сообщения, а их порядок определяется временем возникновения. То есть сообщения, расположенные на диаграмме последовательности выше, инициируются раньше тех, которые расположены ниже. Масштаб на оси времени не указывается, поскольку диаграмма последовательности моделирует лишь временную упорядоченность взаимодействий типа «раньше-позже».

Диаграммы последовательности обязательно опираются на диаграмму классов. Объекты должны принадлежать классам, описанным в диаграмме классов. Если создаваемый объект не может принадлежать ни одному из существующих классов, возможно, следует доработать диаграмму классов. Поэтому в диаграммах последовательности нет необходимости создавать новые объекты, их достаточно просто перетащить в рабочую область текущей диаграммы из соответствующей диаграммы классов и определить имя экземпляра данного класса (рис. 3.7–3.8).

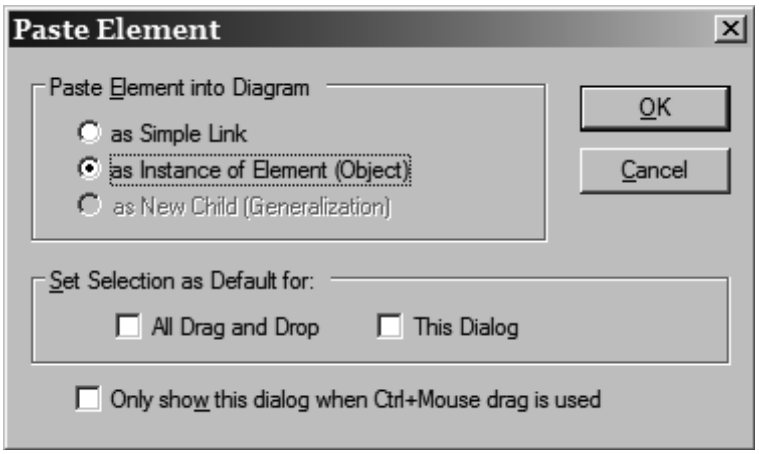


Рис. 3.7 – Создание экземпляра класса в Enterprise Architect

Рис. 3.8 – Экземпляр класса Сотрудники с заданной ролью «Деканат»

Линия жизни объекта

Линия жизни объекта (object lifeline) изображается пунктирной вертикальной линией (рис. 3.8), ассоциированной с единственным объектом на диаграмме последовательности. Линия жизни служит для обозначения периода времени, в течение которого объект существует в системе и, следовательно, может потенциально участвовать во всех ее взаимодействиях. Если объект существует в системе постоянно, то и его линия жизни должна продолжаться по всей плоскости диаграммы последовательности от самой верхней ее части до самой нижней.

Отдельные объекты, выполнив свою роль в системе, могут быть уничтожены, чтобы освободить занимаемые ими ресурсы. Для таких объектов линия жизни обрывается в момент его уничтожения. Для обозначения момента уничтожения объекта в языке UML используется специальный символ в форме латинской буквы «*X*». Ниже этого символа пунктирная линия не изображается, поскольку соответствующего объекта в системе уже нет, и этот объект должен быть исключен из всех последующих взаимодействий.

Не обязательно создавать все объекты диаграммы в начальный момент времени. Отдельные объекты могут создаваться по мере необходимости, экономя ресурсы системы и повышая ее производительность. В этом случае прямоугольник объекта изображается не в верхней части диаграммы последовательности, а в той части, которая соответствует моменту создания объекта. При этом прямоугольник объекта вертикально располагается в том месте диаграммы, которое по оси времени совпадает с моментом его возникновения в системе. Объект обязательно создается со своей линией жизни и, возможно, с фокусом управления.

Фокус управления

В процессе функционирования объектно-ориентированных систем одни объекты могут находиться в активном состоянии, непосредственно выполняя определенные действия или состоянии пассивного ожидания сообщений от других объектов. Чтобы явно выделить подобную активность объектов, в языке UML применяется специальное понятие, получившее название фокуса управления (focus of control). Фокус управления изображается в форме вытянутого узкого прямоугольника, верхняя сторона которого обозначает начало получения фокуса управления объекта (начало активности), а его нижняя сторона — окончание фокуса управления (окончание активности). Прямоугольник располагается ниже обозначения соответствующего объекта и может заменять его линию жизни, если на всем ее протяжении он является активным.

Периоды активности объекта могут чередоваться с периодами его пассивности или ожидания. В этом случае у такого объекта имеются несколько фокусов управления. Важно сознавать, что получить фокус управления может только существующий объект, у которого в этот момент имеется линия жизни. Если же некоторый объект был уничтожен, то вновь возникнуть в системе он уже не может. Вместо него лишь может быть создан другой экземпляр этого же класса, который, строго говоря, будет являться другим объектом.

В отдельных случаях инициатором взаимодействия в системе может быть актер или внешний пользователь. В этом случае актер изображается на диаграмме после-

довательности самым первым объектом слева со своим фокусом управления. Чаще всего актер и его фокус управления будут существовать в системе постоянно, отмечая характерную для пользователя активность в инициировании взаимодействий с системой. При этом актер может иметь собственное имя или оставаться анонимным.

Иногда некоторый объект может инициировать рекурсивное взаимодействие с самим собой. Наличие во многих языках программирования специальных средств построения рекурсивных процедур требует визуализации соответствующих понятий в форме графических примитивов. На диаграмме последовательности рекурсия обозначается небольшим прямоугольником, присоединенным к правой стороне фокуса управления того объекта, для которого изображается это рекурсивное взаимодействие.

Сообщения

В UML каждое взаимодействие описывается совокупностью сообщений, которыми участвующие в нем объекты обмениваются между собой. Сообщение (message) представляет собой законченный фрагмент информации, который отправляется одним объектом другому. Прием сообщения инициирует выполнение определенных действий, направленных на решение отдельной задачи тем объектом, которому это сообщение отправлено.

Таким образом, сообщения не только передают некоторую информацию, но и требуют или предполагают выполнения ожидаемых действий от принимающего объекта. Сообщения могут инициировать выполнение операций объектом соответствующего класса, а параметры этих операций передаются вместе с сообщением. На диаграмме последовательности все сообщения упорядочены по времени своего возникновения в моделируемой системе. В таком контексте каждое сообщение имеет направление от объекта, который инициирует и отправляет сообщение, к объекту, который его получает. Иногда отправителя сообщения называют клиентом, а получателя сервером. Тогда сообщение от клиента имеет форму запроса некоторого сервиса, а реакция сервера на запрос после получения сообщения может быть связана с выполнением определенных действий или передачи клиенту необходимой информации тоже в форме сообщения.

Сообщения изображаются горизонтальными стрелками, соединяющими линии жизни или фокусы управления двух объектов на диаграмме последовательности. В языке UML различаются несколько разновидностей сообщений, каждое из которых имеет свое графическое изображение:

- первая разновидность сообщения является наиболее распространенной и используется для вызова процедур, выполнения операций или обозначения отдельных вложенных потоков управления. Начало этой стрелки всегда соприкасается с фокусом управления или линией жизни того объекта-клиента, который инициирует это сообщение. Конец стрелки соприкасается с линией жизни того объекта, который принимает это сообщение и выполняет в ответ определенные действия. Принимающий объект, как правило, получает фокус управления, становясь активным;
- вторая разновидность сообщения используется для обозначения простого потока управления. Каждая такая стрелка указывает на выполнение одно-

го шага потока. Такие сообщения, обычно, являются асинхронными, то есть могут возникать в произвольные моменты времени. Передача такого сообщения, как правило, сопровождается получением фокуса управления принявшим его объектом;

- третья разновидность явно обозначает асинхронное сообщение между двумя объектами в некоторой процедурной последовательности. Примером такого сообщения может служить прерывание операции при возникновении исключительной ситуации. В этом случае информация о такой ситуации передается вызывающему объекту для продолжения процесса дальнейшего взаимодействия;
- четвертая разновидность сообщения используется для возврата из вызова процедуры. Примером может служить простое сообщение о завершении некоторых вычислений без предоставления результата расчетов объекту-клиенту. В процедурных потоках управления эта стрелка может быть опущена, поскольку ее наличие неявно предполагается в конце активизации объекта. Считается, что каждый вызов процедуры имеет свою пару — возврат вызова. Для непроцедурных потоков управления, включая параллельные и асинхронные сообщения, стрелка возврата должна указываться явным образом.

Предполагается, что время передачи сообщения достаточно мало по сравнению с процессами выполнения действий объектами, то есть за время передачи сообщения с соответствующими объектами не может произойти никаких изменений. Если же это предположение не может быть признано справедливым, то стрелка сообщения изображается под некоторым наклоном, так, чтобы конец стрелки располагался ниже ее начала.

В отдельных случаях объект может посылать сообщения самому себе, инициируя так называемые рефлексивные сообщения. Такие сообщения изображаются прямоугольником со стрелкой, начало и конец которой совпадают. Подобные ситуации возникают, например, при обработке нажатий на клавиши клавиатуры при вводе текста в редактируемый документ, при наборе цифр номера телефона абонента.

Таким образом, в языке UML каждое сообщение ассоциируется с некоторым действием, которое должно быть выполнено принявшим его объектом. Действие может иметь некоторые аргументы или параметры, в зависимости от конкретных значений которых может быть получен различный результат. Соответствующие параметры будет иметь и вызывающее это действие сообщение. Более того, значения параметров отдельных сообщений могут содержать условные выражения, образуя ветвление или альтернативные пути основного потока управления.

Ветвление потока управления

Для изображения ветвления потока управления рисуются две или более стрелки, выходящие из одной точки фокуса управления объекта. При этом соответствующие условия должны быть явно указаны рядом с каждой из стрелок в форме сторожевого условия. Сторожевые условия должны взаимно исключать одновременную передачу альтернативных сообщений.

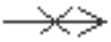
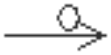
Стереотипы сообщений

В языке UML предусмотрены некоторые стандартные действия, выполняемые в ответ на получение соответствующего сообщения. Эти действия могут быть явно указаны на диаграмме последовательности в форме стереотипа рядом с сообщением, к которому относятся. В этом случае они записываются в кавычках. Используются следующие стереотипы сообщений:

- *«call» (вызвать)* — сообщение, требующее вызова операции или процедуры принимающего объекта. Если сообщение с этим стереотипом рефлексивное, то оно инициирует локальный вызов операции у самого пославшего это сообщение объекта;
- *«return» (возвратить)* — сообщение, возвращающее значение выполненной операции или процедуры вызвавшему ее объекту. Значение результата может инициировать ветвление потока управления;
- *«create» (создать)* — сообщение, требующее создания другого объекта для выполнения определенных действий. Созданный объект может получить фокус управления, а может и не получить его;
- *«destroy» (уничтожить)* — сообщение с явным требованием уничтожить соответствующий объект. Посылается в том случае, когда необходимо прекратить нежелательные действия со стороны существующего в системе объекта либо когда объект больше не нужен и должен освободить задействованные им системные ресурсы;
- *«send» (послать)* — обозначает посылку другому объекту некоторого сигнала, который асинхронно инициируется одним объектом и принимается другим. Отличие сигнала от сообщения заключается в том, что сигнал должен быть явно описан в том классе, объект которого инициирует его передачу.

Кроме стереотипов, сообщения могут иметь собственное обозначение операции, вызов которой они инициируют у принимающего объекта (таб. 3.1). В этом случае рядом со стрелкой записывается имя операции с круглыми скобками, в которых могут указываться параметры или аргументы соответствующей операции. Если параметры отсутствуют, то скобки все равно должны присутствовать после имени операции. Примерами таких операций могут служить следующие: *«Выдать клиенту наличными сумму (n)», «Установить соединение между абонентами (a, b)», «Сделать вводимый текст невидимым ()», «Подать звуковой сигнал тревоги ()».*

Таблица 3.1 – Виды сообщений

Название свойства	Графическое изображение стрелки	Назначение свойства
Simple (Простое)		Данное сообщение выполняется в одном потоке управления. Это свойство задается добавляемому на диаграмму сообщению по умолчанию
Synchronous (Синхронное)		После передачи данного сообщения клиент ожидает ответа от объекта-приемника о результате выполнения соответствующей операции
Balking (С отказом)		После передачи данного сообщения объект-приемник отказывает клиенту в выполнении соответствующей операции, если он занят выполнением других операций
Timeout (С ожиданием)		После передачи данного сообщения объект-приемник может поместить данное сообщение в очередь с ограниченным временем ожидания, если он занят выполнением других операций
Procedure Call (Вызов процедуры)		Клиент посылает данное сообщение объекту-приемнику и, чтобы продолжить свою работу, ожидает, пока вся дальнейшая вложенная последовательность сообщений не будет обработана приемником
Asynchronous (Асинхронное)		Клиент посылает данное сообщение и продолжает свою работу, не ожидая подтверждения от объекта-приемника о получении этого сообщения. При этом соответствующая операция может быть как выполнена, так и не выполнена
Return (Возврат)		Данное сообщение посылается клиенту после окончания выполнения вызова процедуры

Временные ограничения на диаграммах последовательности

В отдельных случаях выполнение тех или иных действий на диаграмме последовательности может потребовать явной спецификации временных ограничений, накладываемых на сам интервал выполнения операций или передачу сообщений. В языке UML для записи временных ограничений используются фигурные скобки. Временные ограничения могут относиться как к выполнению определенных действий объектами, так и к самим сообщениям, явно специфицируя условия их передачи или приема. В отличие от условий ветвления, которые должны выполняться альтернативно, временные ограничения имеют обязательный или директивный характер для ассоциированных с ними объектов.

Временные ограничения могут записываться рядом с началом стрелки соответствующего сообщения. Но наиболее часто они записываются слева от стрелки на одном уровне с ней. Если временная характеристика относится к конкретному объекту, то имя этого объекта записывается перед именем характеристики и отделяется от нее точкой.

Примерами таких ограничений на диаграмме последовательности могут служить ситуации, когда необходимо явно специфицировать время, в течение которого допускается передача сообщения от клиента к серверу или обработка запроса клиента сервером:

- {время_приема_сообщения время_отправки_сообщения < 1 сек.};
- {время_ожидания_ответа < 5 сек.};
- {время_передачи_пакета < 10 сек.};
- {объект_1. время_подачи_сигнала_тревоги > 30 сек.}.

Рекомендации по построению диаграмм последовательности (пример)

Прежде всего, при создании диаграмм взаимодействия необходимо определить сценарии, которые будут отображены на диаграммах. Для каждого сценария составляется описание, отражающее ситуацию, которую будет описывать диаграмма. При составлении описания сценария следует придерживаться следующих рекомендаций:

- Сценарий должен отображать вполне конкретную, реальную ситуацию работы с системой. Следует по возможности исключать абстрактные понятия и альтернативные варианты развития событий. Например, сценарий *«Администратор входит в систему»* будет изначально ошибочным, так как оперирует абстрактным понятием Пользователь и неконкретным развитием событий *«Входит в систему»* (вход может быть успешным или не успешным). Вместо абстрактного понятия *Администратор* следует взять конкретного пользователя системы (например, *Администратор Иванов И. И.*) и рассмотреть конкретную последовательность действий (например, *«Администратор Иванов И. И. вводит логин и пароль и успешно входит в систему»*).
- Так как диаграмма последовательности отображает конкретную последовательность событий, то для отображения других вариантов развития со-

бытий следует построить несколько диаграмм. Например, в дополнение сценарию *«Администратор Иванов И. И. вводит логин и пароль и успешно входит в систему»* можно рассмотреть последовательность *«Администратор Петрова П. П. вводит верный логин и неверный пароль и система отказывает в доступе»*.

- При выделении сценариев пользуйтесь диаграммой вариантов использования. Диаграммы взаимодействия обычно соответствуют одному или нескольким вариантам использования. Например, сценарий *«Администратор Иванов И. И. вводит логин и пароль и успешно входит в систему»* привязан к варианту использования *«Вход в систему»*. Диаграмма взаимодействия *«Формирование отчета по студентам на дату 10/10/2010 Администратором Ивановым И. И.»* будет охватывать несколько вариантов использования: *«Подтверждение прав на доступ», «Просмотр справочника «Студент» и выбор позиций», «Формирование отчета», «Печать отчета»*.
- Старайтесь избегать избыточности диаграмм взаимодействия. Например, если в диаграмме *«Администратор Иванов И. И. вводит логин и пароль и успешно входит в систему»* вы описали последовательности входа в систему администратора, то в диаграмме *«Администратор Иванов И. И. вносит новую запись в справочник «Студент»»* последовательность входа можно уже не описывать. Старайтесь, чтобы диаграммы взаимодействия отражали наиболее сложные и неочевидные участки работы системы.

Рассмотрим простейший сценарий *«Администратор Иванов И. И. вводит логин и пароль и успешно входит в систему»*.

Первым этапом создания диаграммы последовательности является выделение объектов-участников сценария. В данной диаграмме выделим два объекта:

- *Иванов И. И.: Администратор БД*
- *Учетная запись администратора: Учетная запись*

Следующим этапом создания диаграммы последовательности является определение событий, которыми обмениваются выделенные объекты. При создании событий помните, что подавляющее большинство событий будет являться либо методами классов-получателей, доступными классу-отправителю, либо воздействием на атрибуты класса, либо результатом выполнения какого-либо метода. Если при создании события вы считаете, что оно должно являться методом соответствующего класса и обнаруживаете, что этот метод отсутствует, следует добавить его для соответствующего класса в диаграмме классов.

На рис. 3.9 отображена диаграмма последовательности для сценария *«Администратор Иванов И. И. вводит логин и пароль и успешно входит в систему»*.

Вводит логин и пароль — это изменение значений атрибутов *логин* и *пароль* класса *Администратор БД* (эти атрибуты наследуются им из класса *Сотрудники*) на *ivan69* и *s2w#er!* соответственно.

connecting(ivan69, s2w#er!) — это вызов открытого (Public) метода класса *user*, доступного классу *Администратор БД* благодаря связи *Авторизируется* от класса *Сотрудники* к классу *user* (*Администратор БД* наследует не только атрибуты и методы класса Сотрудники, но и ассоциации) (см. рис. 3.6).

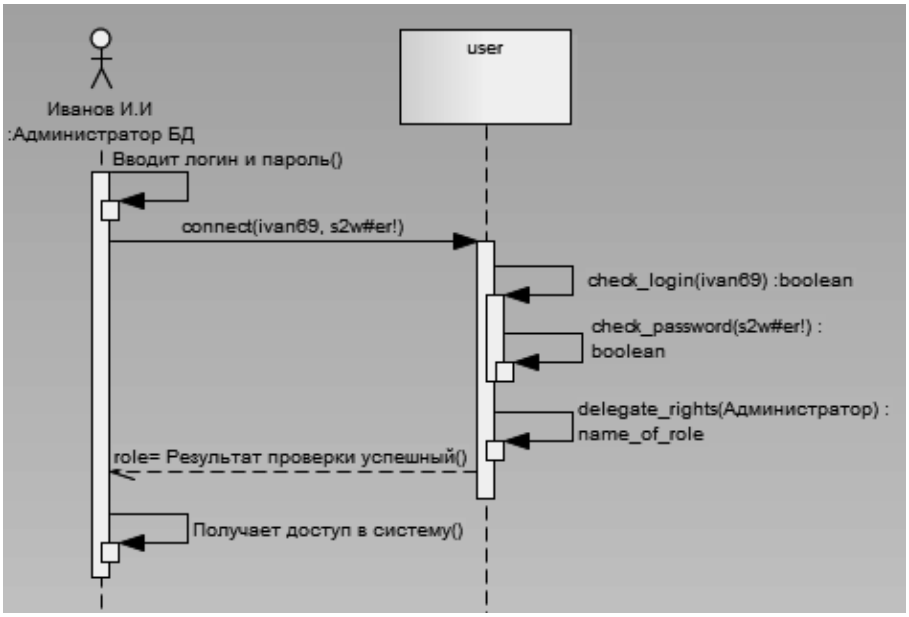


Рис. 3.9 – Диаграмма последовательности «Процедура верификации»

Далее идёт проверка логина и пароля с помощью приватных (Private) методов класса user, причём от результата выполнения функции *check_login()* зависит начало выполнения функции *check_password()*.

В случае успешного завершения проверки (а мы условились рассматривать ситуацию, когда логин и пароль введены верно), вызывается ещё один приватный метод класса *user*, изменяющий значение текущей роли пользователя на *«Администратор»* с помощью функции *delegate_rights()*.

Результат проверки успешный — Результат выполнения метода *delegate_rights()*.

Получает доступ в систему — действие, отражающее тот факт, что Иванов И. И. зашел в систему с правами администратора.

На рис. 3.10 отображена еще одна диаграмма последовательности, отражающая обслуживание, — печать списка студентов по состоянию на 10.10.2010. Для данной диаграммы составлено следующее описание:

Администратор Иванов И. И. проходит успешную проверку своих прав и формирует отчет за 10/10/2010 для группы 516-1 по справочнику «Студент», проверяет настройки принтера HP3015, после чего распечатывает отчет.

Обратите внимание, что так же как и при создании диаграммы вариантов использования и диаграммы классов, в данном случае рассматривается только то, что входит в рамки проектируемой системы. События, происходящие вне нее (например, *Иванов И. И. визирует отчет своей подписью* или *Электрик дядя Вася меняет пробки для того, чтобы восстановить подачу электроэнергии в корпус, обесточенный 5 минут назад*) не рассматриваются.

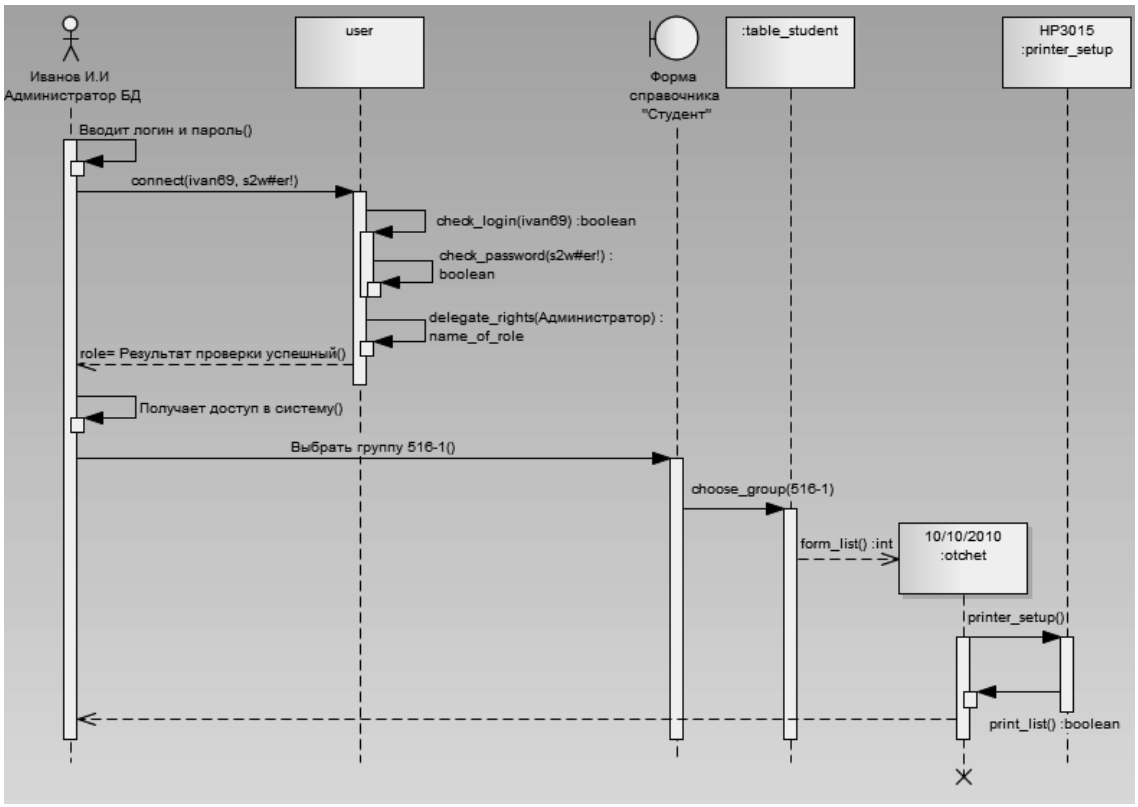


Рис. 3.10 – Диаграмма последовательности «Формирование отчета»

Задание

В вашей модели создайте не менее 3-х диаграмм последовательности.

