Урок №3

Диаграмма классов, диаграмма состояний, диаграмма деятельности

Содержание

1. Диаграмма классов 6

1.1. Цели данного типа диаграмм. 6

1.2. Базовые понятия. 9

1.3. Отношения между классами 20

1.4. Сравнительный анализ отношений. 27

1.5. Практические примеры построения диаграмм классов 28

2. Диаграмма состояний. 31

2.1. Цели данного типа диаграмм. 31

2.2. Базовые понятия. 32

2.3. Составное состояние и подсостояние. 39

2.4. Историческое состояние. 41

2.5. Сложные переходы. 43

2.6. Практические примеры построения диаграмм состояний 46

3. Диаграмма деятельности 47
3.1. Цели данного типа диаграмм. 47
3.2. Базовые понятия. 48
3.3. Практические примеры построения диаграмм деятельности. 57

Вступление

Начать изложение хотелось бы с того, чтобы напомнить основные этапы разработки программного обеспечения. Так, в общем случае, разработка состоит из:

- анализа проблемы и планирование;
- моделирования;
- реализация, тестирование и документирование;
- внедрение и сопровождение.

Нас в на данном этапе особенно будут интересовать анализ проблемы, планирование и моделирование.

Разработка начинается с этапа анализа проблемы, поставленной перед группой разработчиков. На этом этапе формируется терминология процесса решения проблемы, а так же выделяются основные понятия, которыми будут оперировать процессы, протекающие в информационной системе, разрабатываемой в рамках решения поставленной проблемы.

Считаем важным отметить, что любое программное обеспечение – это информационная система, а, значит, прежде всего, – система; и, следовательно, рассматривать её необходимо в терминологии теории систем. Это упростит понимание излагаемого в уроке материала.

После того, как были выделены базовые понятия и процессы, начинается этап моделирования системы и её компонентов, которые в более детальном рассмотрении, как правило, тоже представляют собой системы.

Здесь, раз уж речь идёт о моделях, считаем необходимым напомнить, что модели по сложности делят на глобальные и базовые, а по степени детализации на обобщённые и детализированные.

Этап моделирования предполагает составление как обобщенных моделей глобальных систем и процессов, так и детализированных моделей базовых элементов и активностей (под активностью понимается выполнение системой некоторого алгоритма, которая обычно представляется в виде блок-схемы).

Моделирование важно с той точки зрения, что оно позволяет всем членам команды разработчиков одинаково представлять конечный результат, выраженный архитектором в виде UML-моделей, а анализ важен в том смысле, что позволяет выделить и сформировать понятия, которые разработчики будут использовать, описывая процедуры анализа и принятия решений в информационной системе.

Система понятий (типов), которыми оперирует информационная система, обычно выражается в виде структурной модели приложения, обычно представленной в виде совокупности диаграмм классов, которым и будет посвящена следующая глава.

1. Диаграмма классов

Диаграммы классов – это способ представления в UML внутренней структуры класса и интерфейса взаимодействия с ним. Напомним, что под структурой подразумеваются поля класса, то есть те данные, которые класс инкапсулирует, а под интерфейсом – все общедоступные (public) методы, которые определяют способы взаимодействия с объектами данного класса. Так же благодаря тому, что в диаграмме классов предусмотрена возможность спецификации доступности атрибута, существует возможность отобразить скрытые предобразования, определяемые классом – те методы, которые не доступны извне, но объясняют внутреннее поведение класса.

Диаграмма классов представляет собой статическое представление модели приложения с точки зрения архитектора (лица, которое отвечает за проектирование системы).

1.1. Цели данного типа диаграмм

Можно выделить следующие цели использования диаграмм классов:

- описание модели системы типов, используемой приложением. Иначе модель системы типов ещё называют словарём системы, поскольку, она составляет понятийную базу, используемую системой для описания процесса, который она поддерживает;
- отражение простых кооперaций, имеющих место в моделируемой системе. Под кооперациями пони-

мается совокупности классов, интерфейсов и т.д., функционирующих совместно для реализации некоторого кооперативного поведения.
Эту цель можно считать наиболее важной с точки зрения моделирования систем, поскольку она отражает эмерджентность – наиболее важное свойство, присущее системам. Эмерджентность – это общее свойство систем, которое говорит о том, что система наделена некоторыми свойствами, не сводимыми к свойствам составляющих её элементов.
Рассматривая отдельные классы, входящие в кооперацию, мы не видим той функции, которую выполняет кооперация. и только рассматривая кооперацию обобщенно, в её целостности, мы находим то свойство, которое она реализует и соответственно вносит в конечную систему.

- отражение структурного аспекта организации данных в приложении. Так же диаграмма классов может использоваться для описания логической схемы базы данных, используемой информационной системой.

Согласно Мартину Фаулеру существуют три основных точки зрения, с которых необходимо рассматривать любую модель, и в особенности диаграммы классов, то есть структурную модель системы:

- **концептуальная точка зрения**. С концептуальной точки зрения диаграммы классов служат для «представления понятий изучаемой предметной области». Однако, вследствие того, что, зачастую, реализация

достаточно сильно отличается от модели, конечные типы данных (классы) могут совершенно не соответствовать понятиям, которыми оперирует архитектор при проектировании системы. А значит, концептуальная модель может иметь слабое (или не иметь совсем) отношение к конечному программному обеспечению.

- **точка зрения спецификации**. С точки зрения спецификации значение имеет и интерфейс информационной системы и его реализация. Но объектно-ориентированное проектирование рассматривает только интерфейсы программной системы, в отрыве от реализации. В принципе, когда говорят о проектировании класса, в основном имеется в виду проектирование интерфейса класса, поскольку с точки зрения конечного использования важна не внутренняя структура объектов, а их поведение. Ведь, согласитесь, абсолютно не важными будут структурные отличия между двумя реализациями класса, если результат их поведения будет оставаться одинаковым.
- **точка зрения реализации**. С данной точки зрения мы проектируем собственно классами, а, соответственно, являются важными все свойства, присущие классам, как лексическим конструкциям некоторого языка программирования, поскольку мы «спускаемся» на «уровень реализации». Данная точка зрения наиболее приближена к реальному программному продукту. Однако, для проектирования точка зрения спецификации более адекватна и предпочтительна.

Описанные выше точки зрения можно условно сопоставить с разными этапами процесса разработки программного обеспечения и определить как уровни процесса разработки. Так, на этапе анализа и планирования мы рассматриваем информационную систему с концептуальной точки зрения и определяем этот этап разработки как концептуальный уровень; на этапе моделирования (проектирования) – с точки зрения спецификации и определяем его как уровень моделирования; а на этапе разработки – с точки зрения реализации и определяем его как уровень реализации.

1.2. Базовые понятия

1.2.1. Класс

Класс является базовым понятием объектно-ориентированного подхода в программировании, а значит, оно необходимо должно использоваться в объектно-ориентированном анализе и проектировании. Как уже понятно из сказанного выше, всякое понятие языка моделирования UML зависит от уровня, на котором оно используется. Таким образом, на концептуальном уровне класс представляет собой одно из понятий, используемых для описания проблемной области, в рамках которой решается поставленная перед проектировщиками задача. На уровне моделирования – представляет собой некоторый тип данных, а на уровне реализации – формальный тип данных, описанный на некотором объектно-ориентированном языке программирования.

Но для успешного оперирования понятием класса необходимо одновременно подразумевать под ним все три аспекта, в которых он может существовать – другими словами, воспринимать понятие класса в его целостности.

Атрибут

Упрощённо можно сказать, что атрибут – это некоторые данные, хранимые классом и некоторым образом его характеризующие. Например, возьмем такое понятие как должность. Необходимо, чтобы должность имела некоторое название и сопоставлялась с некоторой стартовой (минимальной зарплатой). Таким образом, у понятия должность будут атрибуты «имя», сообщающий нам название должности, и «минимальная заработная плата», сообщающий стартовый оклад рабочего, находящегося на этой должности.

На концептуальном уровне наличие атрибута «имя» у понятия «должность», говорит нам о том, что должность наделена некоторым именем. На уровне спецификации мы понимаем, что объект тип «должность» может сообщить нам и своё название, а так же, возможно, получить от нас новое название, если это предусмотрено концепцией. На уровне реализации мы понимаем, что класс Position будет иметь поле Name.

Синтаксически определение атрибута выглядит следующим образом:

```
видимость имя_атрибута: тип = значение_по_умолчанию
```

Синтаксис определения атрибута описан в общем виде. Поэтому ниже мы укажем как фактически описывается каждый элемент:

- **видимость** указывается при помощи знаков «+», «#» и «–», где «+» означает, что атрибут имеет спецификатор public, «#» – protected, а «–» – private;
- **имя_атрибута** указывается в виде последовательности символов;
- **тип** подразумевает спецификацию имени типа атрибута;
- в качестве значения по умолчанию используют указание фактического значения, которое будет устанавливаться атрибуту в качестве умалчиваемого. Как правило, значение по умолчанию указывают атрибутам базовых типов данных.