Контрольные вопросы по главе 3

1) В чем отличие актера от обычного класса?
2) В чем отличие анонимного объекта от объекта-сироты?
3) Какому виду сообщения соответствует графическое изображение в виде сплошной линии с V-образной стрелкой?
4) Зачем на диаграмме последовательности нужна линия жизни?
5) Перечислите основные виды сообщений.

Глава 4

ДИАГРАММЫ СОСТОЯНИЙ

Для систем различной природы и назначения характерно взаимодействие между собой отдельных образующих их элементов. Для представления динамических особенностей взаимодействия элементов модели, в контексте реализации вариантов использования, предназначены диаграммы кооперации и последовательности. Однако для моделирования процессов функционирования большинства сложных систем, особенно систем реального времени, этих представлений недостаточно.

Диаграмма состояний (statechart diagram) *— диаграмма, которая представляет конечный автомат.*

Семантика понятия состояния довольно сложна. Дело в том, что характеристика состояний системы явным образом не зависит от логической структуры, зафиксированной на диаграмме классов. При рассмотрении состояний системы приходится отвлекаться от особенностей ее объектной структуры и мыслить категориями, описывающими динамический контекст поведения моделируемой системы.

Ранее отмечалось, что любая прикладная система характеризуется не только структурой составляющих ее элементов, но и некоторым поведением или функциональностью. Для общего представления функциональности моделируемой системы предназначены диаграммы вариантов использования, которые на концептуальном уровне описывают поведение системы в целом. Для того чтобы представить наиболее общее поведение на логическом уровне, следует ответить на вопрос: «В процессе какого поведения система реализует необходимую пользователям функциональность?».

Главное назначение диаграммы состояний — описать возможные последовательности состояний и переходов, которые в совокупности характеризуют поведение моделируемой системы в течение всего ее жизненного цикла. Диаграмма состояний представляет динамическое поведение сущностей, на основе специфи-

кации их реакции на восприятие некоторых конкретных событий. Системы, которые реагируют на внешние действия от других систем или от пользователей, иногда называют реактивными. Если такие действия инициируются в произвольные случайные моменты времени, то говорят об асинхронном поведении модели.

Диаграммы состояний чаще всего используются для описания поведения отдельных систем и подсистем. Они также могут быть применены для спецификации функциональности экземпляров отдельных классов, т. е. для моделирования всех возможных изменений состояний конкретных объектов. Диаграмма состояний по существу является графом специального вида, который служит для представления конечного автомата.

Диаграммы состояний могут быть вложены друг в друга, образуя вложенные диаграммы для более детального представления состояний отдельных элементов модели. Для понимания семантики конкретной диаграммы состояний необходимо представлять особенности поведения моделируемой сущности, а также иметь общие сведения из теории конечных автоматов.

Конечный автомат (state machine) *— модель для спецификации поведения объекта в форме последовательности его состояний, которые описывают реакцию объекта на внешние события, выполнение объектом действий, а также изменение его отдельных свойств.*

В контексте языка UML понятие конечного автомата обладает дополнительной семантикой. Вершинами графа конечного автомата являются состояния и другие типы элементов модели, которые изображаются соответствующими графическими символами. Дуги графа служат для обозначения переходов из состояния в состояние. Конечный автомат описывает поведение отдельного объекта в форме последовательности состояний, охватывающих все этапы его жизненного цикла, начиная от создания объекта и заканчивая его уничтожением. Каждая диаграмма состояний представляет собой конечный автомат.

Основными понятиями, характеризующими конечный автомат, являются состояние и переход. Ключевое различие между ними заключается в том, что длительность нахождения системы в отдельном состоянии существенно превышает время, которое затрачивается на переход из одного состояния в другое. Предполагается, что в пределе время перехода из одного состояния в другое равно нулю (если дополнительно ничего не сказано). Другими словами, переход объекта из состояния в состояние происходит мгновенно.

В общем случае конечный автомат представляет динамические аспекты моделируемой системы в виде ориентированного графа, вершины которого соответствуют состояниям, а дуги — переходам. При этом поведение моделируется как последовательное перемещение по графу состояний от вершины к вершине по связывающим их дугам с учетом их ориентации. Для графа состояний системы можно ввести в рассмотрение специальные свойства.

Среди таких свойств — выделение из всей совокупности состояний двух специальных: начального и конечного. Ни в графе состояний, ни на диаграмме состояний

время нахождения системы в том или ином состоянии явно не учитывается, однако предполагается, что последовательность изменения состояний упорядочена во времени. Другими словами, каждое последующее состояние может наступить позже предшествующего ему состояния.

4.1 Состояние и его графическое изображение

Моделирование поведения объектов и системы в целом основывается на понятии состояния.

Состояние (state) *— условие или ситуация в ходе жизненного цикла объекта, в течение которого он удовлетворяет логическому условию, выполняет определенную деятельность или ожидает события.*

Состояние может быть задано в виде набора конкретных значений атрибутов объекта некоторого класса, при этом изменение отдельных значений этих атрибутов будет отражать изменение состояния моделируемого объекта или системы в целом. Однако не каждый атрибут класса может характеризовать состояние его объектов. Как правило, имеют значение только те свойства элементов системы, которые отражают динамический или функциональный аспект ее поведения. В этом случае состояние будет характеризоваться некоторым инвариантным условием, включающим в себя только принципиальные для поведения объекта или системы атрибуты классов и их значения.

Такое условие может соответствовать ситуации, когда моделируемый объект находится в состоянии ожидания возникновения внешнего события. В то же время нахождение объекта в некотором состоянии может быть связано с выполнением определенных действий. В последнем случае соответствующая деятельность начинается в момент перехода моделируемого элемента в рассматриваемое состояние, а после и элемент может покинуть данное состояние в момент завершения этой деятельности.

Состояние на диаграмме изображается прямоугольником со скругленными вершинами (рис. 4.1). Этот прямоугольник, в свою очередь, может быть разделен на две секции горизонтальной линией. Если указана лишь одна секция, то в ней записывается только имя состояния (рис. 4.1, *а*). В противном случае в первой из них записывается имя состояния, а во второй — список некоторых внутренних действий или переходов в данном состоянии (рис. 4.1, *б*). При этом под действием в языке UML понимают некоторую атомарную операцию, выполнение которой приводит к изменению состояния или возврату некоторого значения (например, «истина» или «ложь»).

Имя состояния представляет собой строку текста, которая раскрывает содержательный смысл или семантику данного состояния. Имя должно представлять собой законченное предложение и всегда записываться с заглавной буквы. Поскольку состояние системы является частью процесса ее функционирования, рекомендуется

в качестве имени использовать глаголы в настоящем времени или соответствующие причастия. Как исключение, имя у состояния может отсутствовать, т. е. оно необязательно для некоторых состояний. В этом случае состояние является анонимным. Если на одной диаграмме состояний несколько анонимных состояний, то все они должны различаться между собой.

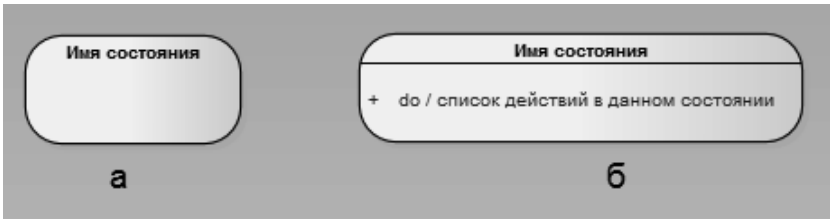


Рис. 4.1 – Графическое изображение состояний на диаграмме состояний

Действие (action) *— спецификация выполнимого утверждения, которая образует абстракцию вычислительной процедуры.*

Действие обычно приводит к изменению состояния системы и может быть реализовано посредством передачи сообщения объекту, модификацией связи или значения атрибута. Для ряда состояний может потребоваться дополнительно указать действия, которые должны быть выполнены моделируемым элементом. Для этой цели служит добавочная секция в обозначении состояния, содержащая перечень внутренних действий или деятельность, которые производятся в процессе нахождения моделируемого элемента в данном состоянии. Каждое действие записывается в виде отдельной строки и имеет следующий формат:

<метка действия/выражение действия>.

Метка действия указывает на обстоятельства или условия, при которых будет выполняться деятельность, определенная выражением действия. При этом выражение действия может использовать любые атрибуты и связи, принадлежащие области имен или контексту моделируемого объекта. Если список выражений действия пустой, то метка действия с разделителем в виде наклонной черты «/» не указывается. Перечень меток действий в языке UML фиксирован, причем эти метки не могут быть использованы в качестве имен событий:

- Входное действие (entry action) — действие, которое выполняется в момент перехода в данное состояние. Обозначается с помощью ключевого слова — метки действия entry, которое указывает на то, что следующее за ней выражение действия должно быть выполнено в момент входа в данное состояние.
- Действие выхода (exit action) — действие, производимое при выходе из данного состояния. Обозначается с помощью ключевого слова — метки действия exit, которое указывает на то, что следующее за ней выражение действия должно быть выполнено в момент выхода из данного состояния.
- Внутренняя деятельность (do activity) — выполнение объектом операций или процедур, которые требуют определенного времени. Обозначается с по-

мощью ключевого слова — метки деятельности do, которое специфицирует так называемую «ду-деятельность», выполняемую в течение всего времени, пока объект находится в данном состоянии, или до тех пор, пока не будет прервано внешним событием. При нормальном завершении внутренней деятельности генерируется соответствующее событие.

Во всех остальных случаях метка действия идентифицирует событие, которое запускает соответствующее выражение действия. Эти события называются внутренними переходами. Семантически они эквивалентны переходам в само это состояние, за исключением той особенности, что выход из этого состояния или повторный вход в него не происходит. Это означает, что действия входа и выхода не производятся. При этом выполнение внутренних действий в состоянии не может быть прервано никакими внешними событиями, в отличие от внутренней деятельности, выполнение которой требует определенного времени.

В качестве примера состояния можно рассмотреть аутентификацию клиента для доступа к ресурсам моделируемой информационной системы (рис. 4.2). Список внутренних действий в данном состоянии может включать следующие действия. Первое действие — входное, которое выполняется при входе в это состояние и связано с получением строки символов, соответствующих паролю клиента. Далее выполняется деятельность по проверке введенного клиентом пароля. При успешном завершении этой проверки выполняется действие на выходе, которое отображает меню доступных для клиента опций.

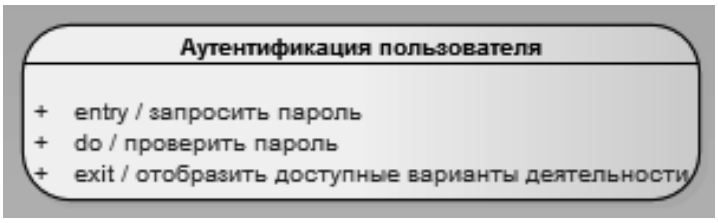


Рис. 4.2 – Пример состояния «Аутентификация пользователя»

Кроме обычных состояний, на диаграмме состояний могут размещаться псевдосостояния.

Псевдосостояние (pseudo-state) *— вершина в конечном автомате, которая имеет форму состояния, но не обладает поведением.*

Примерами псевдосостояний, которые определены в языке UML, являются начальное и конечное состояния.

Начальное состояние (start state) *— разновидность псевдосостояния, обозначающее начало выполнения процесса изменения состояний конечного автомата или нахождения моделируемого объекта в составном состоянии.*

В этом состоянии находится объект по умолчанию в начальный момент времени. Оно служит для указания на диаграмме состояний графической области, от которой начинается процесс изменения состояний. Графически начальное состояние в языке UML обозначается в виде закрашенного кружка (рис. 4.3, *а*), из которого может только выходить стрелка-переход.

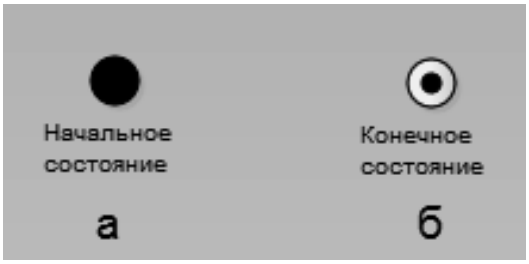


Рис. 4.3 – Графическое изображение начального и конечного состояний на диаграмме состояний

На самом верхнем уровне представления объекта переход из начального состояния может быть помечен событием создания (инициализации) данного объекта. В противном случае этот переход никак не помечается. Если этот переход не помечен, то он является первым переходом на диаграмме состояний в следующее за ним состояние. Каждая диаграмма или поддиаграмма состояний должна иметь единственное начальное состояние.

Конечное состояние (final state) *— разновидность псевдосостояния, обозначающее прекращение процесса изменения состояний конечного автомата или нахождения моделируемого объекта в составном состоянии.*

В этом состоянии должен находиться моделируемый объект или система по умолчанию после завершения работы конечного автомата. Оно служит для указания на диаграмме состояний графической области, в которой завершается процесс изменения состояний или жизненный цикл данного объекта. Графически конечное состояние в языке UML обозначается в виде закрашенного кружка, помещенного в окружность (рис. 4.3, *б*), в которую может только входить стрелка-переход. Каждая диаграмма состояний или подсостояний может иметь несколько конечных состояний, при этом все они считаются эквивалентными на одном уровне вложенности состояний.

4.2 Переход и событие

Пребывание моделируемого объекта или системы в первом состоянии может сопровождаться выполнением некоторых внутренних действий или деятельности. При этом изменение текущего состояния объекта будет возможно либо после завершения этих действий (деятельности), либо при возникновении некоторых внешних

событий. В обоих случаях говорят, что происходит переход объекта из одного состояния в другое.

Переход (transition) *— отношение между двумя состояниями, которое указывает на то, что объект в первом состоянии должен выполнить определенные действия и перейти во второе состояние.*

Переход осуществляется при наступлении некоторого события: окончания выполнения деятельности (do activity), получения объектом сообщения или приема сигнала. На переходе указывается имя события, а также действия, производимые объектом в ответ на внешние события при переходе из одного состояния в другое.

Переход может быть направлен в то же состояние, из которого он выходит. В этом случае его называют переходом в себя. Исходное и целевое состояния перехода в себя совпадают. Этот переход изображается петлей со стрелкой и отличается от внутреннего перехода. При переходе в себя объект покидает исходное состояние, а затем снова входит в него. При этом всякий раз выполняются внутренние действия, специфицированные метками entry и exit.

Срабатывание <перехода> (fire) *— выполнение перехода из одного состояния в другое.*

Срабатывание перехода может зависеть не только от наступления события, но и от выполнения определенного условия, называемого сторожевым. Объект перейдет из одного состояния в другое в том случае, если произошло указанное событие и сторожевое условие приняло значение «истина». До срабатывания перехода объект находится в предыдущем от него состоянии, называемом исходным, или в источнике (не путать с начальным состоянием — это разные понятия), а после его срабатывания объект находится в последующем от него состоянии (целевом состоянии).

На диаграмме состояний переход изображается сплошной линией со стрелкой, которая выходит из исходного состояния и направлена в целевое состояние. Каждый переход может быть помечен строкой текста, которая имеет следующий общий формат:

<имя события>'('<список параметров, разделенных запятыми>')' '['<сторожевое условие>']' <выражение действия>.

Событие (event) *— спецификация существенных явлений в поведении системы, которые имеют местоположение во времени и пространстве.*

Формально, событие представляет собой спецификацию факта, имеющего место в пространстве и во времени. Про события говорят, что они «происходят», при

этом отдельные события должны быть упорядочены во времени. После наступления события нельзя уже вернуться к предыдущим, если такая возможность явно не предусмотрена в модели.

Семантика понятия события фиксирует внимание на внешних проявлениях качественных изменений, происходящих при переходе моделируемого объекта из состояния в состояние. Например, при включении электрического переключателя происходит событие, в результате которого комната освещается. После успешного ремонта компьютера также происходит немаловажное событие — восстановление его работоспособности. Если поднять трубку обычного телефона, то, в случае его исправности, мы ожидаем услышать тоновый сигнал. Это тоже является событием.

В языке UML события играют роль стимулов, которые инициируют переходы из одних состояний в другие. В качестве событий можно рассматривать сигналы, вызовы, окончание фиксированных промежутков времени или моменты окончания выполнения определенных действий. В зависимости от вида происходящих событий — стимулов в языке UML различают два типа переходов: триггерные и нетриггерные.

*Переход называется **триггерным**, если его специфицирует событие-триггер, связанное с внешними условиями по отношению к рассматриваемому состоянию.*

В этом случае рядом со стрелкой триггерного перехода обязательно указывается имя события в форме строки текста, начинающейся со строчной буквы. Наиболее часто в качестве имен триггерных переходов задают имена операций, вызываемых у тех или иных объектов системы. После имени такого события следуют круглые скобки для явного задания параметров соответствующей операции. Если таких параметров нет, то список параметров со скобками может отсутствовать. Например, переход на рис. 4.4, *а* является триггерным, поскольку с ним связано конкретное событие-триггер, происходящее асинхронно при срабатывании некоторого датчика.

*Переход называется **нетриггерным**, если он происходит по завершении выполнения ду-деятельности в данном состоянии.*

Нетриггерные переходы часто называют переходами по завершении ду-деятельности. Для них рядом со стрелкой перехода не указывается никакого имени события, а в исходном состоянии должна быть описана внутренняя ду-деятельность, по окончании которой произойдет тот или иной нетриггерный переход.

***Сторожевое условие (guard condition)** — логическое условие, записанное в прямых скобках и представляющее собой булевское выражение.*

Рис. 4.4 – Графическое изображение триггерного (*а*) и нетриггерного (*б*) переходов на диаграмме состояний

При этом булевское выражение должно принимать одно из двух взаимно исключающих значений: «истина» или «ложь». Из контекста диаграммы состояний должна явно следовать семантика этого выражения, а для записи выражения может использоваться обычный язык, псевдокод или язык программирования.

Дополнение триггерных и нетриггерных переходов сторожевыми условиями позволяет явно специфицировать семантику их срабатывания. Если сторожевое условие принимает значение «истина», то соответствующий переход при наступлении события-триггера или завершении деятельности может сработать, в результате чего объект перейдет в целевое состояние. Если же сторожевое условие принимает значение «ложь», то переход не может сработать, даже если произошло событие-триггер или завершилась деятельность в исходном состоянии. Очевидно, в случае невыполнения сторожевого условия моделируемый объект или система останется в исходном состоянии. Однако вычисление истинности сторожевого условия в модели происходит только после возникновения ассоциированного с ним события-триггера или завершения деятельности, которые инициируют соответствующий переход.

Поскольку общее количество выходящих переходов из любого состояния в языке UML не ограничено, хотя и является конечным, не исключена ситуация, когда из одного состояния могут выходить несколько переходов с идентичным событием-триггером. Каждый такой переход должен содержать собственное сторожевое условие, при этом никакие два или более сторожевых условий не должны одновременно принимать значение «истина». В противном случае на диаграмме состояний возникнет конфликт триггерных переходов, что делает несостоятельной (ill formed) модель системы в целом.

Аналогичное замечание справедливо и для нетриггерных переходов, когда из одного состояния выходят несколько переходов по завершении деятельности. Каждый из таких переходов также должен содержать собственное сторожевое условие, при этом никакие два или более сторожевых условий не должны одновременно принимать значение «истина». В противном случае на диаграмме состояний будет

иметь место конфликт нетриггерных переходов, что также делает несостоятельной (ill formed) модель системы в целом.

Выражение действия (action expression) *представляет собой вызов операции или передачу сообщения, имеет атомарный характер и выполняется сразу после срабатывания соответствующего перехода до начала действий в целевом состоянии.*

Выражение действия выполняется в том и только в том случае, когда переход срабатывает. Атомарность действия означает, что оно не может быть прервано никаким другим действием до тех пор, пока не закончится его выполнение. Данное действие может оказывать влияние как на сам объект, так и на его окружение, если это с очевидностью следует из контекста модели. Данное выражение записывается после знака «/» в строке текста, присоединенной к соответствующему переходу.

В общем случае, выражение действия может содержать целый список отдельных действий, разделенных символом *«;»*. Обязательное требование — все действия из списка должны четко различаться между собой и следовать в порядке их записи. На синтаксис записи выражений действия не накладывается никаких ограничений. Главное — их запись должна быть понятна разработчикам модели и программистам. Поэтому чаще всего выражения записывают на одном из языков программирования, который предполагается использовать для реализации модели.

В качестве примера выражения действия перехода (рис. 4.5) может служить процедура получения допуска к сдаче экзамена, где, в зависимости от количества набранных баллов, осуществляется переход в одно или другое состояние.

Рис. 4.5 – Выражение действия перехода на диаграмме состояний

4.3 Составное состояние и подсостояние

Моделирование сложных объектов и систем, как правило, связано с многоуровневым представлением их состояний. В этом случае возникает необходимость детализировать отдельные состояния, сделав их составными.

Составное состояние (composite state) *— сложное состояние, которое состоит из других вложенных в него состояний.*

Составное состояние называют также состоянием-композитом. Вложенные состояния выступают по отношению к составному состоянию как подсостояния (substate). И хотя между ними имеет место отношение композиции, графически все вершины диаграммы, которые соответствуют вложенным состояниям, изображаются внутри символа составного состояния (рис. 4.6). В этом случае размеры графического символа составного состояния увеличиваются так, чтобы вместить в себя все подсостояния.

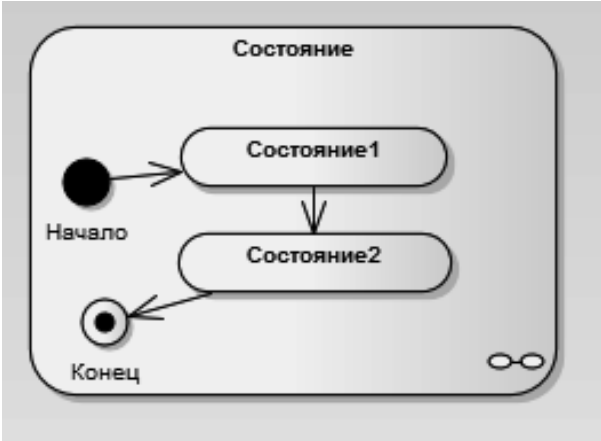


Рис. 4.6 – Графическое представление составного состояния с двумя вложенными в него последовательными подсостояниями

Составное состояние может содержать или несколько последовательных подсостояний, или несколько параллельных конечных подавтоматов. Каждое состояние-композит может уточняться только одним из указанных способов. При этом любое из подсостояний, в свою очередь, может быть состоянием-композитом и содержать внутри себя другие вложенные подсостояния. Количество уровней вложенности составных состояний в языке UML не фиксировано.

Последовательные подсостояния (sequential substates) *— вложенные состояния состояния-композита, в рамках которого в каждый момент времени объект может находиться в одном и только одном подсостоянии.*

Поведение объекта в этом случае представляет собой последовательную смену подсостояний, от начального до конечного. Моделируемый объект или система продолжает находиться в составном состоянии, тем не менее введение в рассмотрение последовательных подсостояний позволяет учесть более тонкие логические аспекты его внутреннего поведения.

В качестве примера моделируемой системы стоит рассмотреть аналоговый телефонный аппарат. Он может находиться в различных состояниях, в частности в состоянии дозвона до абонента. Очевидно, для того чтобы позвонить, необходимо снять телефонную трубку, услышать тоновый сигнал, после чего набрать нужный телефонный номер. Таким образом, состояние дозвона до абонента является составным и состоит из двух последовательных подсостояний: «Телефонная трубка поднята» и «Набор телефонного номера». Фрагмент диаграммы состояний для этого примера содержит одно состояние-композит, которое состоит из двух последовательных подсостояний (рис 4.7).

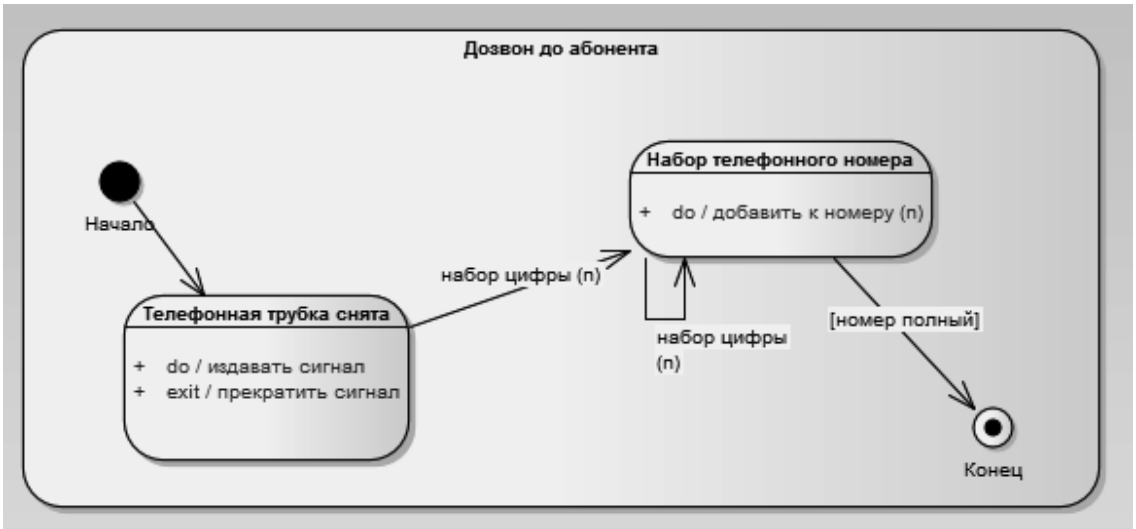


Рис. 4.7 – Пример составного состояния с двумя вложенными последовательными подсостояниями

Некоторых пояснений могут потребовать переходы. Два из них специфицируют событие-триггер, которое имеет имя: «набор цифры(*n*) с параметром *n*». В качестве параметра, как нетрудно предположить, выступает отдельная цифра на диске телефонного аппарата. Переход из начального подсостояния не содержит никакой строки текста. Последний переход в конечное подсостояние также не имеет события-триггера, но имеет сторожевое условие, проверяющее полноту набранного номера абонента. Только в случае истинности этого условия телефонный аппарат может перейти в конечное состояние для состояния-композита «Дозвон до абонента».

Каждое составное состояние должно содержать в качестве вложенных состояний начальное и конечное состояния. При этом начальное подсостояние является исходным, когда происходит переход объекта в данное составное состояние. Если составное состояние содержит внутри себя конечное состояние, то переход в это вложенное конечное состояние означает завершение нахождения объекта в данном составном состоянии. Важно помнить, что для последовательных подсостояний начальное и конечное состояния должны быть единственными в каждом составном состоянии.

Это можно объяснить следующим образом. Каждая совокупность вложенных последовательных подсостояний представляет собой конечный подавтомат того

конечного автомата, которому принадлежит рассматриваемое составное состояние. Поскольку каждый конечный автомат может иметь по определению единственное начальное и единственное конечное состояния, то для любого его конечного подавтомата это условие также должно выполняться.

В некоторых случаях бывает желательно скрыть внутреннюю структуру составного состояния. Например, отдельный конечный подавтомат, специфицирующий составное состояние, может быть настолько большим по масштабу, что его визуализация затруднит общее представление диаграммы состояний. В подобной ситуации допускается не раскрывать на исходной диаграмме состояний данное составное состояние, а указать в правом нижнем углу специальный символ-пиктограмму (рис. 4.8). В последующем диаграмма состояний для соответствующего конечного подавтомата может быть изображена отдельно от основной диаграммы с необходимыми комментариями.

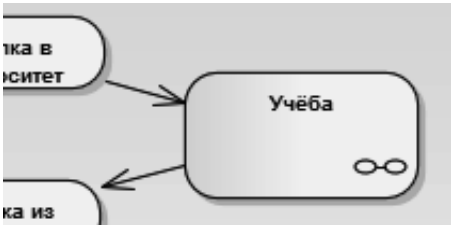


Рис. 4.8 – Составное состояние со скрытой внутренней структурой и специальной пиктограммой

4.4 Исторические состояния

Обычный конечный автомат не позволяет учитывать предысторию в процессе моделирования поведения систем и объектов. Однако функционирование ряда систем основано на возможности выхода из отдельного состояния-композита с последующим возвращением в это же состояние. Может оказаться необходимым учесть ту часть деятельности, которая была выполнена на момент выхода из этого состояния-композита, чтобы не начинать ее выполнение сначала. Для этой цели в языке UML существует историческое состояние.

Историческое состояние (history state) *— псевдосостояние, используемое для запоминания того из последовательных подсостояний, которое было текущим в момент выхода из составного состояния.*

Историческое состояние применяется только в контексте составного состояния. При этом существует две разновидности исторического состояния: неглубокое или недавнее и глубокое или давнее (рис. 4.9).

Неглубокое историческое состояние (shallow history state) обозначается в форме небольшой окружности, в которую помещена латинская буква «*H*» (рис. 4.9, *а*).

Рис. 4.9 – Графическое изображение недавнего (*а*) и давнего (*б*) исторического состояния

Это состояние обладает следующей семантикой. Во-первых, оно является первым подсостоянием в составном состоянии, и переход извне в рассматриваемое составное состояние должен вести непосредственно в данное историческое состояние. Во-вторых, при первом попадании в неглубокое историческое состояние оно не хранит никакой истории. Другими словами, при первом переходе в недавнее историческое состояние оно заменяет собой начальное состояние соответствующего конечного подавтомата.

Далее могут последовательно изменяться вложенные подсостояния. Если в некоторый момент происходит выход из составного состояния (например, в случае наступления некоторого события), то рассматриваемое историческое состояние запоминает то из подсостояний, которое было текущим на момент выхода из данного составного состояния. При последующем входе в это составное состояние неглубокое историческое подсостояние имеет непустую историю и сразу отправляет конечный подавтомат в запомненное подсостояние, минуя все предшествующие ему подсостояния.

Историческое состояние теряет свою историю в тот момент, когда конечный подавтомат доходит до своего конечного состояния. При этом неглубокое историческое состояние запоминает историю только того конечного подавтомата, к которому оно относится. Другими словами, этот тип псевдосостояния способен запомнить историю только одного с ним уровня вложенности.

Если запомненное подсостояние также является составным состоянием, а при выходе из исходного составного состояния необходимо запомнить подсостояние второго уровня вложенности, то в этом случае следует воспользоваться более сильным псевдосостоянием — глубоким историческим состоянием.

Глубокое историческое состояние (deep history state или состояние глубокой истории) также обозначается в форме небольшой окружности, в которую помещена латинская буква «*H*» с дополнительным символом «*» (рис. 4.9, *б*), и служит для запоминания всех подсостояний любого уровня вложенности для исходного составного состояния.

4.5 Сложные переходы и псевдосостояния

Рассмотренное выше понятие перехода вполне достаточно для большинства типичных расчетно-аналитических задач. Однако современные программные системы могут реализовывать сложную логику поведения отдельных своих компонентов. Иногда для адекватного представления процесса изменения состояний се-

мантика обычного перехода для них недостаточна. С этой целью в языке UML специфицированы дополнительные обозначения и свойства, которыми могут обладать отдельные переходы на диаграмме состояний.

В отдельных случаях возникает необходимость явно показать ситуацию, когда переход может иметь несколько исходных состояний или целевых состояний. Такой переход получил название — *параллельный переход*. Введение в рассмотрение параллельных переходов может быть обусловлено необходимостью синхронизировать и/или разделить отдельные процессы управления на параллельные нити без спецификации дополнительной синхронизации в параллельных конечных подавтоматах.

Графически такой переход изображается вертикальной черточкой, аналогично обозначению перехода в известном формализме сетей Петри. Если параллельный переход имеет две или более исходящих из него дуг, то его называют разделением (fork). Если же он имеет две или более входящие дуги, то его называют слиянием (join).

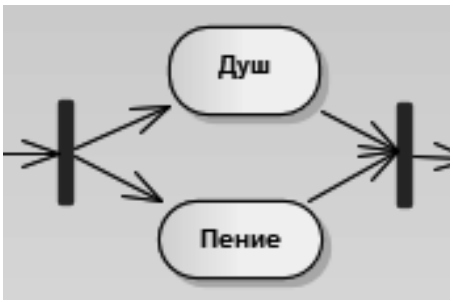


Рис. 4.10 – Параллельные подсостояния «Пение в душе»

Срабатывание параллельного перехода происходит следующим образом. В первом случае происходит разделение составного конечного автомата на два конечных подавтомата, образующих параллельные ветви вложенных подпроцессов. При этом после срабатывания перехода-разделения моделируемая система или объект одновременно будет находиться во всех целевых подсостояниях этого параллельного перехода («Пение» и «Принятие душа»). После одновремнного окончания обоих состояний поток вновь соединяется.

В общем случае поведение параллельных конечных подавтоматов происходит независимо друг от друга, что позволяет, например, моделировать многозадачность в программных системах. Однако в отдельных ситуациях может возникнуть необходимость учесть в модели синхронизацию наступления отдельных событий и срабатывание соответствующих переходов. Для этой цели в языке UML имеется псевдосостояние, которое называется синхронизирующим состоянием или состоянием синхронизации.

Состояние синхронизации (synch state) — *псевдосостояние в конечном автомате, которое используется для синхронизации параллельных областей конечного автомата.*

Синхронизирующее состояние обозначается небольшой окружностью, внутри которой помещен символ звездочки «*». Оно используется совместно с переходом-слиянием или переходом-разделением для того, чтобы явно указать события в других конечных подавтоматах, оказывающие непосредственное влияние на поведение данного подавтомата.

Так, например, при включении компьютера с некоторой сетевой операционной системой параллельно начинается выполнение нескольких процессов. В частности, происходит проверка пароля пользователя и запуск различных служб. При этом работа пользователя на компьютере станет возможной только в случае успешной его аутентификации, в противном случае компьютер может быть выключен. Рассмотренные особенности синхронизации этих параллельных процессов учтены на соответствующей диаграмме состояний с помощью синхронизирующего состояния (рис. 4.11).

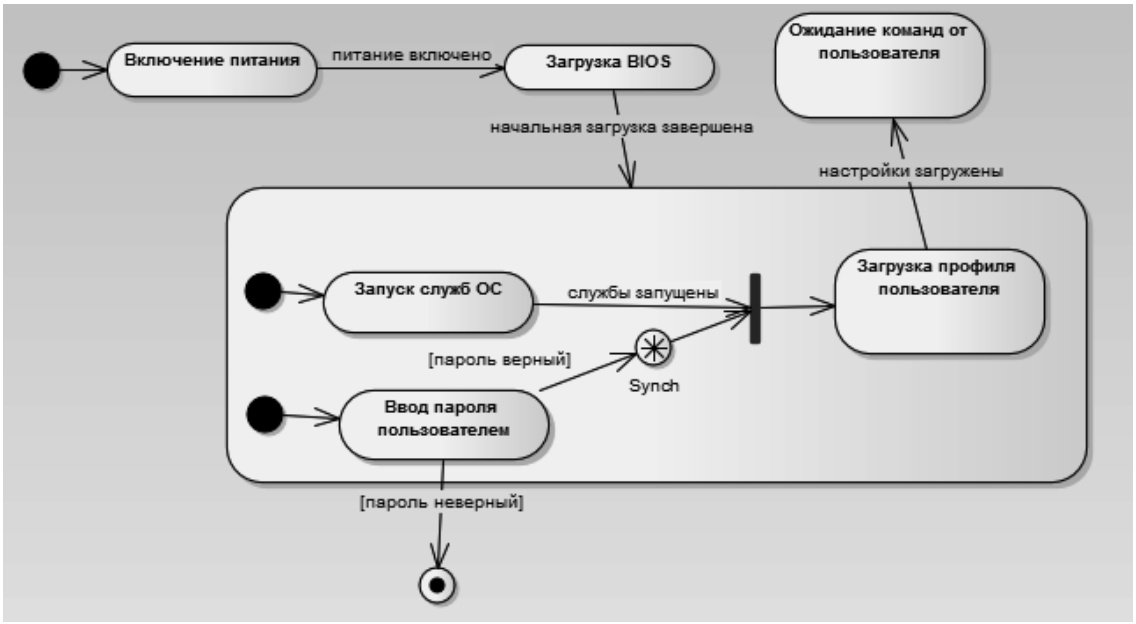


Рис. 4.11 – Диаграмма состояний для примера включения компьютера

Рекомендации по построению диаграмм состояний

По своему назначению диаграмма состояний не является обязательным представлением в модели и как бы «присоединяется» к тому элементу, который, по замыслу разработчиков, имеет нетривиальное поведение в течение своего жизненного цикла. Наличие у системы нескольких состояний, отличающихся от простой дихотомии «исправен — неисправен», «активен — неактивен», «ожидание — реакция на внешние действия», уже служит признаком необходимости построения диаграммы состояний. В качестве начального варианта диаграммы состояний, если нет очевидных соображений по поводу состояний объекта, можно воспользоваться подобными состояниями, в качестве составных, уточняя их (детализируя их внутреннюю структуру) по мере рассмотрения логики поведения моделируемой системы или объекта.

При выделении состояний и переходов следует помнить, что длительность срабатывания отдельных переходов должна быть существенно меньшей, чем нахождение моделируемых элементов в соответствующих состояниях. Каждое из состояний должно характеризоваться определенной устойчивостью во времени. Другими словами, из каждого состояния на диаграмме не может быть самопроизвольного перехода в какое бы то ни было другое состояние. Все переходы должны быть явно специфицированы, в противном случае построенная диаграмма состояний является либо неполной (неадекватной), либо ошибочной с точки зрения нотации языка UML (ill formed).

При разработке диаграммы состояний нужно постоянно следить, чтобы объект в каждый момент мог находиться только в единственном состоянии. Если это не так, то данное обстоятельство может быть как следствием ошибки, так и неявным признаком наличия параллельности поведения у моделируемого объекта. В последнем случае следует явно специфицировать необходимое число конечных подавтоматов, вложив их в то составное состояние, которое характеризуется нарушением условия одновременности.

Следует произвести обязательную проверку, чтобы никакие два перехода из одного состояния не могли сработать одновременно. Другими словами, необходимо выполнить требование отсутствия конфликтов у всех переходов, выходящих из одного и того же состояния. Наличие такого конфликта может служить признаком ошибки либо параллельности или ветвления рассматриваемого процесса. Если параллельность по замыслу разработчика отсутствует, то следует ввести дополнительные сторожевые условия либо изменить существующие, чтобы исключить конфликт переходов. При наличии параллельности следует заменить конфликтующие переходы одним параллельным переходом типа ветвления.

Использование исторических состояний оправдано в том случае, когда необходимо организовать обработку исключительных ситуаций (прерываний) без потери данных или выполненной работы. При этом применять исторические состояния, особенно глубокие, необходимо с известной долей осторожности. Нужно помнить, что каждый из конечных подавтоматов может иметь только одно историческое состояние. В противном случае возможны ошибки, особенно, когда подавтоматы изображаются на отдельных диаграммах состояний.

Введение для перехода сторожевого условия позволяет явно специфицировать семантику его срабатывания. Однако вычисление истинности сторожевого условия происходит только после возникновения ассоциированного с ним события-триггера (рис. 4.12), инициирующего соответствующий переход.

В общем случае из одного состояния может быть несколько переходов с одним и тем же событием-триггером. При этом никакие два сторожевых условия не должны одновременно принимать значение «истина». Каждое из сторожевых условий необходимо вычислять всякий раз при наступлении соответствующего события-триггера.

Один из недостатков обычных блок-схем алгоритмов связан с проблемой изображения параллельных ветвей отдельных вычислений. Поскольку распараллеливание вычислений существенно повышает общее быстродействие программных систем, необходимы графические примитивы для представления параллельных процессов. В языке UML для этой цели используется специальный символ для

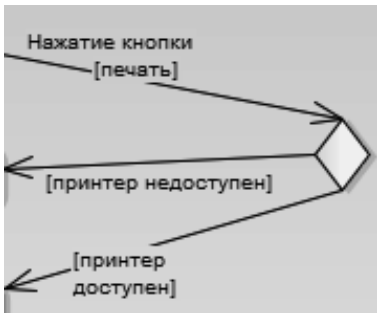


Рис. 4.12 – Событие-триггер «Доступен ли принтер?»

разделения и слияния параллельных вычислений или потоков управления. Таким символом является прямая черточка (рис. 4.10). Как правило, такая черточка изображается отрезком горизонтальной/вертикальной линии, толщина которой несколько шире основных сплошных линий диаграммы деятельности. При этом *разделение* (fork) имеет один входящий переход и несколько выходящих, а *слияние* (join) имеет несколько входящих переходов и один выходящий.

В случае, если переход из состояния в состояние сопровождается возникновением некоторой *исключительной ситуации* (exception), при использовании Enterprise Architect уместно использовать специальный блок обработки, который графически обозначается так же, как и обычный блок состояния, но со специально обозначенной точкой входа (рис. 4.13). Для обозначения факта прерывания потока данных используется специальная связь «разрыв потока» (interrupt flow).

Рис. 4.13 – Исключительная ситуация «Закончилась бумага»

Контрольные вопросы по главе 4

1) Зачем на диаграммах состояний нужны подсостояния начала и конца?
2) Для каких целей служит переход объекта из одного состояния в другое?
3) В чем отличие триггерного от нетриггерного перехода?
4) Возможно ли скрывать внутреннюю структуру подсостояния и какими средствами (если это возможно)?
5) В чём заключается особенность параллельного перехода?

Глава 5

ПРОЕКТИРОВАНИЕ БАЗ ДАННЫХ

База данных (БД) *— представленная в объективной форме совокупность самостоятельных материалов (статей, расчётов, нормативных актов, судебных решений и иных подобных материалов), систематизированных таким образом, чтобы эти материалы могли быть найдены и обработаны с помощью электронной вычислительной машины.*

Многие специалисты указывают на распространённую ошибку, состоящую в некорректном использовании термина «база данных» вместо термина «система управления базами данных», и указывают на необходимость различения этих понятий.

Система управления базами данных (СУБД) *— совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление созданием и использованием баз данных.*

Правильно разработанная, спроектированная и реализованная структура базы данных с грамотно и оперативно написанной СУБД позволяет эффективно решать колоссальный круг задач, связанный с систематизацией данных.

5.1 Нормальные формы

Нормальная форма *— свойство отношения в реляционной модели данных, характеризующее его с точки зрения избыточности, которая потенциально может привести к логически ошибочным результатам выборки или изменения данных. Нормальная форма определяется как совокупность требований, которым должно удовлетворять отношение* [1].

Процесс преобразования отношений базы данных к виду, отвечающему нормальным формам, называется ***нормализацией***.

Нормализация предназначена для приведения структуры БД к виду, обеспечивающему минимальную логическую избыточность, и не имеет целью уменьшение или увеличение производительности работы или же уменьшение или увеличение физического объёма базы данных. Конечной целью нормализации является уменьшение потенциальной противоречивости хранимой в базе данных информации. Как отмечает К. Дейт, общее назначение процесса нормализации заключается в следующем:

- исключение некоторых типов избыточности;
- устранение некоторых аномалий обновления;
- разработка проекта базы данных, который является достаточно «качественным» представлением реального мира, интуитивно понятен и может служить хорошей основой для последующего расширения;
- упрощение процедуры применения необходимых ограничений целостности.

Устранение избыточности производится, как правило, за счёт декомпозиции отношений таким образом, чтобы в каждом отношении хранились только первичные факты (то есть факты, не выводимые из других хранимых фактов).

В создании и развитии теории нормализации принимали участие многие учёные. Однако первые три нормальные формы и концепцию функциональной зависимости предложил Э. Кодд.

Первая нормальная форма (1NF)

Переменная отношения находится в первой нормальной форме тогда и только тогда, когда в любом допустимом значении отношения каждый его кортеж содержит только одно значение для каждого из атрибутов.

В реляционной модели отношение всегда находится в первой нормальной форме по определению понятия «отношение».

Что же касается различных таблиц, то они могут не быть правильными представлениями отношений и соответственно могут не находиться в 1NF. В соот-

ветствии с определением К. Дж. Дейта для такого случая, таблица нормализована (эквивалентно — находится в первой нормальной форме) тогда и только тогда, когда она является прямым и верным представлением некоторого отношения. Конкретнее, рассматриваемая таблица должна удовлетворять следующим пяти условиям:

- Нет упорядочивания строк сверху-вниз (другими словами, порядок строк не несет в себе никакой информации).
- Нет упорядочивания столбцов слева-направо (другими словами, порядок столбцов не несет в себе никакой информации).
- Нет повторяющихся строк.
- Каждое пересечение строки и столбца содержит ровно одно значение из соответствующего домена (и больше ничего).
- Все столбцы являются обычными.

«Обычность» всех столбцов таблицы означает, что в таблице нет «скрытых» компонентов, которые могут быть доступны только в вызове некоторого специального оператора взамен ссылок на имена регулярных столбцов или которые приводят к побочным эффектам для строк или таблиц при вызове стандартных операторов. Таким образом, например, строки не имеют идентификаторов кроме обычных значений потенциальных ключей (без скрытых «идентификаторов строк» или «идентификаторов объектов»). Они также не имеют скрытых временных меток.[2]

Вторая нормальная форма (2NF)

Переменная отношения находится во второй нормальной форме тогда и только тогда, когда она находится в первой нормальной форме и каждый неключевой атрибут неприводимо зависит от ее потенциального ключа.

Неприводимость означает, что в составе потенциального ключа отсутствует меньшее подмножество атрибутов, от которого можно также вывести данную функциональную зависимость. Для неприводимой функциональной зависимости часто используется эквивалентое понятие «полная функциональная зависимость».

Если потенциальный ключ является простым, то есть состоит из единственного атрибута, то любая функциональная зависимость от него является неприводимой (полной). Если потенциальный ключ является составным, то согласно определению второй нормальной формы, в отношении не должно быть неключевых атрибутов, зависящих от части составного потенциального ключа.

Вторая нормальная форма по определению запрещает наличие неключевых атрибутов, которые вообще не зависят от потенциального ключа. Таким образом, вторая нормальная форма запрещает создавать отношения как несвязанные (хаотические, случайные) наборы атрибутов [3].

Третья нормальная форма (3NF)

Переменная отношения *R* находится в 3NF тогда и только тогда, когда выполняются следующие условия:

- *R* находится во второй нормальной форме;
- ни один неключевой атрибут *R* не находится в транзитивной функциональной зависимости от потенциального ключа *R*.

Пояснения к определению:

Неключевой атрибут отношения R — это атрибут, который не принадлежит ни одному из потенциальных ключей R.

Функциональная зависимость множества атрибутов Z от множества атрибутов X (записывается $X \to Z$, произносится «икс определяет зет») является транзитивной, если существует такое множество атрибутов Y, что $X \to Y$ и $Y \to Z$. При этом ни одно из множеств X, Y и Z не является подмножеством другого, то есть функциональные зависимости $X \to Z$, $X \to Y$ и $Y \to Z$ не являются тривиальными.

Определение 3NF, эквивалентное определению Кодда, но по-другому сформулированное, дал Карло Заниоло в 1982 году. Согласно ему, переменная отношения находится в 3NF тогда и только тогда, когда для каждой из её функциональных зависимостей $X \to A$ выполняется хотя бы одно из следующих условий:

- X содержит A (то есть $X \to A$ — тривиальная функциональная зависимость);
- X — суперключ;
- A — ключевой атрибут (то есть A входит в состав потенциального ключа).