Ниже приведён пример описания атрибутов для класса «должность»

```
MinimalSalary: decimal
Name: string
```

А справа приведено графическое представление класса Position в диаграмме классов, или, другими словами, изображение, которое иллюстрирует, как будет выглядеть описание класса Position (должность) на языке UML в диаграмме классов.

Операция

Под операциями подразумеваются процессы, которые (реализует) выполняет некоторый класс. Как правило, операции, определяемые на концептуальном уровне, соответствуют методам класса на уровне спецификации.

Синтаксис определения операции на языке UML выглядит следующим образом:

```
видимость имя_операции(список_аргументов):
тип_возвращаемого_значения{свойства_операции}
```

- **видимость операций** определяется так же, как и для атрибутов;
- **имя_операции** определяется в виде последовательности символов;
- **список_аргументов** состоит из разделенных запятой параметров, описанных следующим образом:

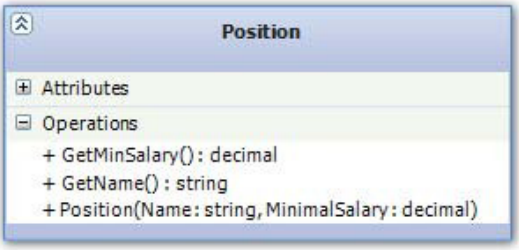
```
направление имя_параметра: тип = значение_по_умолчанию
```

где направление указывает, как аргумент используется операцией, и может принимать следующие значения: (in) – входящий аргумент, (out) – исходящий аргумент, то есть тот, который инициализируется в процессе операции, (inout) – аргумент используется операцией в обоих направлениях. Если **направление** не указано, то подразумевается значение (in), поэтому чаще всего оно опускается.

- **тип_возвращаемого_значения** представляет список из разделённых запятой типов данных. Но большинство разработчиков используют только один тип данных.
- **свойства_операции** представляют собой список из разделенных запятой свойств, применяемых к операции.

Ниже приведён пример описания операций для типа данных «должность»

```
+ GetMinSalary(): decimal
+ GetName(): string
+ Position(Name: string, MinSalary: decimal)
```



Рисунок, представленный сверху, иллюстрирует то, как будет выглядеть объявление операций для типа данных «должность» в реальной UML диаграмме.

1.2.2. Объект

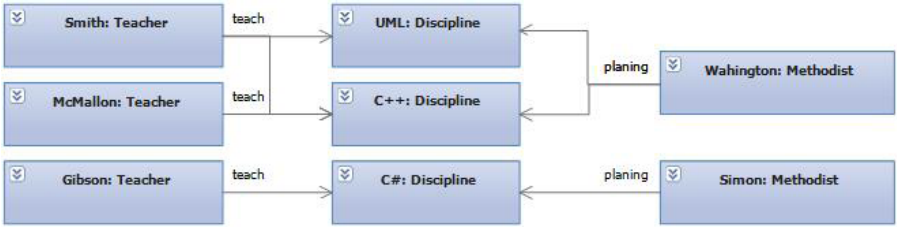
На концептуальном уровне и уровне моделирования объект понимается как экземпляр некоторого класса. Другими словами, если класс описывает некоторое понятие о некотором объекте предметной области, то понятие «объекта» представляет собой собственно сам этот объект, описываемый нами при помощи класса.

На уровне реализации под объектном понимают действующий (созданный в памяти) экземпляр некоторого класса.

В UML предусмотрен такой вид диаграмм, как диаграмма объектов (object diagram). Диаграмма объектов не является классической для UML, но в некоторых случаях используется для уточнения способа использования классов в тех или иных контекстах, а так же для представления функционального назначения класса в общей системе.

Ниже представлено диаграмма объектов, в которой фигурируют три класса Discipline, Teacher и Methodist. Диаграмма иллюстрирует, что один учитель может читать

несколько предметов и выполнять с предметом только одно действие, то есть преподавать его. В то время, как планированием предмета занимается методист. Иными словами на диаграмме концептуальное ограничение, вносимое проектировщиком, которое выражается в разделении обязанностей между преподавателем и методистом.



На диаграмме видно, что при писании объекта после имени через символ «:» указан тип объекта. Таким образом, визуально всегда можно отличить описание класса, от описания объекта.

```
имя_объекта: имя_шаблона <имя_параметра_шаблона = занчение>
```

Так же объекты могут вноситься в диаграммы классов, для того, чтобы описать константные объекты, используемые другими классами.

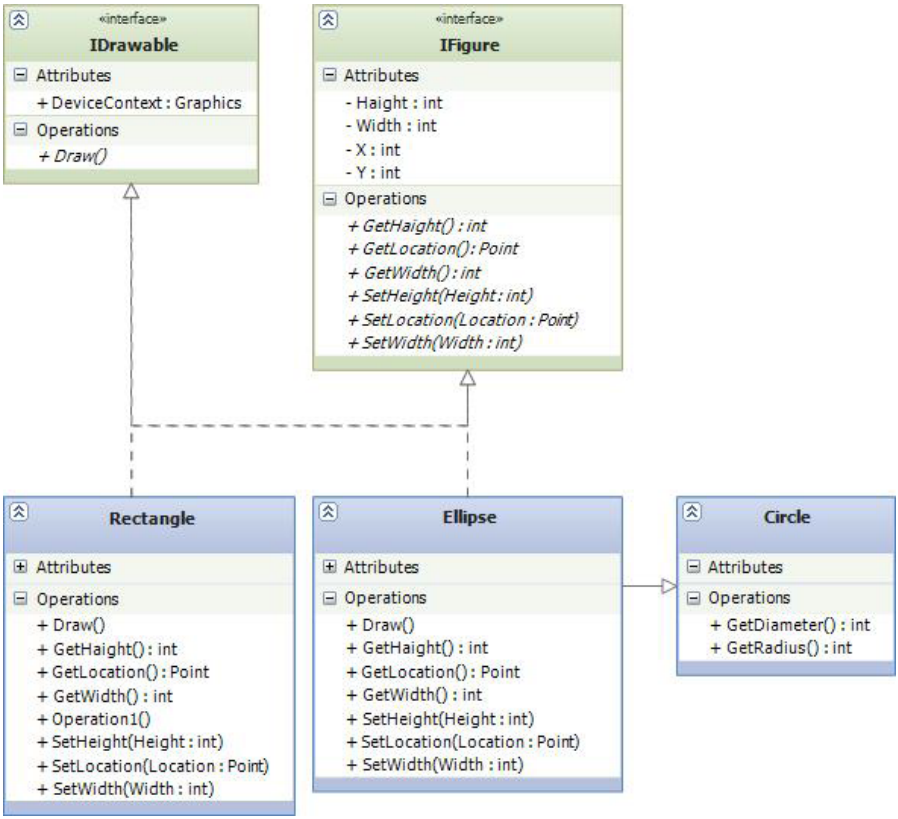
1.2.3. Интерфейс

С одной стороны под интерфейсом мы понимаем набор общедоступных (public) операций у класса, которые в их совокупности представляют различные способы использования класса.

С другой стороны интерфейс – это ещё одна категория языка UML, который содержит описание функциональности

некоторого понятия. Отсюда следует, что интерфейс – это именованный набор открытых свойств, выраженный в виде типа. Основная идея, определяющая существование интерфейсов – это внесение разделения между описанием и функциональности и реализацией.

Синтаксически описание интерфейса на UML практически не отличается от описания класса, за одним исключением: при описании интерфейса, при помощи ключевого слова «interface», указывается, что описывается интерфейс.



Важно помнить, что интерфейс содержит только описание возможностей, но не включает никакой реализации, а значит от него нельзя создавать объекты.

На представленном выше примере объявлены два интерфейса: IDrawable и IFigure, которые наследуются двумя классами: Rectangle и Ellipse. В такой ситуации принято говорить, что описанные классы находятся в отношении наследования (или отношении родства) с описанными интерфейсами. При этом функциональность, как правило, полностью описывается интерфейсами, а производные классы содержат только реализацию. Но такая ситуация не всегда возможна. Иногда бывает совершенно необходимо, чтобы класс, реализующий интерфейс, содержал дополнительную функциональность, специфичную только для этого класса. Например, методы необходимые для вычисления диаметра и радиуса круга нельзя поместить в интерфейс, поскольку они не нужны в классах прямоугольник и эллипс, и их наличие в этих классах не целесообразно. Исходя из этого, мы определяем круг как частный случай эллипса и добавляем описанному классу «круг» (Circle) необходимую функциональность, а основную функциональность косвенно наследуем от интерфейса IFigure через класс Ellipse.