Определение Заниоло четко определяет разницу между 3NF и более строгой нормальной формой Бойса—Кодда (НФБК): НФБК исключает третье условие («A — ключевой атрибут») [4].

Нормальная форма Бойса—Кодда (BCNF)

Переменная отношения находится в BCNF тогда и только тогда, когда каждая её нетривиальная и неприводимая слева функциональная зависимость имеет в качестве своего детерминанта некоторый потенциальный ключ.

Менее формально, переменная отношения находится в нормальной форме Бойса—Кодда тогда и только тогда, когда детерминанты всех ее функциональных зависимостей являются потенциальными ключами.

Для определения BCNF следует понимать понятие функциональной зависимости атрибутов отношения.

Пусть R является переменной отношения, а X и Y — произвольными подмножествами множества атрибутов переменной отношения R. Y функционально зависимо от X тогда и только тогда, когда для любого допустимого значения переменной отношения R, если два кортежа переменной отношения R совпадают по значению X, они также совпадают и по значению Y. Подмножество X называют детерминантом, а Y — зависимой частью.

Функциональная зависимость тривиальна тогда и только тогда, когда ее правая (зависимая) часть является подмножеством ее левой части (детерминанта).

Ситуация, когда отношение будет находиться в 3NF, но не в BCNF, возникает, например, при условии, что отношение имеет два (или более) потенциальных ключа, которые являются составными и имеют общий атрибут. На практике такая ситуация встречается достаточно редко, для всех прочих отношений 3NF и BCNF эквивалентны [8].

Четвёртая нормальная форма (4NF)

Переменная отношения R находится в четвёртой нормальной форме, если она находится в НФБК и все нетривиальные многозначные зависимости фактически являются функциональными зависимостями от её потенциальных ключей.

Эквивалентная формулировка определения:

Переменная отношения R находится в четвёртой нормальной форме тогда и только тогда, когда в случае существования таких подмножеств A и B атрибутов этой переменной отношения R, для которых выполняется нетривиальная многозначная зависимость $A \rightarrow\rightarrow B$, все атрибуты переменной отношения R также функционально зависят от A [5].

Пятая нормальная форма (5NF)

Переменная отношения находится в пятой нормальной форме (иначе — в проекционно-соединительной нормальной форме) тогда и только тогда, когда каждая нетривиальная зависимость соединения в ней определяется потенциальным ключом (ключами) этого отношения [6].

Доменно-ключевая нормальная форма (DKNF)

Переменная отношения находится в ДКНФ тогда и только тогда, когда каждое наложенное на неё ограничение является логическим следствием ограничений доменов и ограничений ключей, наложенных на данную переменную отношения.

Ограничение домена *— ограничение, предписывающее использовать для определённого атрибута значения только из некоторого заданного домена.*

Ограничение по своей сути является заданием перечня (или логического эквивалента перечня) допустимых значений типа и объявлением о том, что указанный атрибут имеет данный тип.

Ограничение ключа *— ограничение, утверждающее, что некоторый атрибут или комбинация атрибутов является потенциальным ключом.*

Любая переменная отношения, находящаяся в ДКНФ, обязательно находится в 5НФ. Однако не любую переменную отношения можно привести к ДКНФ [9].

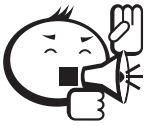
Шестая нормальная форма (6NF)

Переменная отношения находится в шестой нормальной форме тогда и только тогда, когда она удовлетворяет всем нетривиальным зависимостям соединения.

Из определения следует, что переменная находится в 6НФ тогда и только тогда, когда она неприводима, то есть не может быть подвергнута дальнейшей декомпозиции без потерь. Каждая переменная отношения, которая находится в 6НФ, также находится и в 5НФ [7].

Правила разработки структуры базы данных

Для быстрой проработки структуры базы данных, как правило, достаточно создать полную атрибутивную модель, т. е. модель, содержащую все сущности в третьей нормальной форме (3НФ) со всеми атрибутами и связями. Более запоминающееся и наглядное резюме определения 3НФ, чем то, которое было приведено выше, дано Биллом Кентом:



Каждый неключевой атрибут «должен предоставлять информацию о ключе, полном ключе и ни о чем, кроме ключа».

Условие зависимости от «полного ключа» неключевых атрибутов обеспечивает то, что таблица находится во второй нормальной форме; а условие зависимости их от «ничего, кроме ключа» — то, что они находятся в третьей нормальной форме.

Описание структуры следует начинать с создания сущностей — будущих таблиц, а пока кандидатов на роль таблиц БД.

В соответствии с методологией проектирования структуры базы данных, для каждой сущности требуется задать номер, имя сущности, ввести определение и описание сущности.

Следующим шагом является заполнение сущностей *атрибутами* и выделение *ключевых атрибутов*.

5.2 Ключи и внешние ключи

Потенциальный ключ *— в реляционной модели данных подмножество атрибутов отношения, удовлетворяющее требованиям уникальности и минимальности (несократимости).*

Уникальность означает, что не существует двух кортежей данного отношения, в которых значения этого подмножества атрибутов совпадают (равны).

Минимальность (несократимость) означает, что в составе потенциального ключа отсутствует меньшее подмножество атрибутов, удовлетворяющее условию уникальности. Иными словами, если из потенциального ключа убрать любой атрибут, он утратит свойство уникальности.

Поскольку все кортежи в отношении по определению уникальны, в нём всегда существует хотя бы один потенциальный ключ (например, включающий все атрибуты отношения).

В отношении может быть одновременно несколько потенциальных ключей. Один из них может быть выбран в качестве первичного ключа отношения, тогда другие потенциальные ключи называют альтернативными ключами.

Теоретически, все потенциальные ключи равно пригодны в качестве первичного ключа, на практике в качестве первичного обычно выбирается тот из потенциальных ключей, который имеет меньший размер (физического хранения) и/или включает меньшее количество атрибутов.

Первичный ключ (primary key) *— в реляционной модели данных один из потенциальных ключей отношения, выбранный в качестве основного ключа (или ключа по умолчанию).*

Если в отношении имеется единственный потенциальный ключ, он является и первичным ключом. Если потенциальных ключей несколько, один из них выбирается в качестве первичного, а другие называют «альтернативными».

С точки зрения теории все потенциальные ключи отношения эквивалентны, то есть обладают одинаковыми свойствами уникальности и минимальности. Однако в качестве первичного обычно выбирается тот из потенциальных ключей, который наиболее удобен для тех или иных практических целей, например для создания внешних ключей в других отношениях либо для создания кластерного индекса. Поэтому в качестве первичного ключа, как правило, выбирают тот, который имеет наименьший размер (физического хранения) и/или включает наименьшее количество атрибутов.

Другой критерий выбора первичного ключа — сохранение уникальности со временем. Всегда существует вероятность того, что некоторый потенциальный ключ перестанет быть таковым в долговременной перспективе или при изменении требований к системе. Например, если номер студенческой группы включает последнюю цифру года поступления, то номера групп для идентификации групп уникальны только в течение 10 лет. Поэтому в качестве первичного ключа стараются выбирать такой потенциальный ключ, который с наибольшей вероятностью не утратит уникальность.

Исторически термин «первичный ключ» появился и стал использоваться существенно ранее термина «потенциальный ключ». Вследствие этого множество определений в реляционной теории были изначально сформулированы с упоминанием первичного (а не потенциального) ключа, например определения нормальных форм. Также термин «первичный ключ» вошёл в формулировку 12 правил Кодда как основной способ адресации любого значения отношения (таблицы) наряду с именем отношения (таблицы) и именем атрибута (столбца).

Если первичный ключ состоит из единственного атрибута, его называют простым ключом.

Если первичный ключ состоит из двух и более атрибутов, его называют составным ключом. Так, номер паспорта и серия паспорта не могут быть первичными ключами по отдельности, так как могут оказаться одинаковыми у двух и более людей. Но не бывает двух личных документов одного типа с одинаковыми серией и номером. Поэтому в отношении, содержащем данные о людях, первичным клю-

чом может быть подмножество атрибутов, состоящее из типа личного документа, его серии и номера.

Внешний ключ (foreign key) *— понятие теории реляционных баз данных, относящееся к ограничениям целостности базы данных.*

Неформально выражаясь, внешний ключ представляет собой подмножество атрибутов некоторой переменной отношения $R2$, значения которых должны совпадать со значениями некоторого потенциального ключа некоторой переменной отношения $R1$.

Формальное определение. Пусть $R1$ и $R2$ — две переменные отношения, не обязательно различные. Внешним ключом FK в $R2$ является подмножество атрибутов переменной $R2$, такое, что выполняются следующие требования:

В переменной отношения $R1$ имеется потенциальный ключ CK, такой, что FK и CK совпадают с точностью до переименования атрибутов (то есть переименованием некоторого подмножества атрибутов FK можно получить такое подмножество атрибутов FK', что FK' и CK совпадают как по именам, так и по типам атрибутов).

В любой момент времени каждое значение FK в текущем значении $R2$ идентично значению CK в некотором кортеже в текущем значении $R1$. Иными словами, в каждый момент времени множество всех значений FK в $R2$ является (нестрогим) подмножеством значений CK в $R1$.

При этом для данного конкретного внешнего ключа $FK \rightarrow CK$ отношение $R1$, содержащее потенциальный ключ, называют главным, целевым, или родительским отношением, а отношение $R2$, содержащее внешний ключ, называют подчинённым, или дочерним отношением.

Поддержка внешних ключей также называется соблюдением ссылочной целостности. Реляционные СУБД поддерживают автоматический контроль ссылочной целостности.

После определения ключевых атрибутов необходимо определить домены, на которых определены данные атрибуты. Домены будут использоваться при определении типа колонки на уровне физической модели. Необходимо отметить, что на одном домене может быть задано сразу несколько атрибутов (рис. 5.1).

Как правило, необходимо задать четыре параметра:

- имя домена,
- родительский домен,
- базовый тип данных домена,
- проверку на значения атрибутов домена.

После создания сущностей, определения атрибутов и задания их типов необходимо определить между ними *связи*.

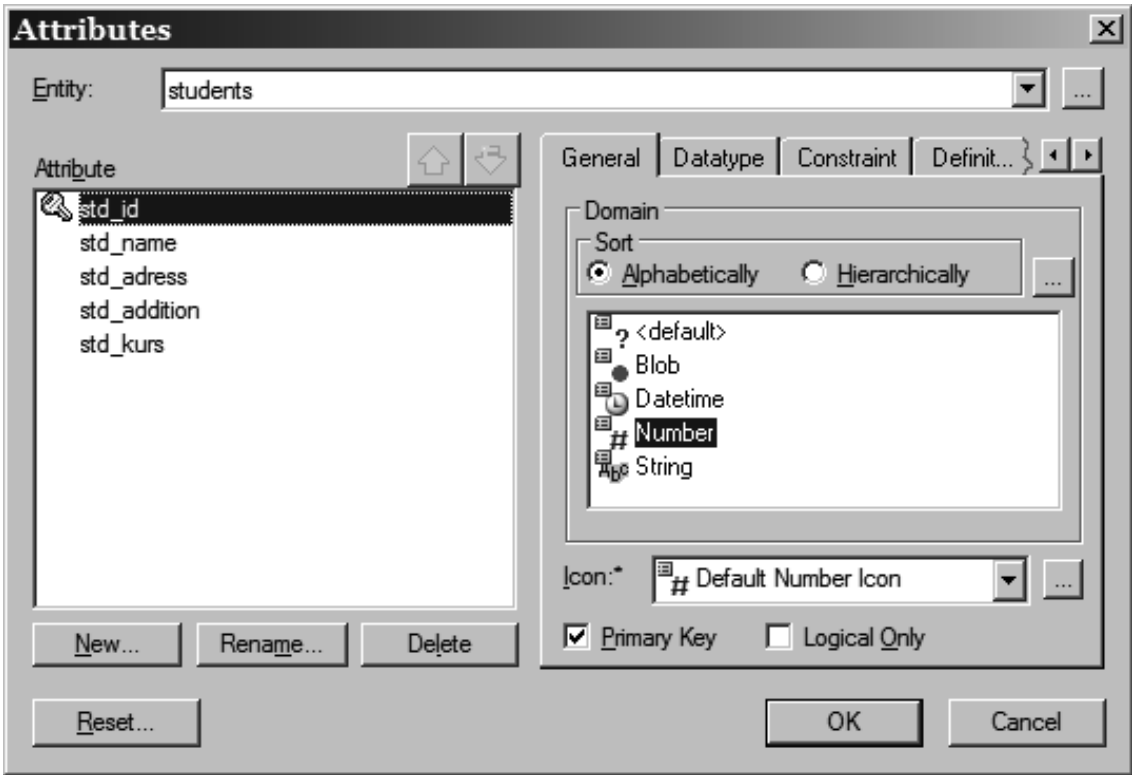


Рис. 5.1 – Определение домена для атрибута на примере ErWin

5.3 Типы связей

Графические представления различных типов связей приведены в табл. 5.1.

Таблица 5.1 – Типы связей и их графическое отображение в ErWin

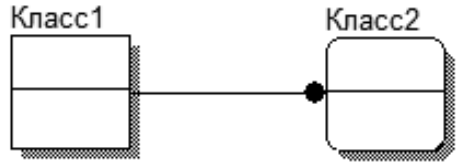
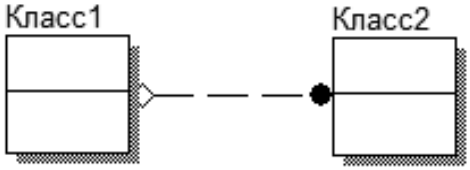
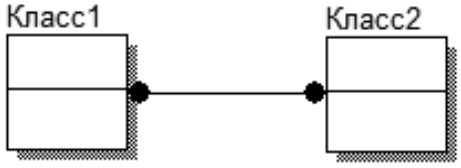
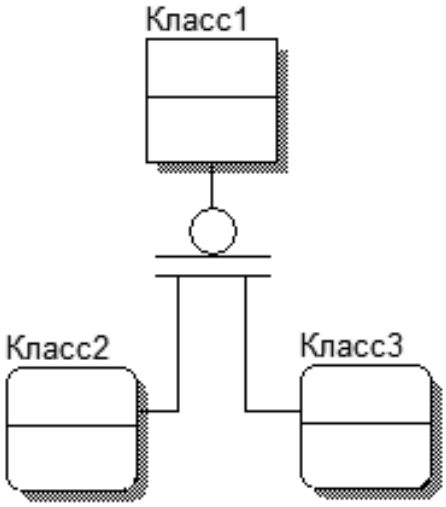
	Идентифицирующая связь
	Неидентифицирующая связь
	Связь типа «многие-ко-многим»
продолжение на следующей странице	

Таблица 5.1 — Продолжение

	Связь супертипа с подтипами

Дадим определение данным типам связей.

Идентифицирующая связь *изображается сплошной линией. Сущность-потомок в идентифицирующей связи является зависимой сущностью, т. е. когда первичный ключ дочерней сущности содержит внешний ключ, идущий от родительской сущности. Сущность-родитель в идентифицирующей связи может быть как независимой, так и зависимой от идентификатора сущностью (это определяется ее связями с другими сущностями).*

Рассмотрим простой пример: пусть даны две сущности — ВОПРОС и ОТВЕТ. Связь ВОПРОС-ОТВЕТ является идентифицирующей, поскольку сущность ОТВЕТ не может быть однозначно определена, если не задана сущность ВОПРОС.

Неидентифицирующая связь *изображается пунктирной линией. Сущность-потомок в неидентифицирующей связи будет независимой от идентификатора, если она не является также сущностью-потомком в какой-либо идентифицирующей связи.*

Например, есть сущность СТУДЕНТ и сущность АККАУНТ НА BLIZZARD.NET. Безусловно, конкретный студент может обладать аккаунтом на сайте игровой компании — но вовсе не каждый, и для определения конктретного аккаунта возможно использовать и иные способы определения принадлежности, нежели паспортные данные конкретного студента. Таким образом, существование данных сущностей никак друг с другом не связано.

После определения связей становится возможным явным образом определить кардинальность (мощность связи) (рис. 5.2).

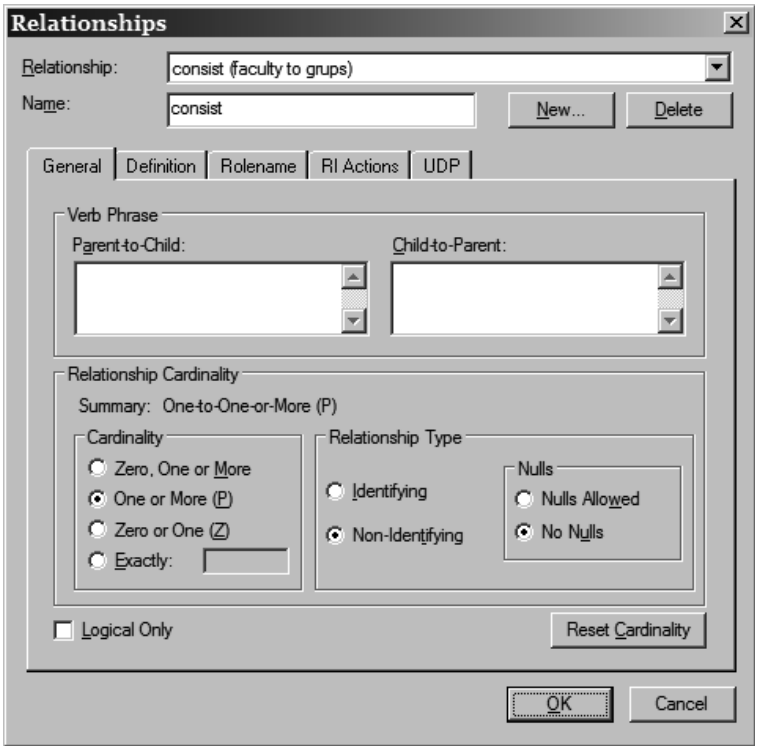


Рис. 5.2 – Окно настроек свойств связи на примере ErWin

Определяется кардинальное число связи на стороне дочерней сущности (область *Cardinality*). Кардинальное число может быть заданно как:

- 0, 1 или более (*Zero, One or More*);
- 1 или более (*One or More*);
- 0 или 1 (*Zero or One*);
- Точное значение (*Exactly*).

Важным шагом при построении полноатрибутной модели является шаг определения всех атрибутов для сущностей, выявленных на предыдущих этапах моделирования. Для задания атрибутов и их доменов необходимо использовать подход, аналогичный подходу, изложенному выше. Каждый атрибут должен быть определен на соответствующем домене. У разных атрибутов домены могут совпадать. После определения всех атрибутов необходимо задать правила поддержки *ссылочной целостности*.

Связи между данными, хранимыми в разных отношениях, в реляционной БД устанавливаются с помощью использования внешних ключей — для установления связи между кортежем из отношения *A* с определённым кортежем отношения *B* в предусмотренные для этого атрибуты кортежа отношения *A* записывается значение первичного ключа (а в общем случае значение потенциального ключа) целевого кортежа отношения *B*. Таким образом, всегда имеется возможность выполнить две операции:

- определить, с каким кортежем в отношении *B* связан определённый кортеж отношения *A*;
- найти все кортежи отношения *A*, имеющие связи с определённым кортежем отношения *B*.

Благодаря наличию связей в реляционной БД можно хранить факты без избыточного дублирования, то есть в нормализованном виде. Ссылочная целостность может быть проиллюстрирована следующим образом:

Дана пара отношений *A* и *B*, связанных внешним ключом. Первичный ключ отношения *B* — атрибут *B.key*. Внешний ключ отношения *A*, ссылающийся на *B* — атрибут *A.b*. Ссылочная целостность для пары отношений *A* и *B* имеет место тогда, когда выполняется условие: для каждого кортежа отношения *A* существует соответствующий кортеж отношения *B*, то есть кортеж, у которого (*B.key* = *A.b*).

База данных обладает свойством ссылочной целостности, когда для любой пары связанных внешним ключом отношений в ней условие ссылочной целостности выполняется.

Если вышеприведённое условие не выполняется, говорят, что в базе данных нарушена ссылочная целостность. Такая БД не может нормально эксплуатироваться, так как в ней разорваны логические связи между зависимыми друг от друга фактами. Непосредственным результатом нарушения ссылочной целостности становится то, что корректным запросом не всегда удаётся получить корректный результат.

На заключительном этапе проектирования следует создать описание модели. Например, для полноатрибутной модели на рис. 5.3 описание будет соответствовать таблицам 5.2–5.7.

Таблица 5.2 – Справочник students

Название	Описание	Тип данных
std_id	Поле — счетчик	INTEGER
std_name	Имя студента	VARCHAR (45)
std_adress	Адрес студента	VARCHAR (100)
std_addition	Поле для примечаний	VARCHAR (100)
std_kurs	Курс студента	CHAR(5)

Таблица 5.3 – Справочник grups

Название	Описание	Тип данных
grup_id	Поле — счетчик	INTEGER
date_create	Дата создания группы	DATE

Таблица 5.4 – Справочник faculty

Название	Описание	Тип данных
fac_id	Поле — счетчик	INTEGER
fac_name	Название факультета	VARCHAR (100)

Таблица 5.5 – Справочник prepods

Название	Описание	Тип данных
prep_id	Поле — счетчик	счетчик
prep_name	Имя преподавателя	VARCHAR (45)
prep_adress	Адрес преподавателя	VARCHAR (100)
prep_persinf	Персональные данные преподавателя	VARCHAR (100)
prep_inn	ИНН преподавателя	VARCHAR (12)

Таблица 5.6 – Справочник predmets

Название	Описание	Тип данных
Pred_id	Поле — счетчик	счетчик
Pred_name	Название предмета	VARCHAR (100)
Pred_kurs	Курс, на котором ведётся предмет	CHAR (5)

Таблица 5.7 – Справочник rate

Название	Описание	Тип данных
rate_id	Поле — счетчик	счетчик
rate_max	Максимальный рейтинг	INTEGER
rate_fact	Фактический рейтинг	INTEGER

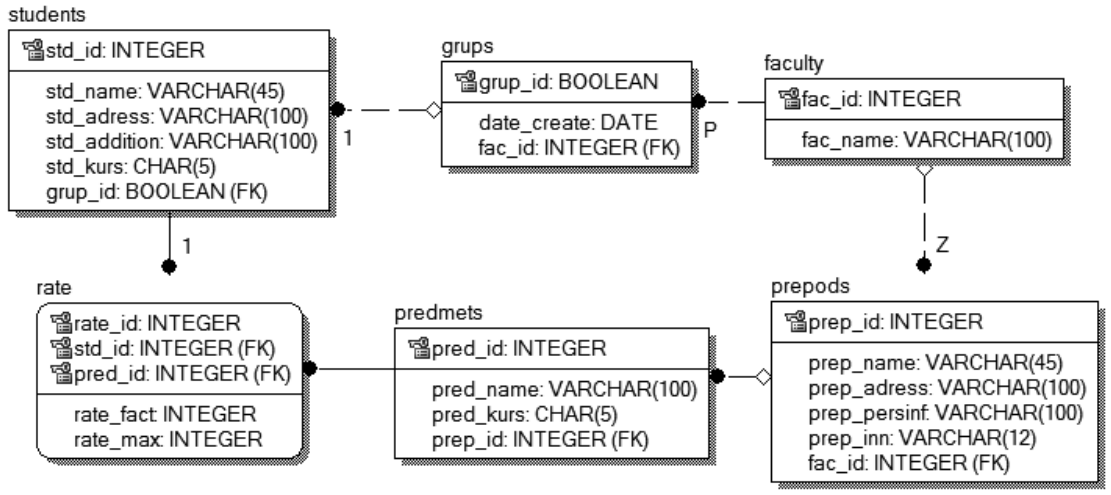


Рис. 5.3 – ER-диаграмма базы данных

Создание модели позволяет приступать к физической реализации базы данных, например с помощью языка MySQL.

5.4 Основные запросы языка SQL

Структурированный язык запросов (Structured Query Language) — стандарт коммуникации с базой данных, который поддержан ANSI. В общих терминах, «SQL база данных» является общим названием для реляционной системы управления базами данных (РСУБД). Для некоторых систем «база данных» также относится к группе таблиц, данных, конфигурационной информации, которые являются неотъемлемо отдельной частью от других, подобных конструкций. В этом случае, каждая инсталляция SQL базы данных может состоять из нескольких баз данных. В других системах они упомянуты как таблицы.

Таблица *— конструкция базы данных, которая состоит из столбцов, содержащих строки данных.*

Обычно таблицы созданы для того, чтобы содержать связанную информацию. В пределах той же самой базы данных могут быть созданы несколько таблиц.

Каждый столбец представляет собой атрибут или совокупность атрибутов объектов, например идентификационные номера служащих, рост, цвет машин и т. п. Часто в отношении столбца используется термин «поле» с указанием имени, например «в поле Name». Поле строки является минимальным элементом таблицы. Каждый столбец в таблице имеет определенное имя, тип данных и размер. Имена столбцов должны быть уникальны в пределах таблицы.

Каждая строка (или запись) представляет собой совокупность атрибутов конкретного объекта, например в строке может содержаться идентификационный номер служащего, размер его зарплаты, год его рождения и т. д. Строки таблиц не имеют названий. Чтобы обратиться к конкретной строке, пользователю необходимо указать какой-то атрибут (или набор атрибутов), уникально ее идентифицирующий.

Одной из важнейших операций, которые выполняются при работе с данными, является выборка хранящейся в базе данных информации. Для этого пользователь должен выполнить запрос (query).

Рассмотрим основные типы запросов к базе данных, которые сосредоточены на манипуляции данными в пределах базы (таб. 5.8–5.10).

Таблица 5.8 – Управление данными (Data Manipulation Language, DML)

DELETE	Удаление записи или нескольких записей
INSERT	Добавление записи или нескольких записей
LOAD DATA INFILE	Добавление в таблицу записей из файла
REPLACE	Добавление записи или записей с заменой
SELECT	Выборка записей
SELET JOIN	Варианты выборки записей из нескольких таблиц
TRUNCATE	Удаление таблицы и создание аналогичной новой
UPDATE	Изменение записей

Таблица 5.9 – Определение данных (Data Definition Language, DDL)

ALTER TABE	Изменение структуры таблицы
CREATE DATABASE	Создание базы данных
CREATE INDEX	Создание индекса
CREATE TABLE	Создание таблицы
DROP DATABASE	Удаление базы данных
DROP INDEX	Удаление индекса
DROP TABLE	Удаление таблицы
RENAME TABLE	Переименование таблицы

Таблица 5.10 – Служебные команды (Service)

DESCRIBE	Отображает информацию о полях таблицы
SHOW	Команда отображает информацию о базах данных, таблицах, полях или о состоянии сервера
USE	Назначает текущую базу данных

Остановимся более подробно на разборе синтаксиса, приведенных выше команд.

CREATE DATABASE

CREATE DATABASE [IF EXISTS] db_name

Создает базу с указанным именем. Если указан параметр `IF EXISTS`, то MySQL не будет сообщать об ошибке («Существует база данных с таким именем»). База данных в MySQL физически представляет собой папку, в которой хранятся файлы описания, индексов и данных таблиц. Так как таблиц еще нет, то данная команда просто создает пустую директорию.

Пример

```
CREATE DATABASE u4eba;
```

DROP DATABASE

DROP DATABASE [IF EXISTS] db_name

Удаляет все таблицы в базе и саму базу данных. Если команда выполняется над ссылкой на базу данных, то удаляются и ссылка, и база.

Будьте ОЧЕНЬ осторожны с этой командой!

Возвращает количество удаленных файлов в папке базы данных. Обычно это утроенное количество таблиц, так как для хранения таблиц нужно обычно три файла («table_name.frm», «table_name.myd», «table_name.myi»). Команда удаляет все файлы в папке базы данных следующих типов: *.BAK, *.DAT, *.db, *.frm, *.HSH, *.ISD, *.ISM, *.MRG, *.MYD, *.MYI.

Удаляются также все вложенные папки с именем из 2-х символов (папки RAID). Начиная с версии 3.22, можно использовать ключевое слово `IF EXISTS` для того, чтобы MySQL не сообщал об ошибке удаления базы данных, которой не существует.

Пример

```
DROP DATABASE u4eba;
```

CREATE TABLE

CREATE [TEMPORARY]
TABLE [IF NOT EXISTS] tbl_name
[(create_definition,...)] [table_options]

Команда создает таблицу с именем `tbl_name` в текущей БД. Если не выбрана БД или таблица с выбранным именем существует, то происходит ошибка. Начиная с MySQl версии 3.22, указать имя таблицы, используя синтаксис `имя_БД.имя_таблицы`, причем при таком синтаксисе можно не выбирать текущую БД.

Каждая таблица хранится в нескольких файлах. Например, для таблиц типа MyISAM эти файлы следующие:

- *.frm — файл определения таблицы;
- *.MYD — файл данных таблицы;
- *.MYI — файл индексов таблицы.

Пример

```
CREATE TABLE u4eba.student (
`std_id` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
`std_name` VARCHAR(45) NOT NULL,
`std_adress` VARCHAR(100),
`std_formofstudy` CHAR(3) NOT NULL,
`std_kurs` INTEGER UNSIGNED NOT NULL,
PRIMARY KEY (`std_id`)
);
```

DROP TABLE

DROP TABLE [IF EXISTS] tbl_name [, tbl_name,…] [RESTRICT | CASCADE]

Удаляет таблицу или несколько таблиц. Все их данные и определения будут уничтожены, так что будьте осторожны с этой командой. Начиная с версии 3.22, можно указать ключевое слово `IF EXISTS`, для того чтобы СУБД не выдала ошибку, если таблицы с таким именем нет. `RESTRICT` и `CASCADE` существуют для совместимости. Пока они ничего не делают. Команда небезопасна для транзакций, поэтому она автоматически зафиксирует все активные транзакции.

Пример

```
DROP TABLE student;
```

CREATE INDEX

CREATE [UNIQUE|FULLTEXT]
INDEX index_name
ON tbl_name (col_name[(length)],…)

Индексирует поле `col_name` в таблице `tbl_name`. Индексировать можно только поля, которые могут принимать значение NULL, или поля типов BLOB/TEXT, если версия MySQL 3.23.2 или новее и таблица типа MyISAM. Команда ничего не делает вплоть до версии MySQL 3.22. В версии 3.22 и более поздних команда `CREATE INDEX` привязана к команде `ALTER TABLE`.

Обычно все индексы создаются вместе с таблицей (см. команду `CREATE TABLE`). С помощью `CREATE INDEX` индексы добавляются к существующей таблице.

Пример

```
CREATE INDEX nameInd ON u4eba.student (std_name);
```

INSERT

INSERT INTO table_name (col1,[col2,…]) VALUES (val1,[val2,…])

Команда вставляет новую запись в существующую таблицу. Вид команды `INSERT … VALUES` вставляет значения, заданные явным образом, а синтаксис `INSERT … SELECT` вставляет записи из другой таблицы (или из других таблиц).

Синтаксис `INSERT … VALUES` с возможностью вставить сразу несколько записей поддерживается в MySQL, начиная с версии 3.22.5, а синтаксис `SET col_name=expression, col_name=expression, …` — начиная с версии 3.22.10.

Пример

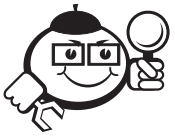
```
INSERT INTO u4eba.student VALUES (
1,
"Sidorov K." ,
"Tomsk, Lytkina str., 12-200" ,
0,
1,
"510-1"
);
```

ALTER TABLE

ALTER [IGNORE] TABLE tbl_name alter_spec [, alter_spec ...]

Команда позволяет изменять структуру существующей таблицы, например добавлять и удалять поля, создавать и уничтожать индексы, менять тип существующего поля, переименовать поля и саму таблицу. Для использования команды необходимо иметь привилегии `ALTER, INSERT` и `CREATE` для таблицы.

При изменении таблицы создается временная копия таблицы, над которой и производятся все изменения, после чего оригинал таблицы удаляется, а временная таблица переименовывается. Все операции реализованы таким образом, что все изменения записей происходят в новой таблице без потерянных изменений. Пока происходит выполнение `ALTER TABLE`, старая таблица доступна для чтения другим клиентам. Любые изменения с записями таблицы откладываются, чтобы выполняться над измененной таблицей.



Пример

```
ALTER TABLE u4eba.student ADD std_primechanie VARCHAR(150),
ADD std_inn INTEGER, DROP std_formofstudy;
```

SELECT

SELECT select_expression,...
[FROM table_references [WHERE where_definition] [ORDER BY {unsigned_integer | col_name | formula} [ASC | DESC] ,...]]

Команда используется для того, чтобы извлечь записи из одной или нескольких таблиц. Все операторы должны встречаться в такой же последовательности, как показано выше. Например, секция `HAVING` должна идти после секции `GROUP BY` и перед секцией `ORDER BY`.

Пример

```
SELECT * FROM u4eba.student ORDER BY std_name;
```

REANAME TABLE

RENAME TABLE tbl_name TO new_table_name[, tbl_name2 TO new_table_name2, …]

Переименовывает таблицу. Переименование автоматическое, это означает, что больше ни один процесс не будет иметь доступ ни к одной из таблиц, имена которых изменяются, поэтому возможно, например, заменить таблицу с записями пустой таблицей:

```
CREATE TABLE new_table (…)
RENAME TABLE old_table TO backup_table, new_table TO
old_table
```

или поменять местами имена двух таблиц (выполнение команды ведется слева направо):

```
RENAME TABLE old_table TO backup_table, new_table TO
old_table, backup_table TO new_table
```

Если две базы данных находятся на одном диске:

```
RENAME TABLE current_database.table_name, TO other_database.
table_name
```

Для выполнения команды не должно быть заблокированных таблиц или активных транзакций. У вас также должны быть привелегии `ALTER` и `DROP` на переименовываемую таблицу и привелегии `CREATE` и `INSERT` для создания новой таблицы. Если во время переименования нескольких таблиц произойдет ошибка, то ни одна таблица не будет переименована.

Пример

```
RENAME TABLE u4eba.student TO studenty;
```

DROP INDEX

DROP INDEX index_name ON tbl_name

Удаляет индекс с именем `index_name` в таблице `tbl_name`. Команда ничего не делает до версии 3.22. В более поздних версиях команда аналогична команде `ALTER TABLE` с синтаксисом `DROP INDEX`.

Пример

```
DROP INDEX nameInd ON u4eba.student;
```

LOAD DATA INFILE

LOAD DATA [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE 'file_name.txt' [REPLACE | IGNORE] INTO TABLE tbl_name [FIELDS [TERMINATED BY '\t'] [[OPTIONALLY] ENCLOSED BY ''] [ESCAPED BY '\\']] [LINES TERMINATED BY '\n'] [IGNORE number LINES] [(col_name, ...)]

Команда считывает записи из текстового файла и добавляет их в таблицу с большой скоростью. Записи в файле должны быть разделены табулятором и не содержать пустых строк.

Пример

```
LOAD DATA LOCAL INFILE "c:\\list.txt" INTO TABLE student;
```

DELETE

DELETE FROM table_name [WHERE where_definition]

Команда удаляет записи из таблицы `table_name`, которые подходят под условия, указанные в выражении `where_definition`, и возвращает количество удаленных записей.

Если условие `WHERE` пропущено, удаляются все записи. Если записи удаляются в режиме `AUTOCOMMIT`, команда работает как `TRUNCATE`. В этом случае команда `DELETE` возвращает ноль как количество удаленных записей, но эта ошибка будет исправлена в версии 4.0.

Если необходимо узнать число записей, удаленных при очистке таблицы, можно использовать следующий синтаксис:

```
DELETE FROM tbl_name WHERE 1>0
```

Обратите внимание, что подобный способ работает гораздо медленнее, чем запрос без условий `WHERE`, так как в последнем случае записи уничтожаются за один запрос.

Пример

```
DELETE FROM u4eba.student WHERE std_kurs=1;
```

SELECT JOIN

SELECT table_reference, table_reference
table_reference [CROSS] JOIN table_reference
table_reference INNER JOIN table_reference join_condition
table_reference STRAIGHT_JOIN table_reference
table_reference LEFT [OUTER] JOIN table_reference join_condition
table_reference LEFT [OUTER] JOIN table_reference
table_reference NATURAL [LEFT [OUTER]] JOIN table_reference
{oj table_reference LEFT OUTER JOIN table_reference ON conditional_expr}
table_reference RIGHT [OUTER] JOIN table_reference join_condition
table_reference RIGHT [OUTER] JOIN table_reference
table_reference NATURAL [RIGHT [OUTER]] JOIN table_reference

Осуществляет выборку из нескольких таблиц

Пример

```
SELECT * FROM student LEFT JOIN predmet ON
student.std_kurs = predmet.pred_kurs;
```

USE

USE db_name

Назначает базу данных `db_name` текущей для последующих запросов. База данных будет текущей до завершения соединения либо до назначения текущей другой базы данных.

Если вы сделали какую-то базу данных текущей, то это не означает, что вы не можете обращаться к другим базам данных.

Пример

```
USE other_base;
SELECT count(*) FROM other_table;
USE u4eba;
SELECT count(*) FROM student;
```

TRUNCATE

TRUNCATE TABLE table_name

Существует в версиях 3.23. Команда подобна запросу `DELETE FROM table_name`. Различия между ними:

- `TRUNCATE` выполнена как удаление таблицы и создание ее заново, поэтому выполняется гораздо быстрее, чем `DELETE`, когда в таблице много записей;
- `TRUNCATE` автоматически завершит текущую транзакцию, точно так же, как если бы выполнилась команда `COMMIT`;
- `TRUNCATE` не завершает количество удаленных записей.

Пока существует неповрежденный файл `'table_name.frm'`, таблица `table_name` может быть создана заново, даже если повреждены файл данных или индексный файл.

Пример

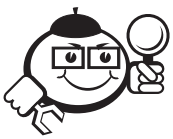
```
TRUNCATE TABLE u4eba.predmet;
```

REPLACE

REPLACE [LOW_PRIORITY | DELAYED] [INTO] tbl_name SET col_name=expression, col_name=expression, …

Команда работает точно так же, как `INSERT`, за исключением того, что если в уникальном индексе есть то же значение, что и в новой записи, то перед тем, как вставить новую запись, старая будет удалена.

Другими словами, Вы не можете получить значение старой записи с помощью этой команды. В некоторых старых версиях было похоже, что это можно сделать, но то была ошибка, которая уже исправлена.



Пример

```
REPLACE INTO u4eba.student VALUES (
6,
"Vasilkov V." ,
"Tomsk, Lytkina str., 12-201" ,
1,
"510-1" ,
" " ,
7012);
```

UPDATE

UPDATE table_name
SET col_name1=expr1, [col_name2=expr2, …]
[WHERE where_definition]
Заменяет значения в полях таблицы новыми значениями.

Пример

```
UPDATE u4eba.student SET
u4eba.std_kurs=3 WHERE u4eba.std_kurs=2;
```

DESCRIBE

{DESCRIBE | DESC} tbl_name {col_name | wild}
Это короткая запись команды `SHOW COLUMNS FROM`
Команда отображает информацию о полях таблицы. `Col_name` может быть именем поля или строкой, содержащей специальные символы SQL «%» и «_».

Пример

```
DESCRIBE u4eba.student;
```

SHOW

SHOW DATABASES [LIKE wild]
SHOW [OPEN] TABLES [FROM db_name] [LIKE wild]
SHOW [FULL] COLUMNS FROM tbl_name [FROM db_name] [LIKE wild]
SHOW INDEX FROM tbl_name [FROM db_name]
SHOW TABLE STATUS [FROM db_name] [LIKE wild]
SHOW STATUS [LIKE wild]
SHOW VARIABLES [LIKE wild]
SHOW LOGS
SHOW [FULL] PROCESSLIST
SHOW GRANTS FOR user
SHOW CREATE TABLE table_name
SHOW MASTER STATUS
SHOW MASTER LOGS
SHOW SLAVE STATUS

Команда отображает информацию о базах данных, таблицах, полях или о состоянии сервера. Если используется необязательный параметр, `LIKE wild` должен содержать строку, в которой используются специальные символы SQL «%» и «_».

SHOW DATABASES показывает список всех баз данных на сервере MySQL. Также этот список можно получить, запустив в командной строке системы mysqlshow.

SHOW TABLES отображает список всех таблиц в заданной базе данных. Также можно в командной строке набрать mysqlshow имя_таблицы. В этих списках не будут указаны таблицы, на которые у вас нет никаких привилегий.

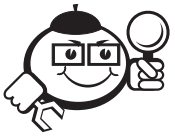
SHOW OPEN TABLES возвращает список таблиц, которые в данный момент открыты в табличном кеше. Поле comment указывает, сколько раз таблица кешировалась и использовалась.

SHOW COLUMNS демонстрирует список полей в таблице. Если задан параметр FULL, то также будет доступна информация о привилегиях пользователя для каждого поля. Типы полей могут отличаться от тех, что были заданы при создании или изменении таблицы.

SHOW FIELDS — синоним команды SHOW COLUMNS, а SHOW KEYS — синоним SHOW INDEX. Список полей и индексов можно также получить из командной строки, запустив утилиту mysqlshow db_name tbl_name или mysqlshow –– k db_name tbl_name.

SHOW INDEX возвращает информацию о ключах (индексах) таблицы в формате, который близок к ответу команды SQLStatistics в ODBC. В ответе будут такие колонки:

- SHOW TABLE STATUS Доступен, начиная с MySQL 3.23. Работает так же, как и SHOW STATUS, но показывает много информации о каждой таблице. Также можно использовать системную команду mysqlshow –status db_name.
- SHOW STATUS Команда сообщает о состоянии сервера (как и mysqladmin extended-status).
- SHOW VARIABLES Команда показывает значение некоторых системных переменных MySQL. Ту же информацию можно получить утилитой mysqladmin variables. Если значение некоторых из них вас не устраивает, можно их изменить из командной строки при старте mysqld.
- SHOW LOGS Информация о существующих файлах протоколов.
- SHOW PROCESSLIST Показывает активные потоки. Также информацию можно получить, запустив утилиту mysqladmin processlist. Если у вас есть привилегии process, то вы увидете все потоки; в противном случае вы увидете только свои потоки. При отсутствии параметра FULL вы увидете только первые 100 символов запроса.
- SHOW GRANTS Показывает список разрешенных пользователю команд.
- SHOW CREATE TABLE Показывает команду SQL, выполнение которой создаст данную таблицу.



Пример

```
SHOW DATABASES;
SHOW TABLES FROM u4eba;
```

Лабораторная работа №4

Цель работы

Целью работы является разработка схемы базы данных выбранной предметной области, её построение и разработка СУБД для базы данных, построенной в ходе предыдущей лабораторной работы.

Краткое изложение теоретической части

Имеются несколько основных компонент (объектов), которые используются для доступа к БД (рис. 5.4). Эти объекты могут быть разделены на три группы:

- невизуальные: *TTable, TQuery, TDataSet, TField* (для MySQL соответственно — *TSQLQuery, TSQLTable* и т. д.);
- визуальные: *TDBGrid, TDBEdit* и т. д.;
- связующие: *TDataSource*.

Первая группа включает невизуальные классы, которые используются для управления таблицами и запросами. Эта группа сосредотачивается вокруг компонент типа *TTable, TQuery, TDataSet* и *TField*. В Палитре Компонент эти объекты расположены на странице *Data Access*, для MySQL, соответственно на вкладке dbExpress.

Вторая важная группа классов — визуальные, которые показывают данные пользователю и позволяют ему просматривать и модифицировать их. Эта группа классов включает компоненты типа *TDBGrid, TDBEdit, TDBImage* и *TDBComboBox*. В Палитре Компонент эти объекты расположены на странице *Data Controls*.

Имеется и третий тип, который используется для того, чтобы связать предыдущие два типа объектов. К третьему типу относится невизуальный компонент *TDataSource* (рис. 5.4).

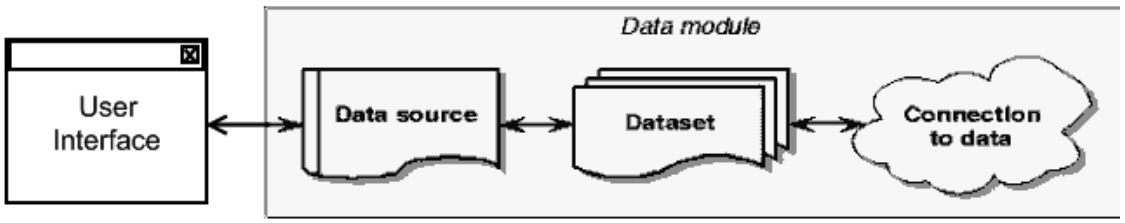


Рис. 5.4 – Схема взаимодействия компонент по доступу к БД

Компонент *TTable* можно поместить на форму в режиме проектирования или создать динамически во время выполнения. Перед вызовом метода *CreateTable()* необходимо установить значения свойств:

- *TableType* — тип таблицы,
- *DatabaseName* — база данных,
- *TableName* — имя таблицы,
- *FieldDefs* — массив описаний полей,
- *IndexDefs* — массив описаний индексов.

Свойство *TableType* имеет тип *TTableType* и определяет тип таблицы в базе данных. Если это свойство установлено в *ttDefault*, тип таблицы определяется по расширению файла.

Свойство *DatabaseName* определяет базу данных, в которой находится таблица. Это свойство может содержать:

- BDE алиас (в случае, если используется BDE),
- директорий для локальных БД,
- директорий и имя файла базы данных,
- локальный алиас, определенный через компонент *TDatabase*.

При использовании *TTable* возможен доступ ко всему набору записей из одной таблицы. В отличие от *TTable, TQuery* позволяет произвольным образом (в рамках SQL) выбрать набор данных для работы с ним. Во многом методика работы с объектом *TQuery* похожа на методику работы с *TTable*, однако есть свои особенности.

SQL запрос создается в ходе использования компонента *TQuery* следующим способом:

- Назначается псевдоним (Alias) *DatabaseName*.
- Используется свойство SQL компонента *TTable* чтобы ввести SQL запрос типа

Пример

«`Select * from Country`».

- Устанавливается свойство *Active* компонента *TTable* в *True*.

Если обращение идет к локальным данным, то вместо псевдонима можно указать полный путь к каталогу, где находятся таблицы.

Свойство SQL имеет тип *TStrings*, который означает, что это ряд строк, сохраняемых в списке. Список действует так же, как и массив, но, фактически, это специальный класс с собственными уникальными возможностями. В следующих нескольких абзацах будут рассмотрены наиболее часто используемые свойства.

При программном использовании *TQuery* рекомендуется сначала закрыть текущий запрос и очистить список строк в свойстве SQL:

Пример

```
Query1.Close
Query1.SQL.Clear
```

Метод *Add* используется для добавления одной или нескольких строк к запросу SQL. Общий объем ограничен только количеством памяти на вашей машине.

Чтобы Delphi отработал запрос и возвратил курсор, содержащий результат в виде таблицы, можно вызвать метод `Open`:

Пример

```
Query1.Open.
```

Пример выполнения задания

Рассмотрим случай, когда СУБД нужна для управления базой данных MySQL, а разработка ведётся в среде CodeGear Delphi 2009.

Создаем новое VCL приложение. Свойство *Name* формы меняем на *MainForm*, после чего сохраняемся в новой чистой папке. Модуль *.pas сохраняем под именем *Main*. Проект — под именем *WorkSUBD*.