На уровне реализации, в большинстве объектно-ориентированных языков программирования для описания интерфейсов существует специальная одноимённая языковая конструкция. В тех языках, в которых такой конструкции нет, для осуществления разделения функциональности и реализации используют абстрактные классы.

1.2.4. Шаблон

Понятие шаблона относиться к области обобщённого программирования. Существует возможность при описании класса, не указывать явно типы принимаемых или возвращаемых операциями значений. Вместо этого можно использовать так называемые «структурные нули» или «заполнители» (placeholders), которые могут быть заменены фактическим значением типа при создании нового класса. Такой подход реализует принцип так называемого «повторного использования кода», а класс, описанный подобным образом, будет называться «шаблонным классом» или просто «шаблоном».

Шаблон на UML описывается подобно классу, за тем исключением, что в правом верхнем углу в пунктирном прямоугольном контейнере, через запятую указываются параметры шаблона в следующем формате:

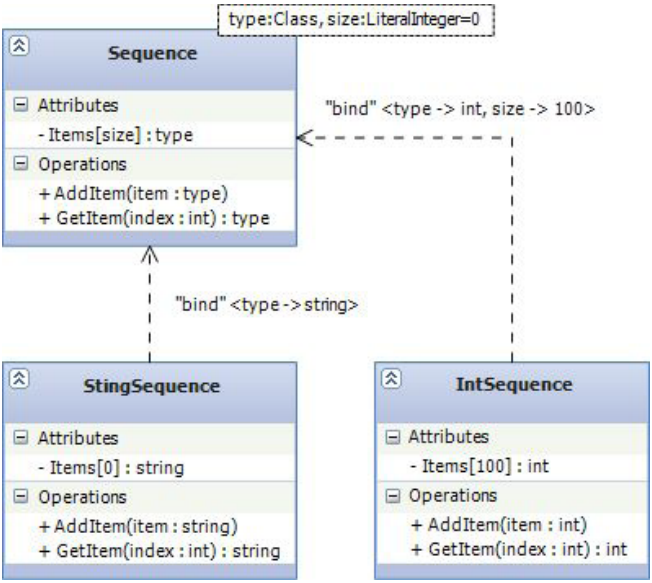
```
имя_параметра: тип_паарметра =
               значение_по_умолчанию
```

Далее приведён пример описания шаблонного класса и явного связывания его с реализующими его экземплярами (явное связывание обеспечивается путём указания отношения зависимости типа «связь» между классом и созданными от него объектами. Отношения будут рассмотрены ниже в соответствующем параграфе).

Для этого, как видно из иллюстрации, необходимо связать объекты с шаблоном при помощи пунктирной стрелки и указать с использованием стереотипа «bind» (напомним, что стереотипы расширяют словарь UML, позволяя создавать новые блоки из уже существующих)

явные значения параметров шаблона в следующем формате:

```
<имя_1_параметра -> значение, …, имя_n_параметра ->
значение >
```

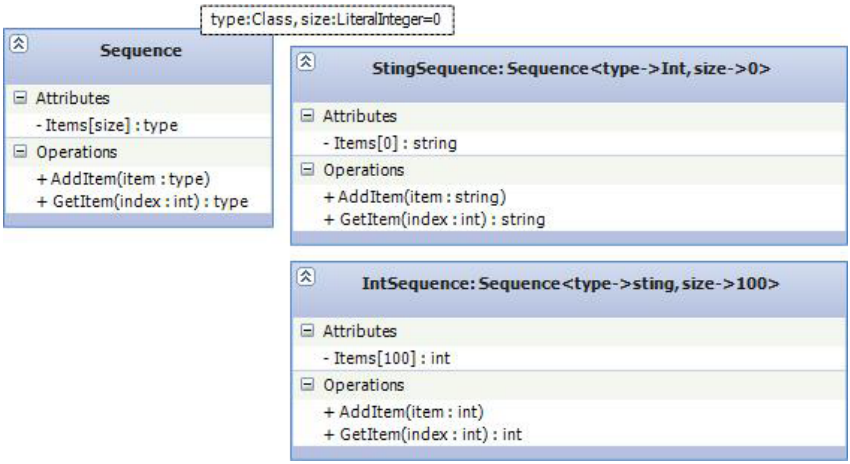


Чаще используют неявное связывание объекта, с реализуемым им шаблоном. Неявное связывание объекта с шаблоном во многом похоже на описание обычного объекта, но с перечислением параметров шаблона в следующей форме:

```
имя_объекта: имя_шаблона <имя_параметра_шаблона =
             занчение>
```

Значения параметров шаблона разделяются запятыми.

Ниже приведён предыдущий пример с тем исключением, что объекты связаны неявно:



1.3. Отношения между классами

Рассматривая вопрос моделирования необходимо рассмотреть понятие системы. Поскольку сам термин моделирования подразумевает, что проектировщик занимается созданием или воссозданием чего-то целого из некоторых частей. Обычно это целое называют системой, оговаривая необходимое условие наличия свойства эмерджентности системы, которое, собственно говоря, и является системовыделяющим свойством.

Можно упрощённо представить систему как совокупность элементов и характеры связей между ними. Под эмерджентностью, в свою очередь, понимают наличие у системы свойств, не сводимых к совокупности свойств составляющих её элементов.

Система имеет две важные характеристики: структуру и характер связей между элементами. Наличие нескольких элементов является аксиомой, очевидно вытекающей из понятия моделирования.

Под структурой обычно понимают совокупность составляющих систему элементов, вне зависимости от того, как именно они между собой связаны или, другими словами, в каких отношениях они между собой находятся. Тогда как характер связей между элементами системы значим в такой же мере, как и структура, поскольку именно он определяет характер процессов, протекающих в системе, а значит – влияет на свойства системы, как целого.

Далее будут рассмотрены различные виду отношений между классификаторами, предусмотренные синтаксисом языка моделирования UML.

1.3.1. Отношение зависимости (dependency relationship)

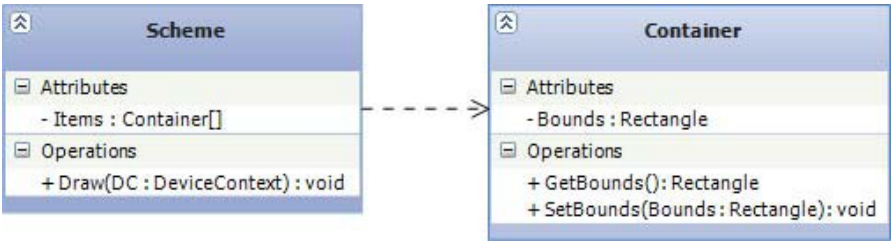
В UML, отношение зависимости – это отношение, при котором один элемент, обычно называемый «клиентом» (client), использует другой элемент, называемый «поставщиком» (supplier), или зависит от него.

Отношение зависимости может быть использовано в диаграммах: классов, компонентов, развёртывания, и вариантов использования, чтобы указать, что изменения в «поставщике» могут требовать соответствующих изменений в «клиенте».

Также можно использовать отношение зависимости, чтобы указать приоритет элементов.

Обычно, отношения зависимости не имеют имён или подписей. Однако их можно использовать, для того,

чтобы определить характер зависимости, существующей между элементами модели.



Синтаксисом UML предусмотрены следующие виды (характеры) отношений зависимости:

- абстракция (ключевые слова, которые могут быть использованы для выражения данного типа зависимости: abstraction, derive, refine, trace) – определяет отношение между двумя или несколькими элементами модели, определяющими (выражающими, описывающими) одно и то же понятие на разных уровнях абстракции;
- связь (ключевое слово – bind) – связывает аргументы шаблона с параметрами шаблона в объектах.
- реализация (ключевое слово – realize) – указывает, что элемент-клиент – это реализация элемента-поставщика.
- замещение (ключевое слово – substitute) – указывает, что элемент-клиент занимает место элемента-поставщика. При использовании такого характера зависимости, интерфейс клиента обязательно должен соответствовать интерфейсу поставщика.
- использование (ключевые слова: use, call, create, instantiate, send) – указывает, что один элемент

требует другой элемент для реализации той или иной функции.

Пример зависимости типа «связь» был показан выше в параграфе, посвящённом шаблонам.

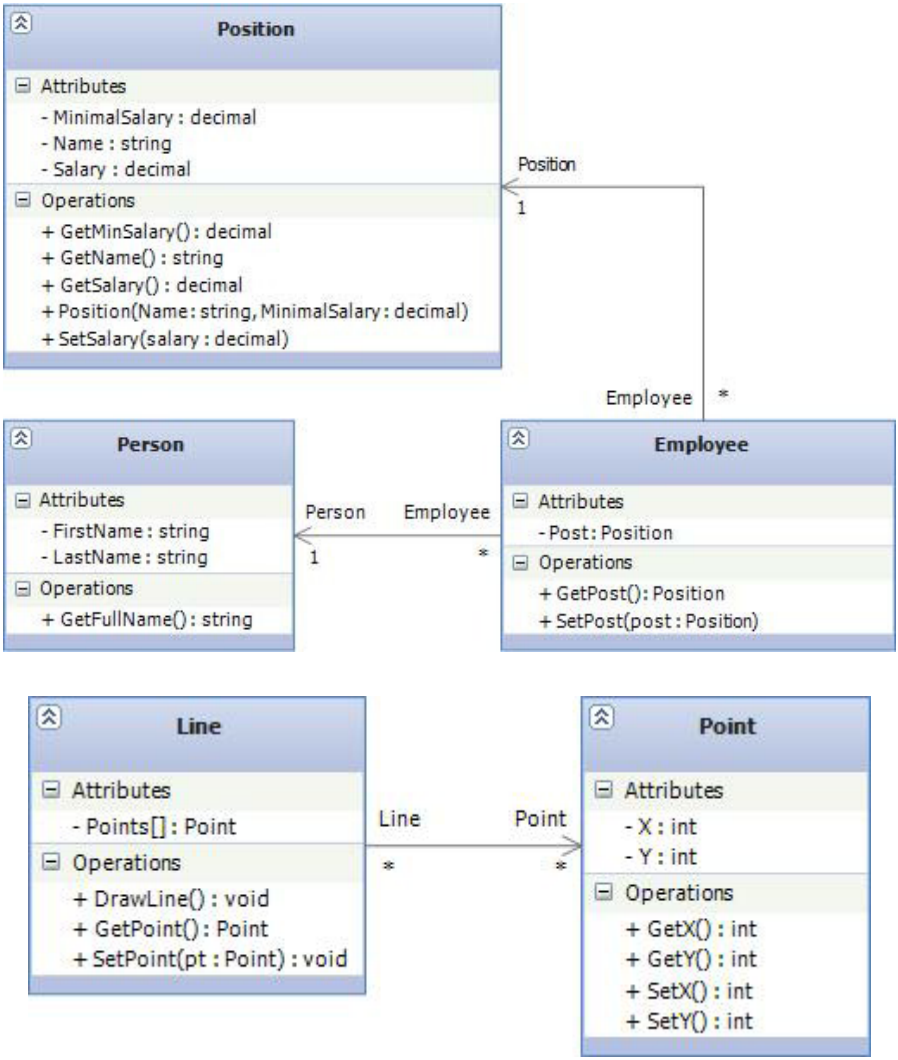
1.3.2. Отношение ассоциации (association relationship)

Отношение ассоциации – это такое отношение между двумя классификаторами, которое описывает причину отношения и правило, которое определяет отношение. С одной стороны, ассоциация определяет структурное отношение, которое объединяет два классификатора, но с другой – как атрибут, определяет свойства классификатора. Например, ассоциации могут быть использованы для того, чтобы представить проектные решения, которые были определены для классификаторов. Это может быть и иллюстрация того, как объект одного класса реализует доступ к объекту другого класса, а может быть и иллюстрация того, что между двумя классификаторами на концептуальном уровне имеется логическая связь.

Приведённый далее пример демонстрирует, что всякий объект, описывающий рабочего, может быть связан с одним объектом, описывающим человека, который является работником, и с одним объектом, описывающим должность. Это может толковаться так, что всякий человек может быть связан только с одной должностью, посредством объекта вспомогательного класса Employee.

Конец ассоциации может быть помечен меткой, которую называют «именем роли», и «кратностью», которая указывает, какое количество объектов может участвовать

в ассоциации. На приведённом ниже примере символ * возле каждого классификатора говорит о том, что, с одной стороны, любая линия может содержать любое количество точек, а, с другой, – любая точка может принадлежать любому количеству линий одновременно.



1.3.3. Отношение обобщения (generalization relationship)

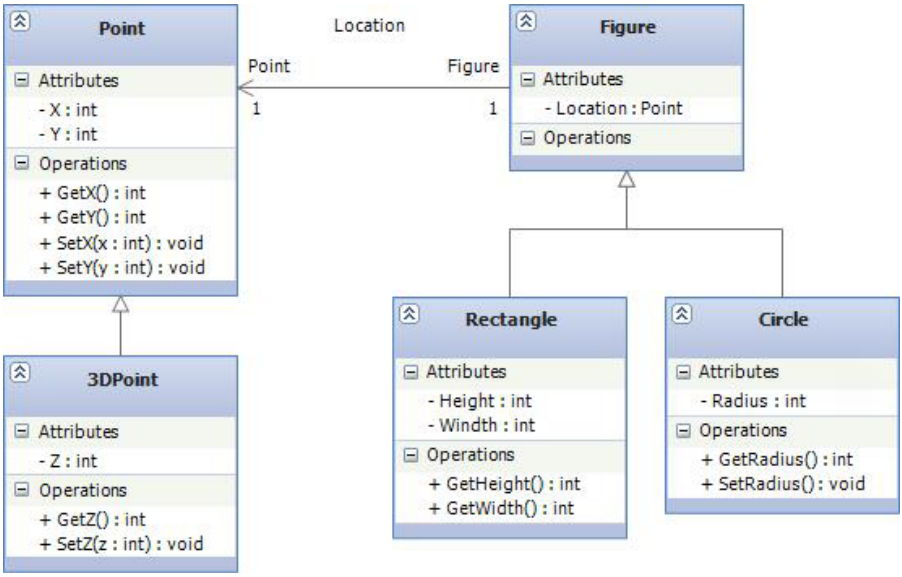
Обобщение – это отношение, при котором один элемент, который называют дочерним, базируется на другом элементе, называемом родительским. Отношение обобщения используется в диаграммах: классов, компонентов, развёртывания и вариантов использования, для того, чтобы проиллюстрировать, что дочерний элемент получает все атрибуты, операции и связи, определённые в базовом элементе.

Необходимо помнить, что отношение обобщения может использоваться только по отношению к однотипным классификаторам. Например, оно может быть использовано для двух классов, или для двух вариантов использования. Но не может существовать между классом и вариантом использования (о вариантах использования будет подробно рассказано в уроке, посвящённом тому вопросу). В отношении обобщения один родительский элемент может иметь один и более дочерних элементов. Один дочерний элемент может несколько родительских элементов, но наиболее предпочтительной практикой считается использование не более одного родительского элемента.

Отношения обобщения изображаются в виде сплошной стрелки, направленной от дочернего элемента к родительскому. Согласно синтаксису UML отношения обобщения не имеют имён.

На уровне реализации отношение обобщения можно условно соотнести с наследованием.

На представленном ниже примере иллюстрируется использование отношения обобщения в диаграммах классов.



1.3.4. Отношение реализации (realization relationship)

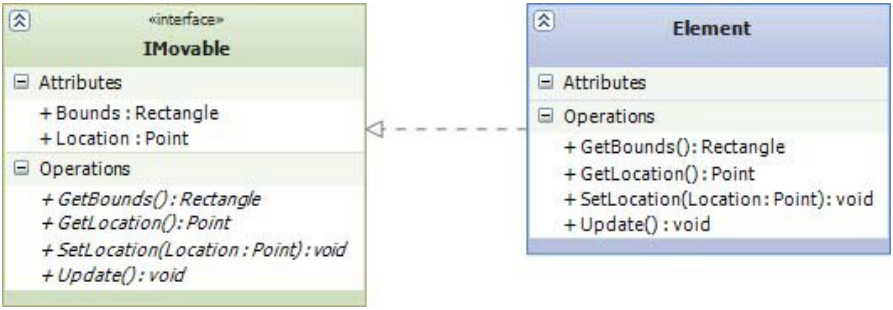
Отношение реализации – это отношение между двумя элементами модели, при котором один элемент модели (элемент-клиент) реализует поведение, которое определяется другим элементом модели (элемент-поставщик). Несколько «клиентов» могут реализовывать поведение одного «поставщика».

Отношение реализации может быть использовано в диаграммах классов и компонентов. Графически

отношение реализации изображается в виде пунктирной стрелки, направленной от клиента к поставщику.

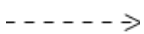
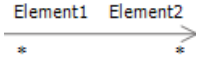
Обычно, как и отношение обобщения, отношение реализации не имеет имени. Но, если отношение именуется, то подпись должна находиться радом с соединительной линией, которая визуализирует отношение.

На приведённом ниже примере, отношение реализации демонстрирует, что интерфейс IMovable реализуется классом Element, который описывает графический элемент на диаграмме. На уровне реализации такое отношение будет выражено в виде наследования.



1.4. Сравнительный анализ отношений

Сравнивая отношения между собой можно прийти только к одному закономерному выводу, что все отношения без исключения необходимы и полезны при моделировании. Ниже приведена сравнительная таблица различных характеристик отношений.