Выбираем пункт меню <*Project/Options*>, заходим на вкладку *Application*. В окно редактирования *Title* вводим название своей предметной области (например, «СУБД учёбы в вузе») — таким образом зададим заголовок сообщений. Опционально, в качестве иконки на этой же вкладке выбираем любую необходимую иконку.

Со вкладки *dbExpress* берём компоненту *TSQLConnection*, меняем свойство *params* на параметры сервера БД (рис. 5.5):

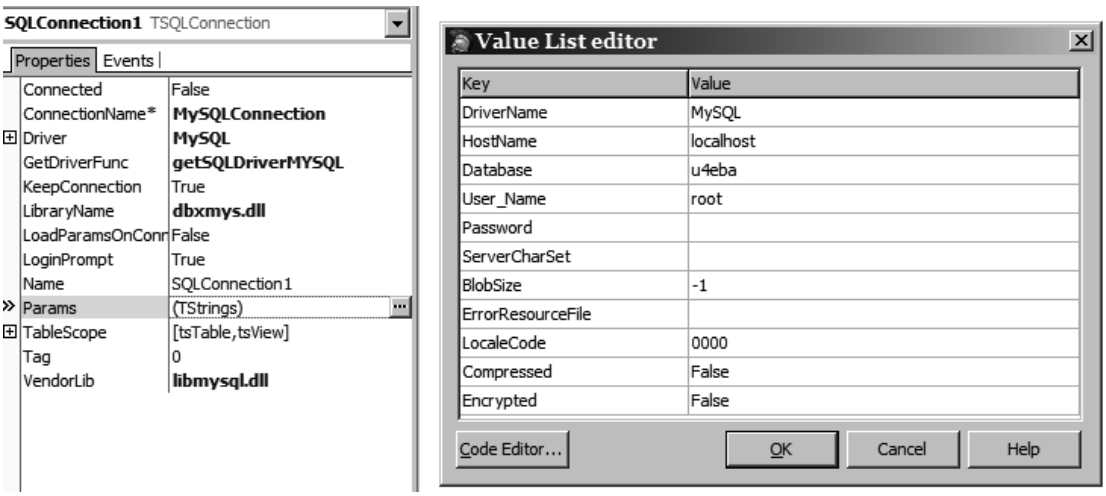


Рис. 5.5 – Параметры для доступа к серверу БД

Подключение к БД возможно проводить, изменив свойство *Connected* компоненты на *True*.

Создаем вторую форму (вызов из меню через <*File/New/Form*>), называем ее `EditForm`. В свойство Caption пишем — «`Редактор`». Сохраним под именем `UEdit`.

Кидаем 5 компонентов *TEdit* (вкладка *Standart*) и 2 кнопки — *TBitButton* (вкладка *Additional*) (рис. 5.6):

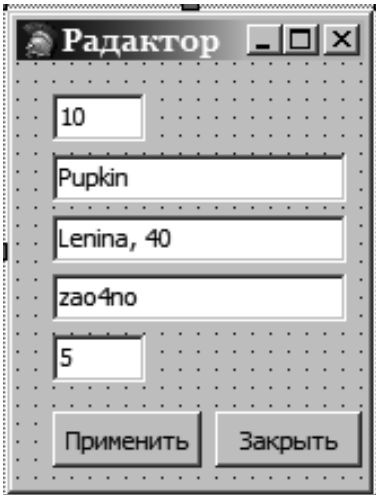
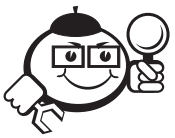


Рис. 5.6 – Внешний вид формы *EditForm*

Для кнопки «Применить» задаем свойство *ModalResult*=mrOK, для кнопки «Отменить» — *ModalResult*=mrCancel.

Обработчик для «Отменить»:


Пример

```
procedure TEditForm.Button2Click(Sender: TObject);
begin
   EditForm.Close;
end;
```

Создадим ещё ряд вспомогательных форм: *RateEditForm* (модуль *EditRate*), *EditFormPredmet* (модуль *EditPredet*), *FormSelect* (модуль *Select*). Расположим элементы управления на данных формах (рис. 5.7–5.10) и зададим обработчики событий.

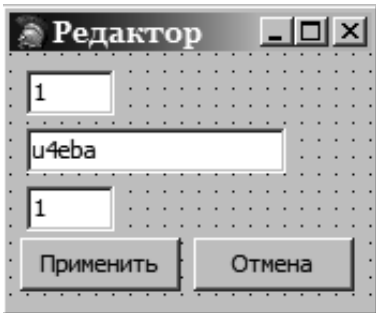


Рис. 5.7 – Внешний вид формы *EditFormPredmet*

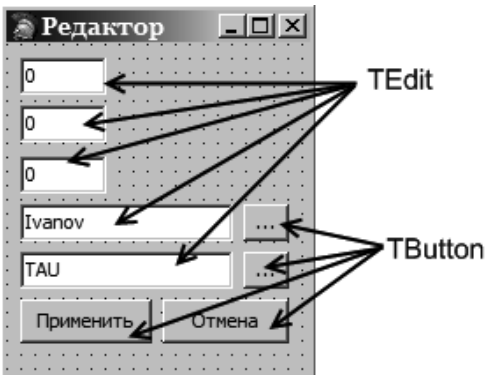


Рис. 5.8 – Внешний вид формы *RateEditForm*

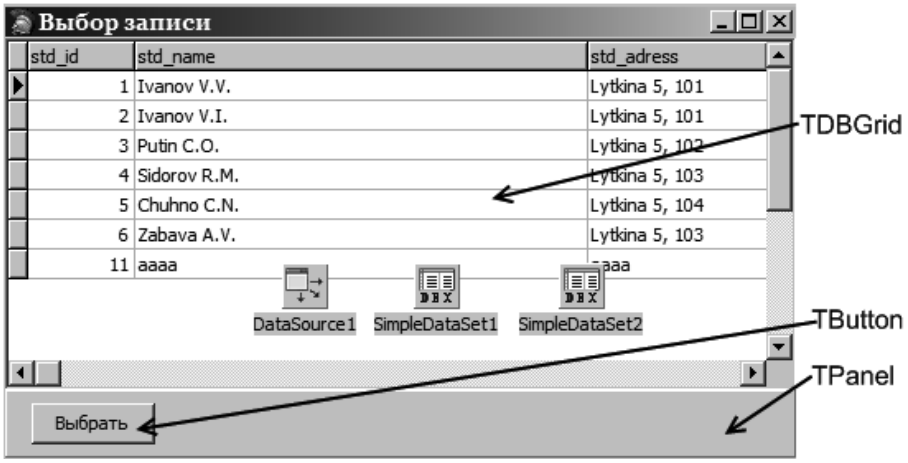


Рис. 5.9 – Внешний вид формы *FormSelect*

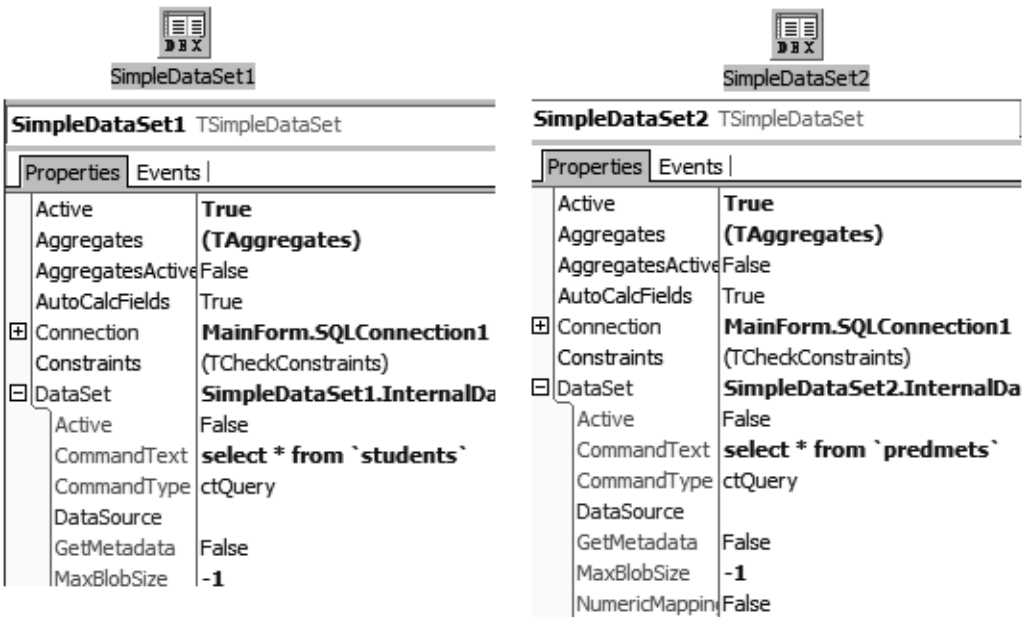


Рис. 5.10 – Настройка *SimpleDataSet1* и *SimpleDataSet2* формы *FormSelect*

Обработчики модуля *EditRate.*

Пример

```
procedure TRateEditForm.Button1Click(Sender: TObject);
begin
   //нажатие кнопки «...» рядом с полем предмета
   FormSelect:=TFormSelect.Create(self);
   FormSelect.DataSource1.DataSet:=FormSelect.SimpleDataSet2;
   FormSelect.ShowModal;
end;
procedure TRateEditForm.Button2Click(Sender: TObject);
begin
   //нажатие кнопки «...» рядом с полем имя студента
   FormSelect:=TFormSelect.Create(self);
   FormSelect.DataSource1.DataSet:=FormSelect.SimpleDataSet1;
   FormSelect.ShowModal;
end;
```

Обработчики модуля *Select.*

Пример

```
procedure TFormSelect.Button1Click(Sender: TObject);
begin
   if DataSource1.DataSet=SimpleDataSet2 then begin
        RateEditForm.Edit4.Text:=SimpleDataSet2pred_name.Value;
        RateEditForm.Edit4.Tag:=SimpleDataSet2pred_id.Value;
        MainForm.SQLQuery5.Params.ParamByName('pred').Value:=
SimpleDataSet2pred_id.Value;
   end;
   if DataSource1.DataSet=SimpleDataSet1 then begin
        RateEditForm.Edit5.Text:=SimpleDataSet1std_name.Value;
        RateEditForm.Edit5.Tag:=SimpleDataSet1std_id.Value;
        MainForm.SQLQuery5.Params.ParamByName('std').Value:=
SimpleDataSet1std_id.Value;
   end;
   FormSelect.Close;
end;
```

Также через меню <*File/USE Unit*> зададим для каждой из форм возможность использовать компоненты другой:

- *Main: uses UEdit, EditPredet, EditRate;*
- *Select: uses Main, EditRate;*
- *EditRate: uses Select.*

Поскольку управлять созданием форм мы будем вручную, отключим автоматическое управление через свойство меню <*Project/Options/Forms*> рис. 5.11.

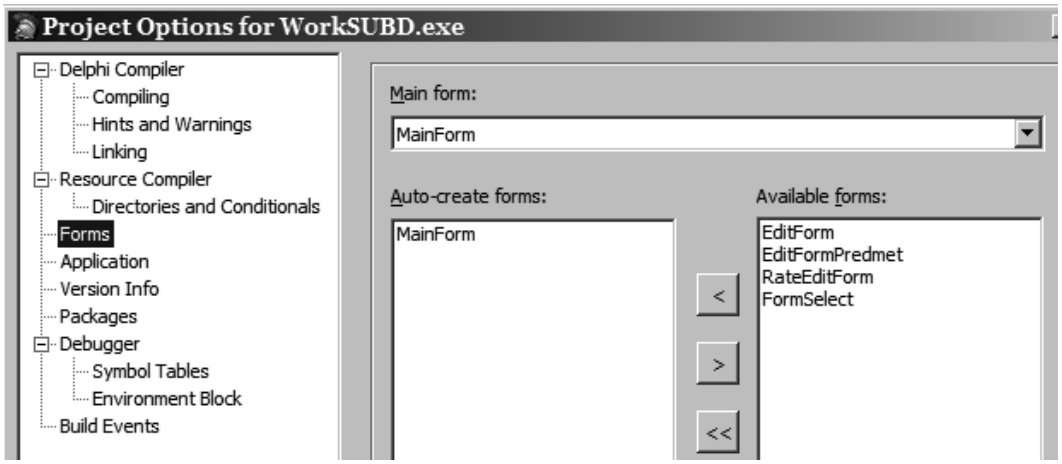


Рис. 5.11 – Отключение автоматического создания ненужных форм

Графическая часть приложения будет решена на основе современных представлений о дизайне приложений (на авторский взгляд, неудобных — но активно продвигаемых). Для этого расположим новый компонент *TRibbon* со вкладки *RibbonControl*. Здесь остановимся поподробнее.

На первый взгляд, разработчики пожадничали и оставили в распоряжение разработчика всего пять элементов:

1) *TRibbon* — основа всего интерфейса (лента);
2) *TRibbonComboBox*;
3) *TRibbonSpinEdit*;
4) *TRibbonScreenTipManager* — менеджер всплывающих подсказок;
5) *TScreenTipsPopup* — всплывающая подсказка.

Но на самом деле, этих компонентов более чем достаточно для разработки полноценного интерфейса. Более того, не факт, что вам когда-либо придётся использовать в работе второй и третий компоненты (*TRibbonComboBox* и *TRibbonSpinEdit*).

Дело в том, что Ribbon в Delphi работает только совместно с компонентом *TActionManager* со страницы *Additional* Палитры Компонентов. И этот компонент берет на себя львиную долю всей работы, в том числе и работы с событиями элементов.

Добавим на форму один *TActionManager* и один *TImageList*, после чего добавляем новую RibbonGroup, нажав правой кнопкой на нашем *Ribbon* и выбрав пункт *AddGroup*.

Через контекстное меню добавим к *ActionManager* новое действие (имя действия будет одновременно и названием кнопки в *Ribbon*, а свойство *ImageIndex*

будет определять индекс картинки из связанного компонента *ImageList*) со следующим обработчиком:

Пример

```
procedure TMainForm.Action1Execute(Sender: TObject);
begin
   SimpleDataSet1.Active:=false;
   DataSource1.DataSet:=SimpleDataSet1;
   SimpleDataSet1.Active:=true;
end;
```

После этого перетаскиваем из открытого *ActionManager* созданное действие на выбранную *RibbonGroup* (рис. 5.12).

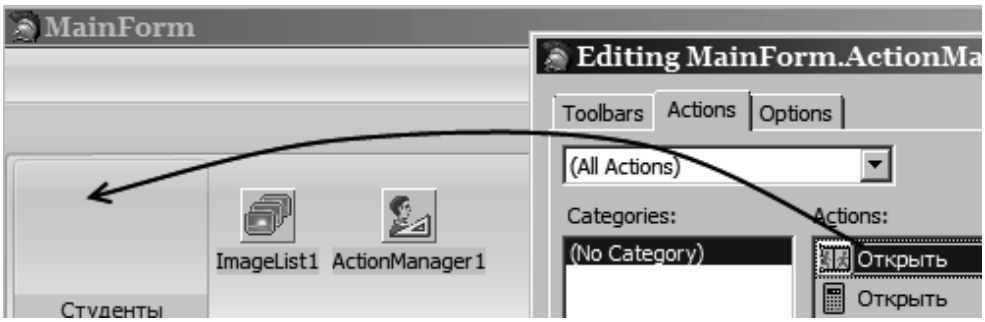


Рис. 5.12 – Добавление кнопки на TRibbon

Аналогичный подход можно использовать для всех используемых таблиц.

Разберёмся в механизме, с помощью которого данные из БД получают своё визуальное отображение в компонентах VCL.

Для отображения данных из таблиц необходимо через *TSQLConnection* установить связь с БД, затем определить свойство *CommandText* компонента *TSimpleDataSet*, описав в нём запрос, а затем связать его с компонентом *TDBGrid*. В упрощённом виде схема приведена на рис. 5.13.

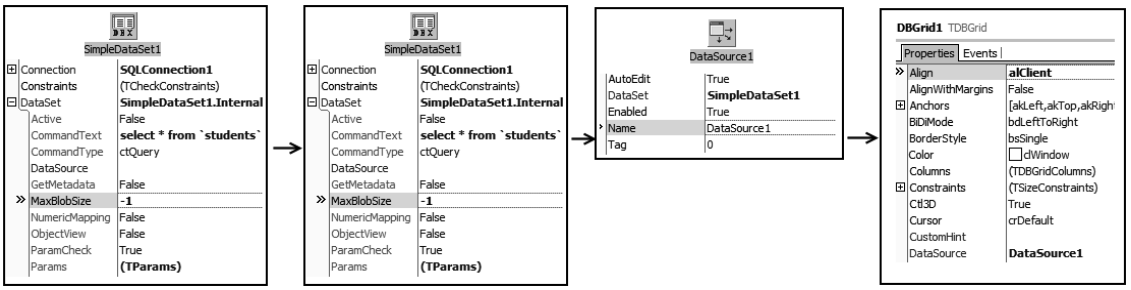


Рис. 5.13 – Отображение на TDBGrid таблицы students

Теоретически, в поле *Command Text* компонента *TSimpleDataSet* можно записать любой запрос. Например, для правильного отображения таблицы rate применяется сложный запрос (рис. 5.14):

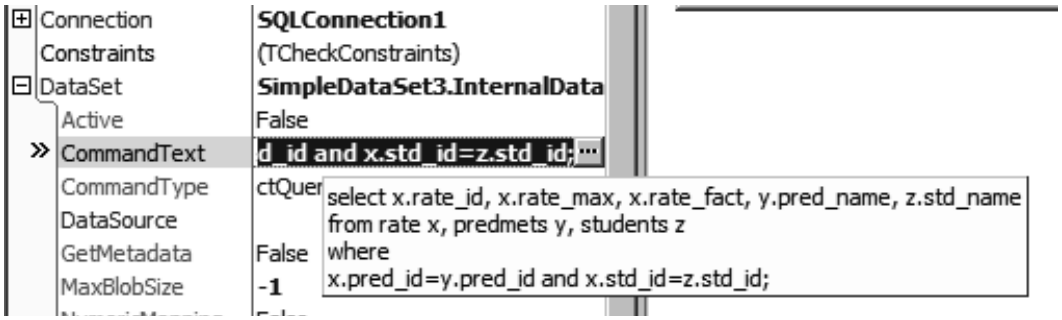


Рис. 5.14 – Запрос для отражения таблицы rate

В случае, когда необходимо ВЫПОЛНИТЬ специальный запрос, прежде чем отобразить данные на *DBGrid*, дополнительно используется компонент *TSQLQuery* (через определение свойства *SQLConnection*), а тот, в свою очередь, связывается через компонент *TSimpleDataSet*. В *TSQLQuery* задаётся запрос к БД. Так, в случае, если необходимо удалить запись из таблицы students, а затем отобразить результат в DBGrid, свойство SQL будет заполнено следующим образом: см. рис. 5.15, а схема связи приобретёт вид рис. 5.16.

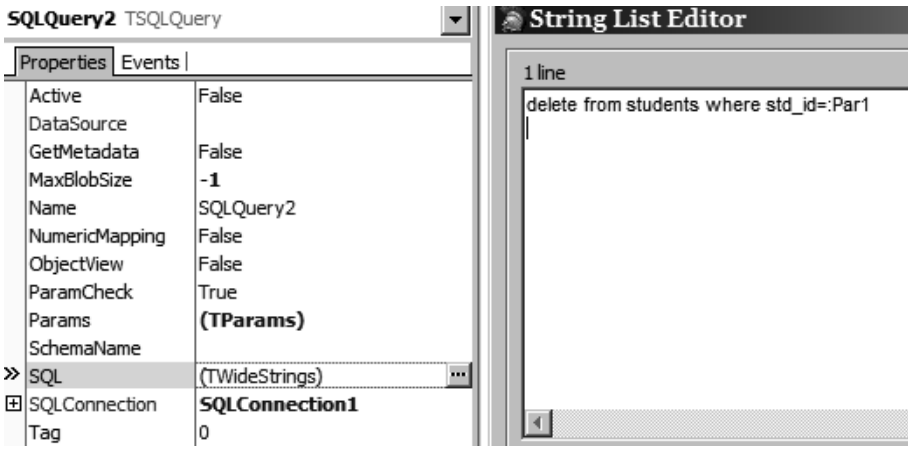


Рис. 5.15 – Заполненное свойство SQL

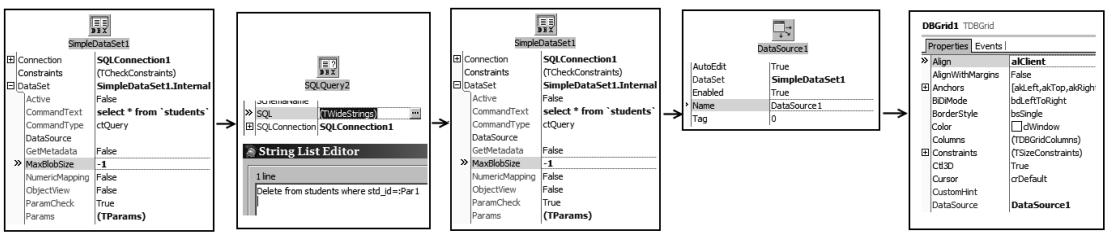


Рис. 5.16 – Отображение содержимого таблицы после изменений в таблице

Разберёмся с элементами *TSimpleDataSet*. Щелкнем два раза по пиктограмме *SimpleDataSet1*. Щелкаем правой кнопкой мыши по появившемуся окошку и выбираем пункт <*Add all fields*> (рис. 5.17).

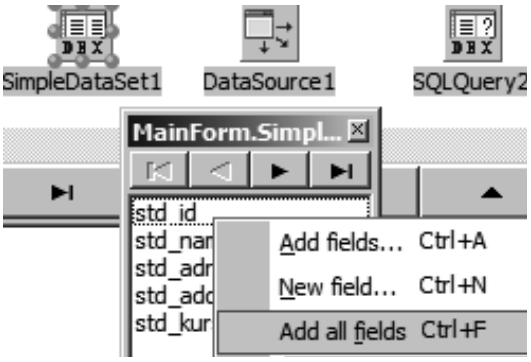


Рис. 5.17 – Меню добавления нового поля

Выбираем *std_id* и в инспекторе объектов, свойство *Visible* ставим в `False`. Теперь это поле не будет отображаться на экране в сетке `DBGrid` во время работы с программой.

Для *std_name* свойство *DisplayLabel* пишем «`Имя студента`».

Для *std_adress* в свойство *DisplayLabel* пишем «`Адрес`».

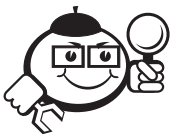
Для *std_addition* в свойство *DisplayLabel* пишем «`Примечание`».

Для *std_kurs* в свойство *DisplayLabel* пишем «`Курс`».

Таким образом определяется название и прочие свойства колонок, отображаемые после получения выборки из БД в графически связанном компоненте.

Разместим несколько *TSQLQuery* на форме, а поле *SQL* для данных компонент изменим на:

SQLQuery1.SQL:


Пример

```
insert into students
(std_id,std_name,std_adress,std_addition,std_kurs) values
(:id,:name,:adress,:add,:kurs)
```

SQLQuery2.SQL:


Пример

```
delete from students where std_id=:Par1
```

SQLQuery3.SQL:


```
insert into predmets (pred_id,pred_name, pred_kurs)
values (:id,:name,:kurs)
```

SQLQuery4.SQL:


```
delete from predmets where pred_id=:Par1
```

SQLQuery5.SQL:


```
insert into rate
(rate_id,rate_max,rate_fact,std_id,pred_id) values
(:id,:max,:fac,:std,:pred)
```

SQLQuery6.SQL:


```
d delete from rate where rate_id=:par1
```

Щелкаем по свойству *Params* компонента *SQLQuery1*, появляется окно редактирования свойств внешних параметров (параметры определяются при вводе запроса, через префикс «двоеточие» — рис. 5.18).

Данные параметры передаются в запрос при явном вызове через метод *ExecSQL()*. Параметры можно менять в процессе исполнения программы — в коде либо с помощью визуальных компонент. Для каждого из параметров необходимо определить тип данных в свойстве *DataType*.

Расположим на форме компонент *TDBNavigator* и выставим значение свойства *DataSource* на *DataSource1* — таким образом, навигатор станет контролировать *DBGrid*. На данном этапе главная форма приложения должна иметь вид, как на рис. 5.19.

Рис. 5.18 – Окно настроек параметров запроса

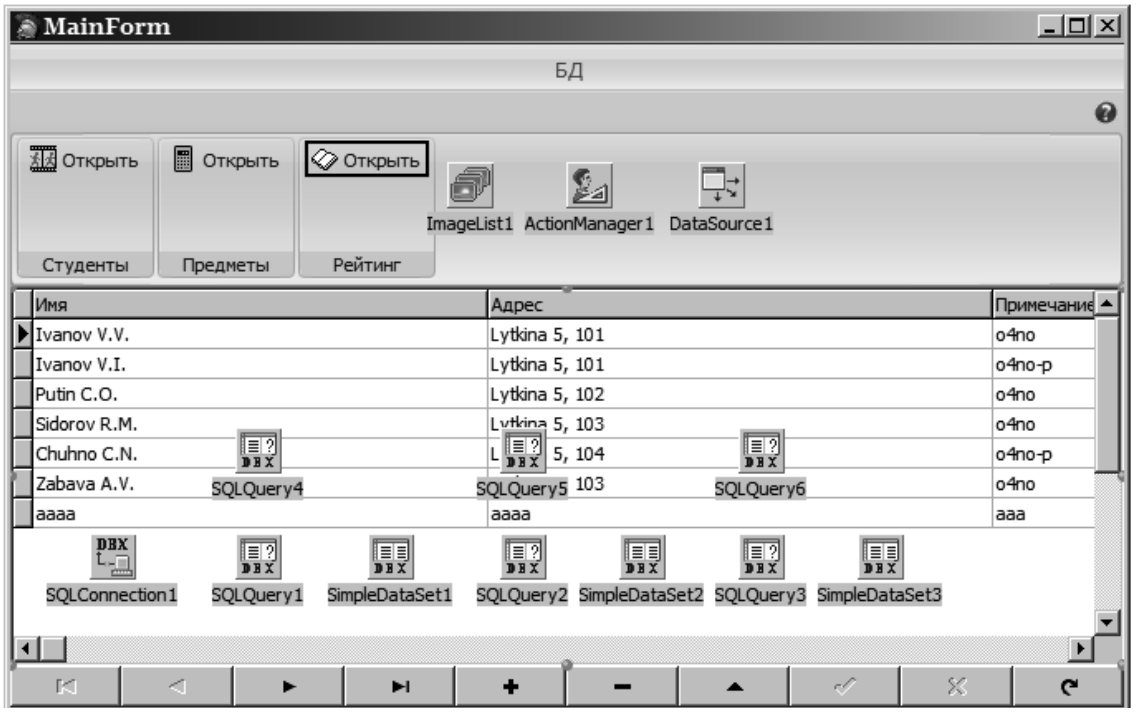


Рис. 5.19 – Внешний вид формы *MainForm*

Опишем обработчик для событий *OnCreate* главной формы, *OnClick* и *BeforeAction* компонента *DBNavigator1*.


```
procedure TMainForm.FormCreate(Sender: TObject);
begin
//Отключаем стандартный диалог на подтверждение удаления
      DBNavigator1.ConfirmDelete:=false;
end;

procedure TMainForm.DBNavigator1BeforeAction(Sender: TObject;
      Button: TNavigateBtn);
```

```
Begin
//Фокус с элемента при удалении теряется ДО самого удаления
      case Button of
//поэтому заранее запоминаем, на какой строке был фокус
      nbDelete: begin
            if DataSource1.DataSet=SimpleDataSet1 then
globid:=SimpleDataSet1std_id.Value;
            if DataSource1.DataSet=SimpleDataSet2 then
globid:=SimpleDataSet2pred_id.Value;
            if DataSource1.DataSet=SimpleDataSet3 then
globid:=SimpleDataSet3rate_id.Value;
              end;
      end;
end;

procedure TMainForm.DBNavigator1Click(Sender: TObject;
Button: TNavigateBtn);
begin
case Button of
//добавление элементов
      nbInsert: begin
//таблица students
       if DataSource1.DataSet=SimpleDataSet1 then begin
//создаем вспомогательнео окно
         EditForm:=TEditForm.Create(self);
//открываем его модально и ждём нажатия OK
         if EditForm.ShowModal=mrOk then begin
//после этого присваиваем параметрам запроса значения Edit-ов
           SqlQuery1.Params.ParamByName('id').Value:=
             StrToInt(EditForm.Edit1.Text);
           SqlQuery1.Params.ParamByName('name').Value:=
             EditForm.Edit2.Text;
           SqlQuery1.Params.ParamByName('adress').Value:=
             EditForm.Edit3.Text;
           SqlQuery1.Params.ParamByName('add').Value:=
             EditForm.Edit4.Text;
           SqlQuery1.Params.ParamByName('kurs').Value:=
             StrToInt(EditForm.Edit5.Text);
//пытаемся выполнить запрос
           try
              SqlQuery1.ExecSQL();
           except
              ShowMessage('students: Ошибка при добавлении
данных');
           end;
         end;
//перерисовка таблицы dbgrid
```

```
          SimpleDataSet1.Active:=false;
          SimpleDataSet1.Active:=true;
//освбождаем память
          EditForm.Free;
        end;
//таблица predmets
        if DataSource1.DataSet=SimpleDataSet2 then begin
          EditFormPredmet:=TEditFormPredmet.Create(self);
          if EditFormPredmet.ShowModal=mrOk then begin
            SqlQuery3.Params.ParamByName('id').Value:=
              StrToInt(EditFormPredmet.Edit1.Text);
            SqlQuery3.Params.ParamByName('name').Value:=
              EditFormPredmet.Edit2.Text;
            SqlQuery3.Params.ParamByName('kurs').Value:=
              StrToInt(EditFormPredmet.Edit3.Text);
            try
              SqlQuery3.ExecSQL();
            except
              ShowMessage('predmets: Ошибка при добавлении
данных');
            end;
          end;
          SimpleDataSet2.Active:=false;
          SimpleDataSet2.Active:=true;
          EditFormPredmet.Free;
        end;
//таблица rate
        if DataSource1.DataSet=SimpleDataSet3 then begin
          RateEditForm:=TRateEditForm.Create(self);
          if RateEditForm.ShowModal=mrOk then begin
            SqlQuery5.Params.ParamByName('id').Value:=
              StrToInt(RateEditForm.Edit1.Text);
            SqlQuery5.Params.ParamByName('max').Value:=
              StrToInt(RateEditForm.Edit2.Text);
            SqlQuery5.Params.ParamByName('fac').Value:=
              StrToInt(RateEditForm.Edit3.Text);
            try
              SqlQuery5.ExecSQL();
            except
              ShowMessage('rate: Ошибка при добавлении
данных');
            end;
          end;
          SimpleDataSet3.Active:=false;
          SimpleDataSet3.Active:=true;
          RateEditForm.Free;
```

```
        end;
      end; //nbInsert
//удаление элементов
      nbDelete: begin
//таблица students
        if DataSource1.DataSet=SimpleDataSet1 then begin
//выводим запрос на удаление
          if MessageDlg('Вы уверены, что хотите удалить
 запись?',
          mtConfirmation, [mbYes,mbNo] ,1)=mrYes then begin
//получаем ID текущей строки
            SqlQuery2.Params.ParamByName('Par1').Value:=globid;
//пытаемся выполнить запрос
            try
              SqlQuery2.ExecSQL();
            except
              ShowMessage('students: Ошибка при удалении
данных');
            end;
          end;
//перерисовка DBGrid
          SimpleDataSet1.Active:=false;
          SimpleDataSet1.Active:=true;
        end;
//таблица predmets

        if DataSource1.DataSet=SimpleDataSet2 then begin
          if MessageDlg('Вы уверены, что хотите удалить
запись?',
          mtConfirmation, [mbYes,mbNo] ,1)=mrYes then begin
            SqlQuery4.Params.ParamByName('Par1').Value:=globid;
            try
              SqlQuery4.ExecSQL();
            except
              ShowMessage('predmets: Ошибка при удалении
данных');
            end;
          end;
          SimpleDataSet2.Active:=false;
          SimpleDataSet2.Active:=true;
        end;
//таблица rate
        if DataSource1.DataSet=SimpleDataSet3 then begin
          if MessageDlg('Вы уверены, что хотите удалить
запись?',
          mtConfirmation, [mbYes,mbNo] ,1)=mrYes then begin
            SqlQuery6.Params.ParamByName('Par1').Value:=globid;
```

```
            try
              SqlQuery6.ExecSQL();
            except
              ShowMessage('rate: Ошибка при удалении
данных');
            end;
          end;
          SimpleDataSet3.Active:=false;
          SimpleDataSet3.Active:=true;
        end;
      end; //nbDelete
      nbRefresh: begin
        if DataSource1.DataSet=SimpleDataSet1 then begin
          SimpleDataSet1.Active:=false;
          SimpleDataSet1.Active:=true;
        end;
        if DataSource1.DataSet=SimpleDataSet2 then begin
          SimpleDataSet2.Active:=false;
          SimpleDataSet2.Active:=true;
        end;
        if DataSource1.DataSet=SimpleDataSet3 then begin
          SimpleDataSet3.Active:=false;
          SimpleDataSet3.Active:=true;
        end;
      end;//nbRefresh
end; //case Button
end;
```

Пример создания таблицы

В результате проделанных действий получается простейшая СУБД, в которой реализованы как отображение данных и склейка таблиц, так и простейшие механизмы по навигации, добавлению и удалению записей.

Рассмотрим в качестве примера запрос, результатом выполнения которого будет создание таблицы *Студент*.

```
CREATE TABLE `students` ()
   `std_id` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
   `std_name` VARCHAR(45) NOT NULL,
   `std_adress` VARCHAR(100),
   `std_addition` VARCHAR(100),
   `std_kurs` CHAR(5) NOT NULL,
   PRIMARY KEY (`std_id`)
) CHARACTER SET cp1251 COLLATE cp1251_general_ci;
```

`CREATE TABLE` — команда создания, в кавычках название таблицы, дальше в скобках указываются параметры таблицы, потом скобка закрывается, устанавливается кодировка данных, в конце запроса обязательно ставится точка с запятой.

`NOT NULL` — означает что поле не может содержать пустых значений — если пользователь попытается сохранить значение для NULL, то произойдет ошибка на уровне СУБД, а обрабатывать вам эту ошибку в своей программе или нет, это уже ваше дело.

`AUTO_INCREMENT` — означает, что при добавлении новых записей значение данного поля будет автоматически увеличиваться на единицу.

Поля `std_name, std_adress` и `std_addition` имеют текстовое значение, в SQL оно обозначается как `VARCHAR`, и в скобках указывается максимальное количество возможных символов.

Поле `std_id` назначается в качестве первичного ключа таблицы.

Задание

Разработать СУБД для базы данных, описанной в виде полноатрибутной модели. Программные средства для проектирования СУБД подобрать самостоятельно. Варианты языка разработки СУБД: PHP, Object Pacsal, C++, Python и т. д. Сложность — от двух таблиц готовой БД.

Контрольные вопросы по главе 5

1) В чём отличия запроса TRUNCATE от DELETE?
2) Дайте определение третьей нормальной форме.
3) Соответствуют ли возможности установить неидентифицирующую связь сущности АВТОМОБИЛЬ и ПАСПОРТ ТРАНСПОРТНОГО СРЕДСТВА?
4) Можно ли с помощью запроса CREATE DATABASE удалить базу данных?
5) Что означает понятие минимальности в отношении определения ключевых атрибутов?

ЗАКЛЮЧЕНИЕ

Не искать никакой науки кроме той, какую можно найти в себе самом или в громадной книге света.

Декарт

В заключение, хочется отметить, что, несмотря на обширность данного курса и стремление автора охватить как можно более широкие пласты знаний о функциональном моделировании, полное постижение всех нюансов построения эффективной и понятной бизнес-модели возможно лишь при многолетней практике в построении данных моделей.

Приведенные в пособие примеры никоим образом не следует воспринимать как абсолютное руководство к действию, как эталон функционального моделирования. Категорически недопустимо в ходе собственных попыток документирования бизнес-процессов придерживаться только данного пособия. Однако необходимо строго следовать следующим правилам:

- Законченность.
- Одноуровневость.
- Полнота.
- Непротиворечивость.
- Самодостаточность.
- Безызбыточная сложность.
- Соответствие нотации UML.

Унифицированный язык моделирования постоянно развивается, дополняется новыми графическими примитивами, расширяет горизонты, доступные для описания с помощью специализированных программных сред. Для каждого, кто ставит своей целью успешное освоение UML, совершенно необходимым является самостоятельное изучение последних изменений в нотации языка на сайте http://www.uml.org.

Те навыки, которые, как мы надеемся, приобрел читатель данного курса, необходимо каждодневно дополнять опытом, приобретаемым в работе с командой, над реальными проектами. Только в обмене опытом возможно становится описывать модели таким образом, чтобы они были понятными другим инженерам с первого взгляда.

Программирование на языках высокого уровня не может быть «высеченным в камне» — для каждого разработчика делом чести является постижение новых языков программирования, освоение новых программных средств функционального моделирования. За последние 70 лет программирование изменилось до неузнаваемости — на смену феноменальной памяти программистов 50-х годов, которые по памяти воспроизводили сложнейшие последовательности двоичных кодов, пришли не менее талантливые программисты, которые, не обладая столь мощным инструментарием, который доступен современному разработчику, — объектно-ориентированным подходом, смогли тем не менее запустить в космос человека, достичь Луны, научиться моделировать ядерные взрывы без вреда для окружающей среды. Несомненно, современный разработчик избалован обилием мощных и доступных для освоения программных сред — Visual Studio, Embarcadero Studio, Eclipse, их разнообразие не знает предела, но вместе с тем создает у разработчика представление о том, что он-де «всё знает». Однако всего постичь невозможно — и только постоянное самосовершенствование позволяет приближать человека к достижению идеала. Следуйте верному пути в дороге бесконечного поиска истины, и ваши модели будут лучшими!

ЛИТЕРАТУРА

Рекомендуемая литература

Настоящее методическое пособие содержит достаточную информацию для выполнения лабораторных работ. Для более подробного рассмотрения излагаемых вопросов, предлагается обратиться к литературе, список которой приводится ниже.

[1] Буч Г. Язык UML. Руководство пользователя / Г. Буч. — М. : ДМК Пресс, 2007. — 496 с.

[2] Леоненков А. В. Самоучитель UML / А. В. Леоненков. — 2-е изд. — СПБ. : БХВ-Петербург, 2006. — 427 с.

[3] Орлов С. А. Технологии разработки программного обеспечения. Разработка сложных программных систем : учеб. пособие / С. А. Орлов. — СПб. : Питер, 2002. — 464 с.

[4] Силич М. П. Системная технология: объектно-ориентированный подход : монография / М. П. Силич; Томский государственный университет систем управления и радиоэлектроники. — Томск : ТУСУР, 2002. — 224 с.

[5] Силич М. П. Системотехника / М. П. Силич, Е. Н. Рыбалка. — Томск : ТУСУР, 2006. — 152 с.

[6] Уилсон А. Информация, вычислительные машины и проектирование систем : пер. с англ. / А. Уилсон, М. Уилсон. — М. : Мир, 1968. — 415 с.

Электронные ресурсы

[1] Википедия [Электронный ресурс] : портал. — Режим доступа: http://ru.wikipedia.org/wiki/Нормальная_форма, свободный. — Загл. с экрана (дата обращения: 01.11.2012)

[2] Википедия [Электронный ресурс] : портал. — Режим доступа: http://ru.wikipedia.org/wiki/Первая_нормальная_форма, свободный. — Загл. с экрана (дата обращения: 01.11.2012)

[3] Википедия [Электронный ресурс] : портал. — Режим доступа: http://ru.wikipedia.org/wiki/Вторая_нормальная_форма, свободный. — Загл. с экрана (дата обращения: 01.11.2012)

[4] Википедия [Электронный ресурс] : портал. — Режим доступа: http://ru.wikipedia.org/wiki/Третья_нормальная_форма, свободный. — Загл. с экрана (дата обращения: 01.11.2012)

[5] Википедия [Электронный ресурс] : портал. — Режим доступа: http://ru.wikipedia.org/wiki/Четвертая_нормальная_форма, свободный. — Загл. с экрана (дата обращения: 01.11.2012)

[6] Википедия [Электронный ресурс] : портал. — Режим доступа: http://ru.wikipedia.org/wiki/Пятая_нормальная_форма, свободный. — Загл. с экрана (дата обращения: 01.11.2012)

[7] Википедия [Электронный ресурс] : портал. — Режим доступа: http://ru.wikipedia.org/wiki/Шестая_нормальная_форма, свободный. — Загл. с экрана (дата обращения: 01.11.2012)

[8] Википедия [Электронный ресурс] : портал. — Режим доступа: http://ru. wikipedia.org/wiki/Нормальная_форма_Бойса_—_Кодда, свободный. — Загл. с экрана (дата обращения: 01.11.2012)

[9] Википедия [Электронный ресурс] : портал. — Режим доступа: http://ru.wikipedia.org/wiki/Доменно-ключевая_нормальная_форма, свободный. — Загл. с экрана (дата обращения: 01.11.2012)

[10] Леоненков А. В. Нотации и семантика языка UML [Электронный ресурс] / А. В. Леоненков. — URL : http://www.intuit.ru/department/pl/umlbasics/1/, свободный. — Загл. с экрана (дата обращения: 01.11.2012).

[11] Буч Г. Язык UML. Руководство пользователя / Г. Буч, Дж. Рамбо, И. Якобсон. — М. : ДМК Пресс, 2006. — 496 с. — ISBN 5-94074-334-X, 0-321–-26797-4.

[12] Леоненков А. В. Самоучитель UML / А. В. Леоненков. — 2-е изд. — СПБ. : БХВ-Петербург, 2006. — 427 с.

Приложение А

СПИСОК ТЕМ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ

1. Кафе.
2. Железнодорожная касса.
3. Автосервис.
4. Больница.
5. Ветеринарная клиника.
6. Спартакиада.
7. Конкурс красоты.
8. Библиотека.
9. Авиакасса.
10. Магазин промышленных товаров.
11. Банк.
12. Прачечная.
13. Речной порт.
14. Гостиница.
15. Фотосалон.
16. Морг.
17. Политическая партия.
18. Отдел кадров.
19. Аэропорт.
20. Компьютерный магазин.

ГЛОССАРИЙ

Абстрактная операция, abstract operation. Объявленная, но не реализованная операция в абстрактном классе. В C++ абстрактные операции объявляются как чисто виртуальные функции-члены.

Абстрактный класс, abstract class. Класс, который не может иметь экземпляров (объектов). Абстрактный класс пишется в предположении, что его конкретные подклассы дополнят его структуру и поведение, скорее всего, реализовав абстрактные операции.

Абстракция, abstraction. Существенные характеристики объекта, которые отличают его от всех других объектов и четко определяют его концептуальные границы для наблюдателя. Абстрагирование — процесс выявления абстракций. Один из основных элементов объектной модели.

Агент, agent. Объект, который подвергается воздействию со стороны и сам воздействует на другие объекты. Обычно агенты создаются для выполнения некоторой работы по поручению актеров или других агентов.

Агрегация, aggregation. Специальная форма ассоциации, которая служит для представления отношения типа «часть-целое» между агрегатом (целое) и его составной частью.

Актер, actor. Объект, воздействующий на другие объекты, но сам не подвергающийся воздействию с их стороны. В некоторых контекстах то же самое, что активный объект.

Активный объект, active object. Объект, которому выделен свой поток управления.

Алгоритмическая декомпозиция, algorithmic decomposition. Процесс разделения системы на части, каждая из которых отражает этап общего процесса. Применение структурного подхода к проектированию приводит к алгоритмической декомпозиции, которая фокусируется на потоке управления в системе.

Архитектура модулей, module architecture. Граф, вершины которого соответствуют модулям, а ребра — отношениям модулей между собой. Архитектура модулей системы представляется совокупностью диаграмм модулей.

Архитектура процессов, process architecture. Граф, вершины которого соответствуют процессорам и устройствам, а ребра — соединениям между ними. Для описания архитектуры процессов системы используются диаграммы процессов.

Архитектура, architecture. Логическая и физическая структура системы, сформированная всеми стратегическими и тактическими проектными решениями.

Ассоциация, association. Отношение, означающее некоторую смысловую связь между классами.

Атрибут, attribute. Содержательная характеристика класса, описывающая множество значений, которые могут принимать отдельные объекты этого класса.

Базовый класс, base class. Наиболее общий класс в какой-либо структуре классов. В большинстве приложений есть несколько таких корневых классов. В некоторых языках программирования определяется всеобщий базовый класс, который является суперклассом для всех остальных классов.

Блокирующий объект, blocking object. Пассивный объект, способный работать в многопоточном окружении. Вызов операции блокирующего объекта блокирует клиента на все время операции.

Вариант использования, use case. Внешняя спецификация последовательности действий, которые система или другая сущность могут выполнять в процессе взаимодействия с актерами.

Видимость, visibility. Качественная характеристика описания элементов класса, характеризующая потенциальную возможность других объектов модели оказывать влияние на отдельные аспекты поведения данного класса (ссылаться на его ресурсы извне). Классы видимы друг другу, только если они находятся в одном пространстве имен. Контроль экспорта может еще более ограничить доступ к видимым классам.

Виртуальная функция, virtual function. Какая-либо операция над объектом. Виртуальная функция может быть переопределена в подклассах, следовательно, ее реализация определяется всем множеством методов, объявленных во всех классах дерева наследования. Термины «обобщенная функция» и «виртуальная функция» взаимозаменяемы.

Внешняя среда — для мехатронных машин это технологическая среда, которая содержит различное основное и вспомогательное оборудование, технологическую оснастку и объекты работ.

Временная сложность, time complexity. Относительное или абсолютное время, за которое выполняется операция.

Выражение действия, action expression. Представляет собой вызов операции или передачу сообщения, имеет атомарный характер и выполняется сразу после срабатывания соответствующего перехода до начала действий в целевом состоянии.

Действие, action. Некое происшествие в системе, требующее, с практической точки зрения, нулевого времени для своего завершения. Действием может быть вызов операции, запуск другого события, начало или остановка деятельности.

Действие, action. Спецификация выполнимого утверждения, которая образует абстракцию вычислительной процедуры.

Делегирование, delegation. При делегировании один объект, ответственный за операцию, передает выполнение этой операции другому объекту.

Деструктор, destructor. Операция класса, которая освобождает состояние объекта и/или уничтожает сам объект.

Деятельность, activity. Операция, выполнение которой требует некоторого времени.

Диаграмма, diagram. Графическое представление совокупности элементов модели в форме связного графа, вершинам и ребрам (дугам) которого приписывается определенная семантика.

Диаграмма взаимодействий, interaction diagram. Часть системы обозначений объектно-ориентированного проектирования; используется для демонстрации выполнения какого-либо сценария в контексте диаграммы объектов.

Диаграмма классов, class diagram. Диаграмма языка UML, на которой представлена совокупность декларативных или статических элементов модели, таких как классы с атрибутами и операциями, а также связывающие их отношения.

Диаграмма модулей, module diagram. Часть системы обозначений объектно-ориентированного проектирования; используется для демонстрации разбиения классов и объектов по модулям в физическом проекте системы. Диаграмма модулей отображает архитектуру модулей системы.

Диаграмма объектов, object diagram. Часть системы обозначений объектно-ориентированного проектирования; используется, чтобы наглядно показать объекты и отношения между ними в логическом проекте системы. Может отражать всю объектную структуру или часть ее; обычно иллюстрирует смысл механизмов в логическом проекте. Отдельная диаграмма объектов — моментальный снимок из жизни системы.