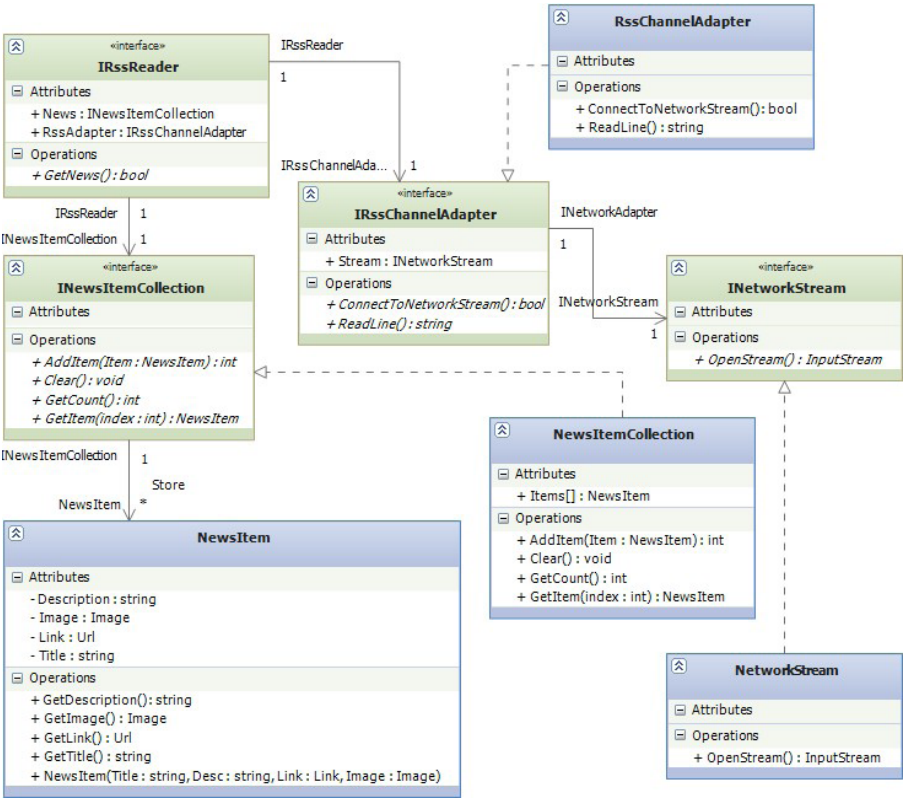
Характеристика	Отношение зависимости (dependency relationship)	Отношение ассоциации (association relationship)	Отношение обобщения (generalization relationship)	Отношение реализации (realization relationship)
Графическое отображение				
Наличие подписи	Не обязательно	Обязательны метки с сторон обоих классификаторов и возможно наличие подписи для самого отношения	Подписи не используются	Не обязательно
Диаграммы, в которых используется	Диаграммы классов, компонентов, развёртывания и вариантов использования	Во всех видах диаграмм	Диаграммы классов, компонентов, развёртывания и вариантов использования	Диаграммы классов и компонентов

1.5. Практические примеры построения диаграмм классов

Далее мы предлагаем пример оформления диаграммы классов, которая отражает концептуальную модель RSS-клиента. Заранее хотим оговорить, что в предложенной модели не учитывается возможность вывода новостей через графический интерфейс пользователя или даже в текстовом режиме, чтобы не усложнять диаграмму.

Концепция достаточно проста: согласно этой модели, предлагается описать тип данных для представления каждой отдельной новости, организовать новости в коллекцию, которая будет инкапсулироваться соответствующим классом, который будет так же посредством специального адаптера получать новости с RSS-канала и организовывать их в коллекцию объектов соответствующего класса, указанного выше.

Предложенная модель использует большинство описанных выше отношений.



Также хотим отметить, что данная модель не является совершенной и может содержать ряд дефектов, внесённых автором умышленно. Задача читателя состоит в том, чтобы спроектировать свой вариант предложенной модели (улучшенный и дополненный).

2. Диаграмма состояний

2.1. Цели данного типа диаграмм

Диаграмма состояний – это диаграмма, отображающая всё множество состояний, в которых может находиться объект за время его жизненного цикла, а так же все возможные переходы между этими состояниями, предусмотренные концепцией, разработанной проектировщиком.

В большинстве случаев диаграмма состояний строиться для одного класса, чтобы показать динамику поведения объектов этого класса.

Диаграммы состояний обычно используют для описания поведения некоторого объекта в различных вариантах использования. Тогда, как для описания поведения нескольких объектов целесообразнее использовать диаграммы взаимодействия.

Использование диаграмм состояний позволяет всем членам группы разработчиков одинаково представлять концепцию динамической составляющей системы, предложенную проектировщиком. Также данный тип диаграмм даёт возможность представить внешние (поведенческие) характеристики реализуемого класса, что позволяет спроектировать его внутреннюю структуру так, чтобы она максимально отвечала требованиям эксплуатации объектов данного класса.

2.2. Базовые понятия

2.2.1. Автоматы

Каждая диаграмма состояний представляет собой автомат. Под автоматом в UML понимается формализм, используемый для моделирования поведения элементов модели и самой системы. С другой стороны, автомат – это последовательность состояний, которые охватывают все этапы жизненного цикла объекта. Основным формализмами, используемыми при описании автомата, являются состояние и переход.

Принято считать, что длительность нахождения объекта в одном из возможных состояний намного превышает время, необходимое на осуществление перехода из одного состояния в другое.

В «идеале» предполагается, что на переход между состояниями совсем не затрачивается времени, или, другими словами, оно равно нулю.

2.2.2. Состояние

Понятие состояния

В языке моделирования UML под состоянием понимается метакласс, который используется для моделирования отдельно взятой ситуации, подразумевающей выполнение некоторого условия. Весь спектр состояний некоторого объекта или класса может выражаться в виде набора значений соответствующего атрибута класса. Подразумевается,

что изменение значения атрибута выражает изменение состояния объекта.

Но не любой атрибут может быть использован в качестве индикатора состояний. Обычно под состояниями подразумеваются такие атрибуты системы, которые отражают динамический аспект её поведения.

Важно также понимать, что в данном случае состояния – это всего лишь метафора, отражающая внутренние изменения, происходящие в системе.

Графически состояние изображается как овал со скруглёнными краями, в центре которого написано имя состояния. Пример изображения состояний показан на изображении слева.

Имя состояния

Текст, используемый в качестве имени состояния должен выражать содержательный смысл действий, выполняемых системой в этом конкретном состоянии (в конкретный момент её функционирования).

В качестве имён состояний следует выбирать глаголы настоящего времени, или соответствующие причастия. Так же имя всегда следует начинать с заглавной буквы.

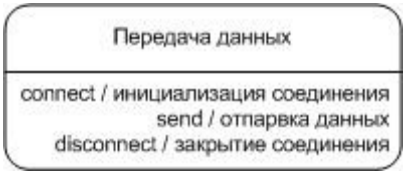
Список внутренних действий

Список внутренних действий содержит перечисление действий, которые система выполняет в соответствующем

состоянии. Действия следует указывать в порядке их выполнения системой. Каждое действие записывается с новой строки в следующем формате:

```
имя_ддействивя/содержание_действия
```

Справа на рисунке показано как будет выглядеть в графическом представлении событие с указанным списком внутренних действий для состояния передачи данных.



Начальное состояние

Начальное состояние – это частный случай состояния, в котором объект находиться в начальный момент времени. У начального состояния отсутствуют внутренние действия, потому что оно предшествует первым выполняемым объектам действиям, а значит и первому внутреннему состоянию объекта. Начальное состояние необходимо только для того, чтобы обозначить момент начала существования объекта. Исходя из вышесказанного, начальное и конечное состояние называют псевдо-состояниями.

Начальное состояние изображается в виде закрашенного круга.

Пример изображения начального состояния приведён на изображении ниже.

Начальное состояние	
Начальное состояние с переходом	

Конечное состояние

Конечное значение, как и начальное, является псевдосостоянием и отражает момент завершения жизненного цикла автомата. Графически конечное состояние представляется в виде закрашенного круга, помещённого в окружность, как это показано на изображении справа.

2.2.3. Переход

Понятие перехода

Переход (transition) – это отношение между двумя последовательными состояниями, которое указывает на смену одного состояния другим. То состояние, из которого выполняется переход, называют исходным состоянием, а то, в которое выполняется переход, – целевым состоянием.

Как правило, пребывая в некотором состоянии, система выполняет соответствующие этому состоянию действия. Поэтому переход будет возможен только после выполнения этих действия, а так же некоторых сопутствующих условий, которые соответствуют переходу в целевое состояние. Выполнение перехода между состояниями принято называть срабатыванием перехода.

Графически переход изображается сплошной стрелкой от исходного состояния к целевому, как показано на представленном далее изображении.

Если возле стрелки не указано никакой подписи, то такой переход считается нетриггерным (триггерные переходы – это переходы, которые происходят как реакция на некоторое событие. О событиях будет рассказано далее). Если переход указан как нетриггерный, то из контекста диаграммы должно быть однозначно понятно, после какого действия он осуществляется.

Каждый переход может быть, при необходимости, помечен строкой текста в следующем формате:

```
сигнатура_события[сторожевое условие]/выражение действия
```

В свою очередь сигнатура события определяет само событие, с необходимыми аргументами (аргументы разделяются запятыми) в следующем формате:

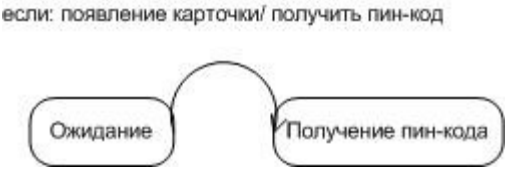
```
имя_события(список_параметров)
```

Событие

Событие (event) представляет собой декларацию некоторого факта, могущего иметь место в пространстве и времени. События строго упорядочены во времени, а значит, после того, как некоторое событие произошло, то вернуться к предыдущему уже невозможно (за исключением тех случаев, когда в модели явно указана цикличная последовательность событий).

События, как правило, указывают на некоторые внешние изменения, при переходе системы из одного состояния

в другое. Например, событие включения говорит о том, что система перешла из пассивного состоянияв активное. Или переключение представления документа в html-редакторе говорит о том, что графическая подсистема перешла из текстового представления в графическое.



Графически событие указывается как текст, подписанный возле перехода.

Приведённый справа рисунок иллюстрирует использование события.

Сторожевое условие

Сторожевое условие (guard condition) – это условие, которое определяет, в какой ситуации, или при каких значениях атрибутов, будет осуществлён переход между состояниями.

Сторожевое условие записывается в прямоугольных скобках после события-триггера. Необходимо помнить, что семантика условия, указанного в качестве сторожевого должна быть понятна из контекста диаграммы.

Выше показано, как выглядит в графической форме использование сторожевого условия.

2.2.4. Выражение действия

Выражение действия (action expression) это описание действия, которое выполняется сразу после того и только при условии того, как срабатывает переход. Выражение действия должно быть выполнено до начала выполнения любого действия в целевом состоянии. Выражение действия представляет собой атомарную операцию, то есть она не может быть прервана, до своего завершения, что исключает возможность обрывания или отката перехода или предотвратить переход в целевое состояние. Выражение действия указывается после знака «/» в имени перехода, к которому оно добавлено. Если используется несколько выражений действия, то они перечисляются через «,».

Ниже приведён пример использования выражения действий, а так же способ его графического представления.

2.3. Составное состояние и подсостояние

Составное состояние (composite state) – это состояние, состоящее из других, вложенных в него состояний. По отношению к вложенным состояниям, составное состояние

выступает в роли суперсостояния или надсостояние (superstate), а они, в свою очередь, к нему – как подсостояния (substates).

Графически составное состояние изображается как обычное состояние, но с вложенными вовнутрь него фигурами подсостояний и переходов между ними.

Возможно так же использование составных состояний со скрытой внутренней структурой. В таком случае подразумевается, что существует отдельная диаграмма состояний, которая описывает структуру данного составного состояния, а само это состояние описывается обычным образом, с указанием всех выполняющихся в нём переходов и символа составного состояния.

Составное состояние может содержать два и более параллельных подавтомата или несколько последовательных подсостояний.

2.3.1. Последовательные подсостояния

Последовательные подсостояния (sequential substates) используются для моделирования такого объекта, который на всём протяжении данного состояния может одновременно находиться в одном и только одном подсостоянии. Хотя поведение объекта такого типа и может быть выражено через последовательную смену обычных

состояний, использование составного состояния даёт возможность более тонко и детально представить специфику его поведения.

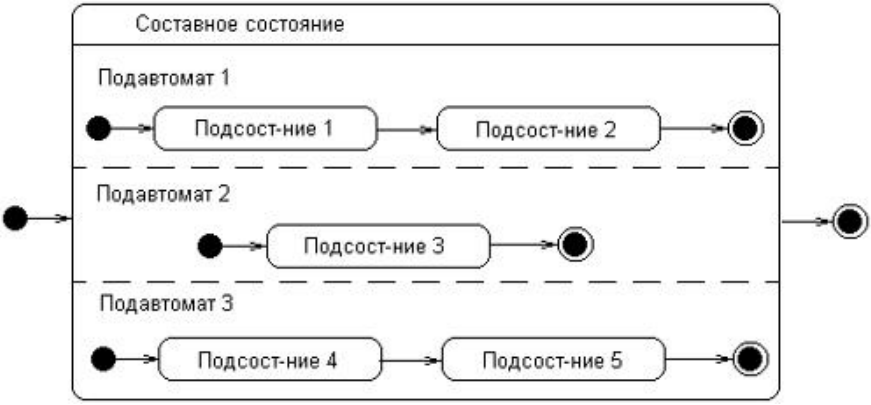
Составное состояние может содержать в качестве подсостояний начальное и конечное состояния.

Ни представленном справа изображении показано, как графически выглядит составное состояние иллюстрирующее последовательные подсостояния.

2.3.2. Параллельные подсостояния

Параллельные подсостояния (concurrent substates) позволяют представить состояние, в рамках которого параллельно выполняются несколько подавтоматов, каждый из которых отдельно представляет собой диаграмму состояний сравнительно небольшого объёма, помещённую во внутрь составного состояния.

На представленном ниже рисунке показан способ графического представления параллельных подавтоматов составного состояния. Из новых графических элементов можно выделить только пунктирные разделительные линии, которые визуально отделяют параллельные подсостояния друг от друга.



2.4. Историческое состояние

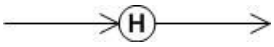
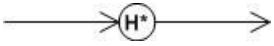
Существование исторического состояния обусловлено необходимость, в ряде случаев, выходить из текущего состояния с сохранением последующей возможности вернуться в него. При этом учитывается та часть деятельности, которая была выполнена до осуществления перехода.

Историческое состояние (history state) применяется в контексте составного состояния. Оно используется для запоминания того подсостояния, которое было текущим на моменты выходя из состояния.

Существует два вида исторического состояния: недавнее состояние (shallow history state) и давнее состояние или состояние глубокой истории (deep history state).

Графически состояние недавней истории изображается в виде окружности с заключённой в неё заглавной буквой «H», а состояние глубокой истории с буквой «H» и символом «*».

Ниже представлены графические обозначения для исторических состояний.

Состояние недавней истории	
Состояние глубокой истории	

Историческое состояние недавней истории должно быть первым подсостоянием в составном состоянии и при первом попадании в это историческое подсостояние, оно выступает в роли начального состояния подавтомата. Если во время работы подавтомата происходит выход, то историческое состояние запоминает, какое из подсостояний было активным в момент выхода, и при возвращении вызова в данный подавтомат, перенаправляет его к необходимому подсостоянию.

Поскольку запомненное подсостояние тоже может является составным, то существует историческое состояние глубокой истории, которое используется для запоминания всех подсостояний текущего подавтомата любого уровня вложенности.

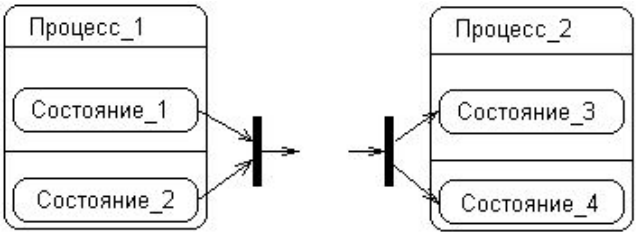
2.5. Сложные переходы

2.5.1. Переходы между параллельными состояниями

Параллельный переход – это переход, который может иметь несколько состояний-источников и несколько целевых состояний одновременно.

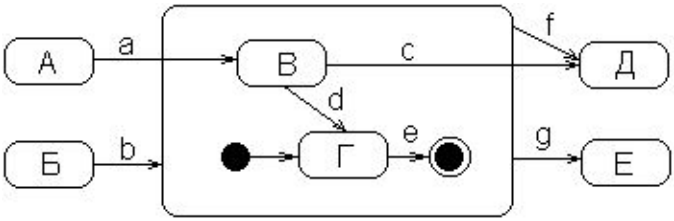
Графически такой переход обозначается закрашенным вертикальным прямоугольником, как показано на изображении.

Если переход имеет много входящих переходов, то его называют соединением или объединением (join), а в случае, когда много исходящих и одну входящую – ветвлением (fork).