Диаграмма переходов и состояний, state transition diagram. Часть обозначений объектно-ориентированного проектирования; используется для отображения пространства состояний данного класса, событий, которые вызывают переход из одного состояния в другое, и действий, возникающих в результате смены состояния.

Диаграмма процессов, process diagram. Часть системы обозначений объектно-ориентированного проектирования; используется, чтобы наглядно показать, как процессы размещены по процессорам в физическом проекте системы. Диаграмма процессов отражает архитектуру процессов.

Диаграмма состояний, statechart diagram. Диаграмма, которая представляет конечный автомат.

Динамическое связывание, dynamic binding. Связывание означает установление соответствия имени (например, объявленной переменной) с классом. Динамическое связывание происходит при выполнении программы в тот момент, когда создается объект, обозначенный именем.

Домен, domen. Допустимое потенциальное ограниченное подмножество значений данного типа. Например, домен ИМЕНА определен на базовом типе строк символов, но в число его значений могут входить только те строки, которые могут представлять имена (в частности, для возможности представления русских имен такие строки не могут начинаться с мягкого или твердого знака и не могут быть длиннее, например, 20 символов). В один домен могут входить значения из нескольких колонок, объединённых, помимо одинакового типа данных, ещё и логически. Если два значения берутся из одного и того же домена, то можно выполнить сравнение этих двух значений. Более простое определение домена — это допустимое потенциальное множество значений одного типа.

Друг, friend. Класс или операция, имеющие доступ к закрытым операциям или данным некоторого класса. Только сам класс может называть своих друзей.

Закрытая часть, private. Часть интерфейса какого-либо класса, объекта или модуля, закрытая (невидимая) для других классов, объектов и модулей.

Защищенная часть, protected. Часть интерфейса какого-либо класса, объекта или модуля, невидимая для всех других классов, объектов и модулей за исключением подклассов.

Идентичность, identity. Природа объекта; то, что отличает его от других объектов.

Идиома, idiom. Выражение, общепринятое в каком-либо языке программирования или культуре какого-либо приложения, отражающее общепринятый способ использования данного языка.

Иерархия, hierarchy. Подчинение или упорядочение абстракций. Две типичных иерархии в сложной системе — структура классов (включая иерархию «общее/частное») и структура объектов (включая иерархию «целое/часть»); иерархии можно также обнаружить в архитектурах модулей и процессов.

Инвариант, invariant. Логическое выражение некоторого условия, истинность которого необходимо соблюдать.

Инкапсуляция, encapsulation. Процесс разделения элементов абстракции, которые образуют ее структуру и поведение. Служит для отделения внешних обязательств объекта от его реализации.

Инстанцирование, instantiation. Подстановка параметров шаблона обобщенного или параметризованного класса; в результате создается конкретный класс, который может иметь экземпляры.

Интерфейс, interface. Внешний вид класса, объекта или модуля, выделяющий его существенные черты и не показывающий внутреннего устройства и секретов поведения.

Исключение, exception. Возбуждение исключения показывает, что некоторый логический инвариант не соблюдается. В C++ мы возбуждаем исключение, чтобы избежать неправомочного исполнения операций и дать знать о возникшей проблеме другим объектам, которые могут перехватить исключение и принять меры.

Использовать, use. Ссылаться на абстракцию извне.

Итератор, iterator. Операция, позволяющая навещать части некоторого объекта.

Категория классов, class category. Логически полный набор классов, одни из которых видимы для других категорий классов, а другие — нет. Классы в категории сотрудничают для предоставления некоторого набора услуг.

Класс, class. Абстрактное описание множества однородных объектов с общей структурой и поведением. Термины «класс» и «тип» в большинстве случаев (но не всегда) взаимозаменяемы. Понятие класса отличается от понятия типа тем, что концентрируется на классификации по структуре и поведению.

Класс-контейнер, container class. Класс, экземпляры которого представляют собой коллекции других объектов. Контейнер может быть однородным (коллекции включают экземпляры только одного класса) либо неоднородным (коллекции вклю-

чают экземпляры разных классов, имеющих обычно общий суперкласс). В C++ контейнеры обычно определяются как параметризованные классы с параметром, обозначающим класс объектов коллекции.

Клиент, client. Объект, который пользуется услугами другого объекта, либо выполняя операции над последним, либо через доступ к его состоянию.

Ключ, key. Атрибут, значение которого однозначно идентифицирует объект.

Ключевая абстракция, key abstraction. Класс или объект, являющийся частью словаря предметной области.

Композит, composite. Класс, который связан отношением композиции с одним или большим числом классов.

Композиция, composition. Разновидность отношения агрегации, при которой составные части целого имеют такое же время жизни, что и само целое. Эти части уничтожаются вместе с уничтожением целого.

Конкретный класс, concrete class. Класс, реализация которого завершена и который, поэтому, может иметь экземпляры или объекты.

Конструктор, constructor. Операция, создающая объект и/или инициализирующая его состояние.

Кооперация, collaboration. Спецификация множества объектов отдельных классов, совместно взаимодействующих с целью реализации отдельных вариантов использования в общем контексте моделируемой системы.

Конечный автомат, state machine. Модель для спецификации поведения объекта в форме последовательности его состояний, которые описывают реакцию объекта на внешние события, выполнение объектом действий, а также изменение его отдельных свойств.

Конечное состояние, final state. Разновидность псевдосостояния, обозначающее прекращение процесса изменения состояний конечного автомата или нахождения моделируемого объекта в составном состоянии.

Линия жизни объекта, object lifeline. Вертикальная линия на диаграмме последовательности, которая представляет существование объекта в течение определенного периода времени.

Метакласс, metaclass. Класс класса; класс, экземпляры которого сами являются классами.

Метод, method. Операция над объектом, определенная как часть описания класса. Не любая операция является методом, но все методы — операции. Термины «метод», «сообщение» и «операция» обычно взаимозаменяемы. В некоторых языках методы существуют сами по себе и могут переопределяться подклассами; в других языках метод не может быть переопределен, — он служит как часть реализации обобщенных или виртуальных функций, которые можно переопределять в подклассах.

Механизм, mechanism. Структура, посредством которой объекты сотрудничают друг с другом, осуществляя поведение, которое соответствует требованиям системы.

Модификатор, modifier. Операция, изменяющая состояние объекта.

Модуль, module. Единица кода, служащая строительным блоком физической структуры системы; программный блок, который содержит объявления, выражен-

ные в соответствии с требованиями языка и образующие физическую реализацию части или всех классов и объектов логического проекта системы. Как правило, модуль состоит из интерфейсной части и реализации.

Модульность, modularity. Свойство системы, которая была разделена на связные и слабо зацепленные между собой модули.

Мономорфизм, monomorphism. Положение теории типов, согласно которому имена (например, переменных) могут обозначать только объекты одного и того же класса.

Мощность, cardinality. Число экземпляров класса: число экземпляров, участвующих в связи классов.

Наследование, inheritance. Отношение между классами, при котором класс использует структуру или поведение другого (одиночное наследование) или других (множественное наследование) классов. Наследование вводит иерархию «общее/частное», в которой подкласс наследует от одного или нескольких более общих суперклассов. Подклассы обычно дополняют или переопределяют унаследованную структуру и поведение.

Начальное состояние, start state. Разновидность псевдосостояния, обозначающего начало выполнения процесса изменения состояний конечного автомата или нахождения моделируемого объекта в составном состоянии.

n-арная ассоциация, n-ary association. Ассоциация между тремя и большим числом классов.

Обобщенная функция, generic function. Какая-либо операция над объектом. Обобщенная функция класса может быть переопределена в подклассах; следовательно, ее реализация определяется всем множеством методов, объявленных во всех классах дерева наследования. Термины «обобщенная функция» и «виртуальная функция» взаимозаменяемы.

Обобщенный класс, generic class. Класс, служащий шаблоном для создания других классов: шаблон параметризуется другими классами, объектами и/или операциями. Обобщенный класс до создания объектов должен быть инстанцирован. Обобщенные классы используются как контейнерные классы. Термины «обобщенный класс» и «параметризованный класс» взаимозаменяемы.

Обратный инжиниринг, reverse-engineering. Восстановление логической или физической модели системы по коду. Противопоставляется прямому инжинирингу.

Объект, object. Сущность с хорошо определенными границами и индивидуальностью, которая инкапсулирует состояние и поведение. Объект имеет состояние, поведение и идентичность. Структура и поведение сходных объектов определены в общем для них классе. Термины «экземпляр» и «объект» взаимозаменяемы.

Объектная модель, object model. Совокупность основополагающих принципов, лежащих в основе объектно-ориентированного проектирования; парадигма программирования, основанная на принципах абстрагирования, инкапсуляции, модульности, иерархичности, типизации, параллелизма и устойчивости.

Объектное программирование, object-based programming. Метод программирования, основанный на представлении программы как совокупности объектов, каждый из которых является экземпляром некоторого типа. Типы образуют иерар-

хию, но не наследственную. В таких программах типы рассматриваются как статические, а объекты имеют более динамическую природу, которую ограничивают статическое связывание и мономорфизм.

Объектно-ориентированная декомпозиция, object-oriented decomposition. Процесс разбиения системы на части, соответствующие классам и объектам предметной области. Практическое применение методов объектно-ориентированного проектирования приводит к объектно-ориентированной декомпозиции, при которой мы рассматриваем мир как совокупность объектов, согласованно действующих для обеспечения требуемого поведения.

Объектно-ориентированное программирование, object-oriented programming (OOP). Методология реализации, при которой программа организуется, как совокупность сотрудничающих объектов, каждый из которых является экземпляром какого-либо класса, а классы образуют иерархию наследования. При этом классы обычно статичны, а объекты очень динамичны, что поощряется динамическим связыванием и полиморфизмом.

Объектно-ориентированное проектирование, object-oriented design (OOD). Методология проектирования, соединяющая процесс объектно-ориентированной декомпозиции и систему обозначений для представления логической и физической, статической и динамической моделей проектируемой системы. Система обозначений состоит из диаграмм классов, объектов, модулей и процессов.

Объектно-ориентированный анализ, object-oriented analysis. Метод анализа, согласно которому требования рассматриваются с точки зрения классов и объектов, составляющих словарь предметной области.

Объект-член, member object. Часть состояния объекта. В совокупности объекты-члены полностью определяют структуру объекта. Термины «переменная экземпляра», «поле», «объект-член» и «слот» взаимозаменяемы.

Ограничение, constraint. Выражение некоторого смыслового условия, которое должно выполняться.

Операция класса, class operation. Операция, например, конструктор или деструктор, общая для всего класса и не принадлежащая конкретному объекту.

Операция, operation. Нечто, проделываемое одним объектом над другим, чтобы вызвать реакцию. Все операции, которые можно выполнить над каким-либо объектом, сосредоточены в свободных подпрограммах и функциях-членах (методах). Термины «операция», «метод» и «сообщение» взаимозаменяемы.

Ответственность, responsibility. Поведение, за которое ответственен объект.

Открытая часть, public. Часть интерфейса какого-либо класса, объекта или модуля, открытая (видимая) для всех классов, объектов и модулей.

Отношение, relationship. Семантическая связь между отдельными элементами модели.

Пакет, package. Общецелевой механизм для организации различных элементов модели в множество, реализующий системный принцип декомпозиции модели сложной системы и допускающий вложенность пакетов друг в друга.

Параллелизм, concurrency. Свойство, отличающее активные объекты от неактивных.

Параллельный объект, concurrent object. Активный объект, способный работать в многопоточной среде.

Параметр, parameter. Спецификация переменной операции, которая может быть изменена, передана или возвращена.

Параметризованный класс, parameterized class. Класс, служащий шаблоном для других классов; шаблон параметризуется другими классами, объектами и/или операциями. Параметризованный класс должен быть инстацирован до создания объектов. Параметризованные классы используются как контейнеры. Термины «обобщенный класс» и «параметризованный класс» взаимозаменяемы.

Пассивный объект, passive object. Объект, не имеющий собственного потока управления.

Переменная класса, class variable. Часть состояния класса. Совокупность всех переменных класса образует его структуру. Переменные класса совместно используются всеми его экземплярами. В C++ переменная класса объявляется как статический член.

Переменная экземпляра, instance variable. Часть состояния объекта. В совокупности переменные экземпляра полностью определяют структуру объекта. Термины «переменная экземпляра», «поле», «объект-член» и «слот» взаимозаменяемы.

Переход, transition. Отношение между двумя состояниями, которое указывает на то, что объект в первом состоянии должен выполнить определенные действия и перейти во второе состояние.

Поведение, behavior. Действия и реакции объекта, выраженные в терминах передачи сообщений и изменения состояния; видимая извне и воспроизводимая активность объекта.

Подкласс, subclass. Класс, наследующий от одного или нескольких классов (которые называются его непосредственными суперклассами).

Подпакет, subpackage. Пакет, который является составной частью другого пакета.

Подсистема, subsystem. Совокупность модулей, часть которых видима для других подсистем, а часть — скрыта.

Поле, field. Часть состояния объекта; совокупность полей объекта образуют его структуру. Термины «поле», «переменная экземпляра», «объект-член» и «слот» означают одно и то же.

Полиморфизм, polymorphism. Положение теории типов, согласно которому имена (например, переменных) могут обозначать объекты разных (но имеющих общего родителя) классов. Следовательно, любой объект, обозначаемый полиморфным именем, может по-своему реагировать на некий общий набор операций.

Последовательное проектирование, round-trip gestalt design. Стиль проектирования, который подчеркивает последовательность и итеративность в развитии системы: посредством уточнения различных, хотя и согласованных логических и физических представлений системы в целом; объектно-ориентированное проектирование основывается на последовательном проектировании, что является выражением взаимозависимости общей картины проекта и его деталей.

Последовательный объект, sequential object. Пассивный объект, рассчитанный на работу в однопоточном окружении.

Постусловие, postcondition. Инвариант, соблюдаемый на выходе из операции.

Поток управления, thread of control. Отдельный процесс. Запуск потока управления приводит к возникновению независимой динамической деятельности в системе; данная система может иметь несколько одновременно выполняемых потоков, некоторые из которых могут динамически возникать и уничтожаться. Многопроцессорные системы допускают истинную многопоточность, в то время как на однопроцессорных компьютерах возможна только иллюзия многопоточности. (Термин «thread of control» переводится также «нить управления». В некоторых случаях используется термин «flow of control», который также переведен.)

Предусловие, precondition. Инвариант, предполагаемый на входе в операцию.

Примесь, mixin. Класс, реализующий какое-либо четко выделенное поведение; используется для уточнения поведения других классов посредством наследования; поведение примеси обычно ортогонально поведению класса, с которым она смешивается.

Производный атрибут, derived element. Атрибут класса, значение которого для отдельных объектов может быть вычислено посредством значений других атрибутов этого же объекта.

Пространственная сложность, space complexity. Относительный или абсолютный объем памяти, занимаемый объектом.

Пространство состояний, state space. Перечислимое множество всех возможных состояний объекта. Пространство состояний программы содержит неопределенное, но конечное число состояний (не обязательно желаемых или ожидаемых).

Протокол, protocol. Способы, которыми объекты могут действовать и реагировать; полное статическое и динамическое представление объекта; протокол объекта определяет допустимое поведение объекта.

Процесс, process. Запуск одного потока управления.

Процессор, processor. Часть аппаратного обеспечения, имеющая вычислительные ресурсы.

Прямой инжиниринг, forward-engineering. Создание исполнимого кода по логической или физической модели. Противопоставляется обратному инжинирингу.

Псевдосостояние, pseudo-state. Вершина в конечном автомате, которая имеет форму состояния, но не обладает поведением.

Раздел, partition. Категории классов или подсистемы, составляющие часть данного уровня абстракции.

Реактивная система, reactive system. Система, движимая событиями. Поведение такой системы не определяется простым отображением «вход-выход».

Реализация, implementation. Внутреннее представление класса, объекта или модуля, включая секреты его поведения.

Роль, role. Способность или цель, с которой класс или объект участвует в отношениях с другими; некоторая четко выделяемая черта поведения объекта в определенный момент времени; роль — это лицо, которое объект являет миру в данный момент.

CRC-карточки, CRC cards. CRC — Class/Responsibilities/Collaborators, Класс/Ответственности/Сотрудники; простое, но достаточно эффективное средство мозгового штурма при выявлении ключевых абстракций и механизмов.

Свободная подпрограмма, free subprogram. Процедура или функция, которая выполняется как непримитивная операция над объектом или объектами одного и того же или различных классов. Свободная подпрограмма — это любая подпрограмма, которая не является методом какого-либо класса.

Связь, link. Связь между объектами, экземпляр ассоциации.

Селектор, selector. Операция, имеющая доступ к состоянию объекта, но не изменяющая его.

Сервер, server. Объект, который никогда не воздействует на другие объекты, но используется ими; объект, предоставляющий некоторые услуги.

Сигнатура, signature. Полная спецификация операции с указанием типов аргументов и возвращаемого значения.

Сильно типизированный, strongly typed. Свойство языка программирования, в соответствии с которым во всех выражениях гарантируется согласованность типов.

Синхронизация, synchronization. Семантика параллельности операции. Операция может быть простой (присутствует только один поток управления); синхронной (рандеву двух потоков); односторонней (рандеву, при котором одному из потоков приходится ждать); по истечении времени (рандеву, в котором один процесс ждет другого определенное время); асинхронной (два процесса независимы друг от друга).

Система реального времени, real-time system. Система, в которой некоторые существенные процессы должны укладываться в отведенное время. Система «жесткого» реального времени должна быть детерминированной; запаздывание с реакцией грозит катастрофой.

Скрытие информации, information hiding. Процесс скрытия всех секретов объекта, которые ничего не добавляют к его существенным характеристикам; обычно скрывают структуру объекта и реализацию его методов.

Срабатывание перехода, fire. Выполнение перехода из одного состояния в другое.

Словарь данных, data dictionary. Полный перечень всех классов в системе.

Слой, layer. Совокупность категорий классов или подсистем одного уровня абстракции.

Слот, slot. Часть состояния объекта; совокупность слотов образуют структуру объекта. Термины «поле», «переменная экземпляра», «объект-член» и «слот» означают одно и то же.

Событие, event. Что-то, что может изменить состояние системы.

Сообщение, message. Операция, которую один объект может выполнять над другим. Термины «сообщение», «метод» и «операция» обычно взаимозаменяемы.

Составной объект (агрегат), aggregate object. Объект, состоящий из других объектов (его частей).

Состояние, state. Совокупный результат поведения объекта: одно из стабильных условий, в которых объект может существовать, охарактеризованных количественно; в любой конкретный момент времени состояние объекта включает в себя

перечень (обычно, статический) свойств объекта и текущие значения (обычно, динамические) этих свойств.

Сотрудничество, collaboration. Процесс, в котором несколько объектов сотрудничают для обеспечения требуемого поведения верхнего уровня.

Сохраняемость, persistence. Способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из одного адресного пространства в другое.

Среда разработки, framework. Набор классов, предоставляющих некоторые базовые услуги в определенной области. Таким образом, среда разработки экспортирует классы и механизмы, которые клиенты могут использовать или адаптировать.

Статическое связывание, static binding. Связывание означает установление соответствия имени (например, объявленной переменной) классу. Статическое связывание происходит при объявлении имени (во время компиляции), до того, как объект будет создан.

Страж, guard. Логическое выражение, применяемое к событию; если выражение истинно, то событие происходит и система изменяет состояние.

Стратегическое проектное решение, strategic design decision. Проектные решения, которые имеют решающее влияние на архитектуру.

Структура классов, class structure. Граф, вершины которого соответствуют классам, а ребра — отношениям классов. Структура классов для конкретной системы представляется в виде совокупности диаграмм классов.

Структура объектов, object structure. Граф, вершины которого соответствуют объектам, а ребра — отношениям объектов. Для отражения структуры объектов или ее части используются диаграммы объектов.

Структура, structure. Конкретное представление состояния объекта. Каждый объект имеет собственное состояние, независимое от других объектов, хотя все объекты одного класса имеют одинаковое представление состояния.

Структурное проектирование, structured design. Метод проектирования, основанный на алгоритмической декомпозиции.

Суперкласс, superclass. Класс, которому наследуют другие классы (называемые непосредственными подклассами).

Сценарий, scenario. Последовательность событий, выражающая некий аспект поведения системы.

Тактическое проектное решение, tactical design decision. Проектное решение, имеющее ограниченное значение для архитектуры.

Тип, type. Определение области допустимых значений, которые может принимать объект, и множества операций, которые могут выполняться над объектом. Термины «класс» и «тип» обычно (но не всегда) взаимозаменяемы; тип отличается от класса тем, что фокусируется на поддержке общего протокола.

Типизация, typing. Механизмы, препятствующие замене объектов одного типа на другой или, в крайнем случае, жестко ограничивающие такую замену.

Трансформационная система, transformational system. Система, поведение которой определяется в терминах отображения «вход-выход».

Управление доступом, access control. Механизм доступа к данным и операциям класса. В C++ открытые элементы доступны всем, защищенные элементы доступны подклассам, так называемым друзьям класса и файлам реализации, закрытые элементы доступны реализации и друзьям класса. Наконец, элементы с доступом на уровне реализации доступны только в файле реализации класса.

Уровень абстракции, level of abstraction. Относительное упорядочение абстракций по структурам классов, объектов, модулей или процессов. В терминах иерархии «часть/целое» объект находится на более высоком уровне абстракции, чем другие, если он строится на основе этих объектов: в терминах иерархии «общее/частное» высокоуровневые абстракции носят более обобщенный характер, чем низкоуровневые.

Уровень представления, layer. Способ организации и рассмотрения модели на одном уровне абстракции, который представляет горизонтальный срез архитектуры модели, в то время как разбиение представляет ее вертикальный срез.

Услуга, service. Поведение, обеспечиваемое некоторой частью системы.

Устройство, device. Часть аппаратуры, не имеющая собственных вычислительных ресурсов.

Утверждение, assertion. Логическое выражение некоторого условия, истинность которого необходимо обеспечить.

Утилита класса, class utility. Совокупность свободных подпрограмм. На C++ — класс, который состоит только из статических членов и/или функций-членов.

Фокус управления, focus of control. Специальный символ на диаграмме последовательности, указывающий период времени, в течение которого объект выполняет некоторое действие, находясь в активном состоянии.

Функциональная точка, function point. В контексте анализа требований к системе — отдельное поведение, видимое извне и поддающееся проверке.

Функция, function. Некоторое преобразование «вход-выход», вытекающее из поведения объекта.

Функция-член, member function. Операция над объектом, определенная как часть описания класса. Все функции-члены — операции, но не все операции — функции-члены. Термины «функции-члены» и «методы» взаимозаменяемы. В некоторых языках функции-члены существуют сами по себе и могут переопределяться подклассами; в других языках функция-член не может быть переопределена, — она служит как часть реализации обобщенных или виртуальных функций, которые можно переопределять в подклассах.

Экземпляр, instance. Нечто, чем можно оперировать. Экземпляр имеет состояние, поведение и идентичность. Структура и поведение всех экземпляров класса определяются этим классом. Термины «объект» и «экземпляр» взаимозаменяемы.

Учебное издание

Изюмов Антон Алексеевич

СПЕЦКУРС. ТЕХНОЛОГИЯ РАЗРАБОТКИ
ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебно-методическое пособие

Корректор Осипова Е. А.
Компьютерная верстка Насынова Н. Е.

Подписано в печать 11.04.13. Формат 60x84/8.
Усл. печ. л. 20,46. Тираж 300 экз. Заказ

Издано в ООО «Эль Контент»
634029, г. Томск, ул. Кузнецова д. 11 оф. 17
Отпечатано в Томском государственном университете
систем управления и радиоэлектроники.
634050, г. Томск, пр. Ленина, 40
Тел. (3822) 533018.