2.5.3. Переходы между составными состояниями

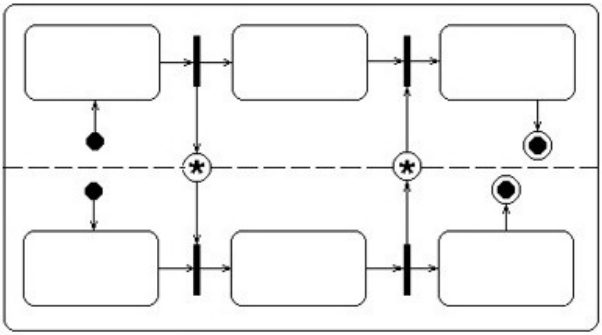
Переход в составное состояние обозначается стрелкой, направленной к границе этого составного состояния. Это графический элемент означает переход к начальному состоянию всех подавтоматов суперсостояния.



Переход, выходящий из границы составного состояния, предполагает выход некоторого объекта из подсостояния суперсостояния. Тогда как переход, направленный из вложенного подсостояния за границы суперсостояния, означает выход из составного состояния. На представленном справа рисунке изображены все перечислены переходы между составными состояниями.

2.5.4. Синхронизирующие состояния

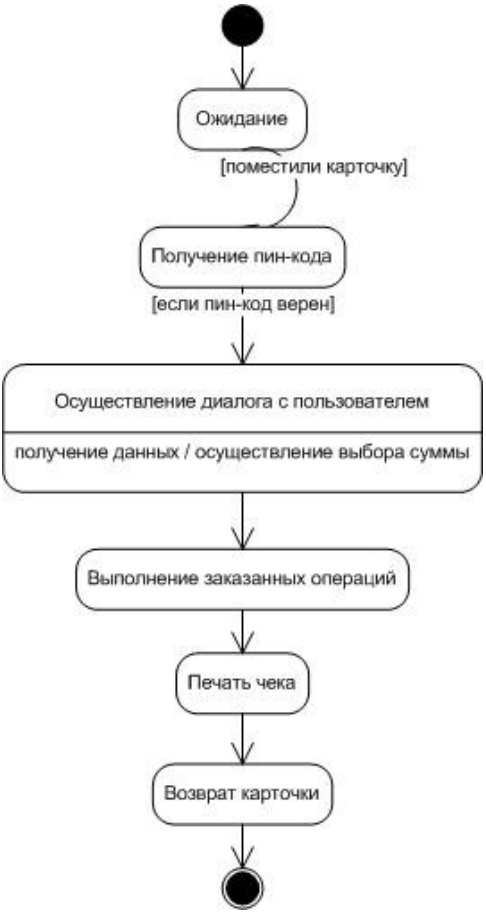
Синхронизирующее состояние (sync state) используется для синхронизации наступления отдельных событий в параллельных подавтоматах с целью их синхронизации. Синхронизирующее событие обозначается небольшой окружностью со вписанным в неё символом «*». Данное состояние используется совместно с переходом-ветвлением и переходом-объединением, для того, чтобы явно указать события в параллельных подавтоматах, оказывающие непосредственное влияние на поведение того подавтомата, для которого они установлены.



2.6. Практические примеры построения диаграмм состояний

В качестве примера мы рассмотрим диаграмму состояний для визуализации поведения устройства банкомат.

В рамках предложенной модели, предполагается, что банкомат находиться в режиме ожидания, до тех пор, пока пользователь не поместит карточку в приёмник. После этого банкомат осуществляет получение пин-кода. Если пин-код верен, то банкомат осуществляет диалог



с пользователем для получения необходимой ему суммы, а после этого осуществляет выполнения отложенных операций, печать чека и возврат карточки.

В данной модели умышленно не предусмотрена ситуация, при которой пользователь забыл пин-код и хочет получить карточку обратно. Или прекратить взаимодействие с банкоматом во время выполнения диалога.

Предлагаем Вам самостоятельно доработать предложенную модель в рамках домашнего задания.

3. Диаграмма деятельности

В процессе моделирования системы важно не только указать её структуру, характер связей между составляющими систему элементами, а также множества состояний, в которых может находиться система, и переходов между этими состояниями, но бывает совершенно необходимым специфицировать алгоритмические особенности функциональных модулей системы, определить характер протекающих в ней процессов.

Обычно для описания алгоритма используют алгоритмические языки, такие как: УАЯ (условный алгоритмический язык), язык блок-схем или структурные схемы алгоритмов. В языке UML для этих целей используются диаграммы деятельности (activity diagrams), предназначенные для представления алгоритмов в графическом виде.

Визуально диаграмма деятельности представляется в виде форме графа, вершинами которого являются состояния действия, а дугами – переходы от одного состояния действия к другому.

Под действиями (actions) понимаются элементарные вычисления, приводящие к некоторому конкретному результату.

3.1. Цели данного типа диаграмм

Основным целью использования диаграммы деятельности является визуализация особенностей реализации

операций классов, а так же описание реакций на внутренние события системы.

3.2. Базовые понятия

3.2.1. Состояние действия

Состояние действия (action state) – это состояние с входным действием и минимум одним исходящим переходом. В силу своей элементарности состояние действия не может иметь внутренних переходов. А условием перехода является завершение действия.

Как правило, действие соответствует одному шагу моделируемого алгоритма.

Графически состояние действия представляется прямоугольником со скруглёнными краями, внутри которого находиться текст, содержащий выражение-действия (action-expression). Выражение-действия должно быть уникальным в пределах одной диаграммы деятельности. Действие может быть записано на обычном языке (например: «сравнить идентификаторы»), а может – на некотором псевдокоде или существующем языке программирования. Если в качестве действия используется естественный язык, то для определения (именования) действия принято использовать глагол, как

правило, выражающего утверждение в побудительном наклонении, с дополнительными пояснениями (например: «выполнить проверку диска»).

Располагать действия в диаграммах деятельности принято сверху вниз.

Пример использования состояний деятельности представлен на изображении права.

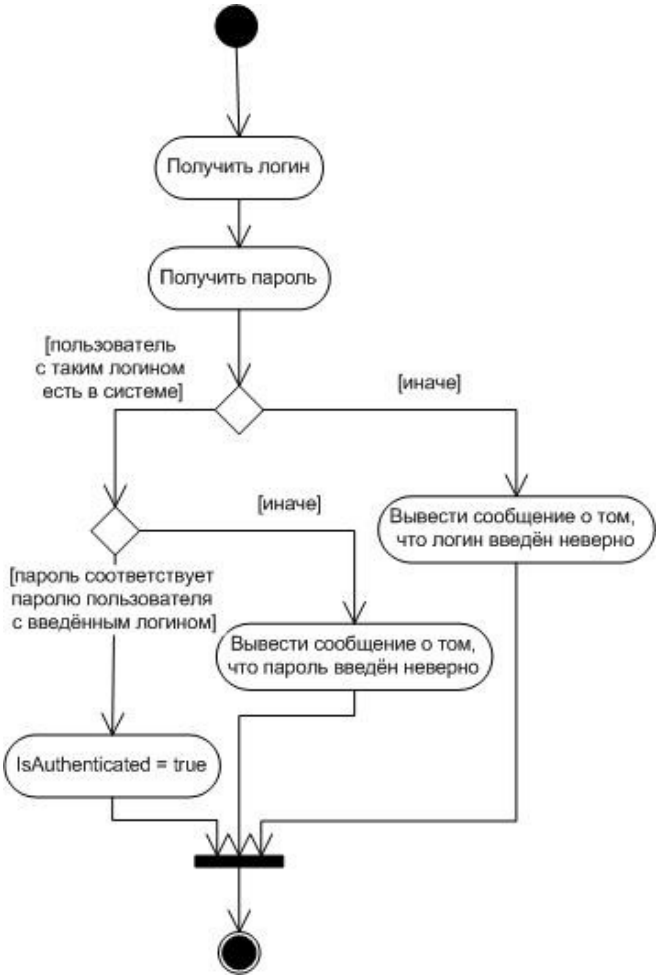
3.2.2. Переходы

Переход, как формализм был рассмотрен нами в предыдущей главе, посвящённой диаграммам состояний. В данном разделе мы рассмотрим только специфику использования переходов в диаграммах деятельности. Во-первых, следует отметить, что в диаграммах деятельности используются только нетриггерные переходы. То есть для перехода не нужно ждать наступления некоторого события, а переход срабатывает сразу после завершения действия.

Если из состояния действия исходит только один переход, то он, как правило, не имеет подписи. В случае, когда переходов более одного, то каждый из них следует пометить условием, которое называется «сторожевым», и указывается в прямоугольных скобках возле линии перехода. Очевидно, что все сторожевые условия должны взаимно исключать друг друга.

Сам переход, как и а диаграмме состояний, изображается в виде сплошной стрелки, направленной от исходного состояния к целевому.

В тех случаях, когда поток исполнения в зависимости от некоторого условия должен разделиться на два альтернативных направления, принято говорить о ветвлении.



Графически ветвление выражается при помощи ромба с одним входящим переходом и, как минимум, двумя исходящими.

Входящий переход принято соединять с верхней либо левой вершинами ромба. Для исходящих переходов ветвления, как уже указывалось выше, обязательно должны быть установлены сторожевые условия.

Слева приведён пример использования переходов и ветвления в диаграммах деятельности.

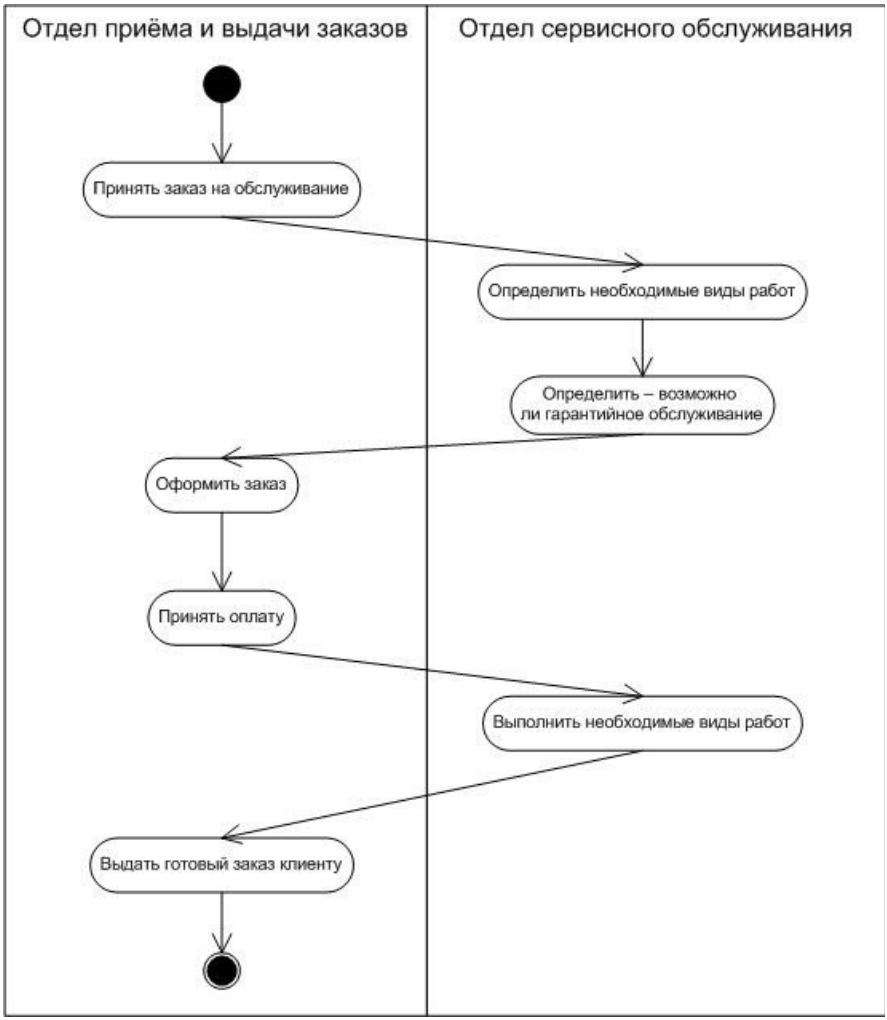
Отдельно хочется отметить один графический элемент, не описанный ранее – это объединение переходов. Объединение переходов используется в тех случаях, когда разные ветви кода «сходятся». Как можно увидеть на примере, объединение изображается в виде закрашенного прямоугольника. Объединение переходов может иметь множество входящих переходов и один исходящий.

3.2.3. Дорожки

Диаграммы деятельности могут использоваться не только для визуализации алгоритмов и спецификации прикладных протоколов моделируемой системы. На современном этапе развития программного обеспечения и сетей коммуникации на передний план выходит задача моделирования, так называемых, бизнес-процессов, то есть порядка выполнения процедур, связанных с обеспечением функционирования коммерческих структур и организаций.

Необходимо отметить, что совершенно необязательно, чтобы эти модели находили своё выражение в действующем программном обеспечении. Востребованным также является моделирование человеческих операций, с одной стороны – для их анализа, а с другой – для достижения наибольшей эффективности функционирования отдельных операторов и коллективов.

Но бизнес имеет свою специфику. Как правило, коммерческие организации имеют сложную разветвлённую структуру, разделяющую все функции предприятия между отделами и подразделениями, а бизнес-процесс



представляют в виде переходов действий из одного подразделения в другое.

Для визуализации этих переходов в UML созданы специальные графические конструкции, которые называю дорожками (swimlanes – с англ. плавательные дорожки

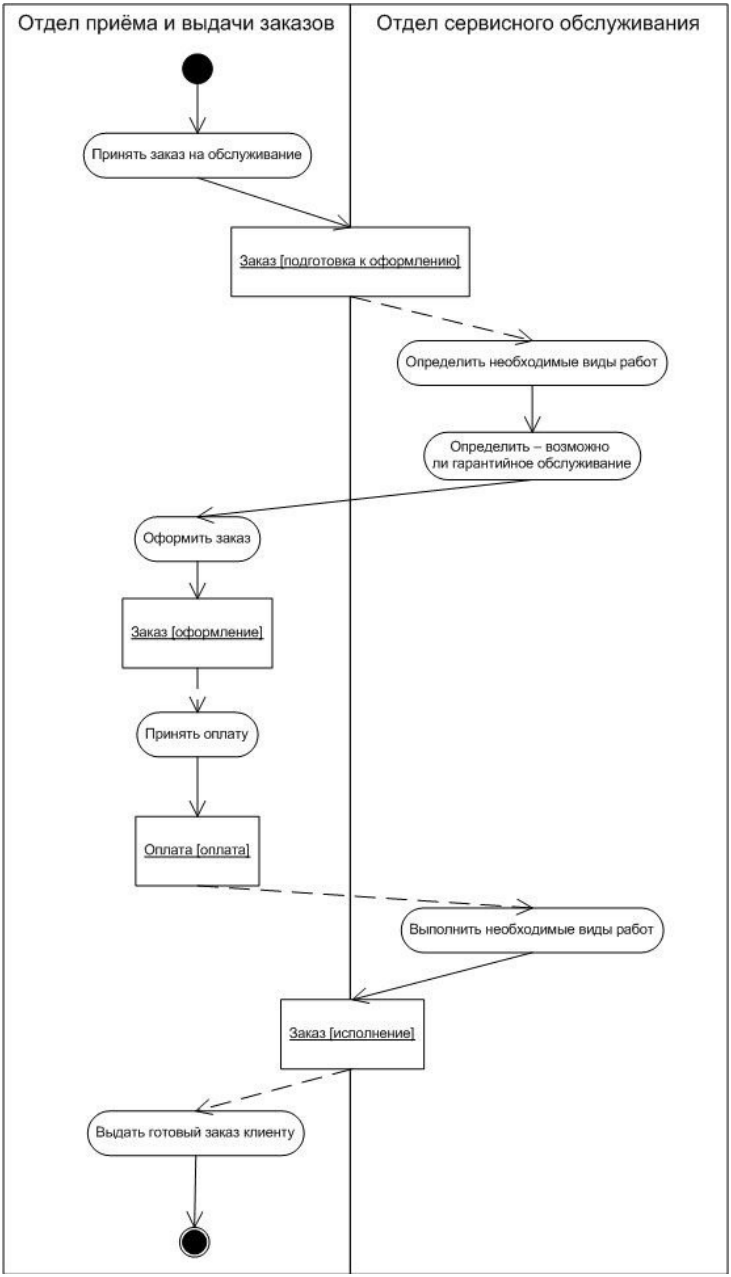
в бассейне). На диаграмме дорожки друг от друга отделены вертикальными линиями (они, как бы, разделяют диаграмму на отсеки), а в верхней части дорожки указывается текстовая подпись, соответствующая названию отдела, выполняющего действия, указанные на соответствующей дорожке.

Ниже иллюстрируется использования дорожек на примере моделирования приёма и исполнения заказов в центре сервисного обслуживания.

3.2.4. Объекты

В данном случае говоря об объектах их необходимо понимать в контексте бизнес-процессов, поскольку бизнес-процессы, обычно, имеют дело с некоторыми объектами. Например, в случае с описанием работы сервис центра – это заказ, принятый в отделе приёма и выдачи заказов и поступающий на обработку в отдел сервисного обслуживания.

За время прохождения всех этапов процесса заказ переходит из одного состояния в другое, которые в нашем случае определяет готовность заказа. Поэтому обычно говорят не об объекте, а только о его состояниях. Состояние объекта графически отображается практически так же, как и в диаграммах классов: он представляется в виде прямоугольника, в верхней части которого указано его имя (как правило, имя объекта подчёркивается); после имени через двоеточие может быть указан тип объекта, если он определён в модели. Отличие состоит только в том, что после имени объекта в прямоугольных скобках указывается состояние объекта при переходе между действиями.


Отдел приёма и выдачи заказов
Отдел сервисного обслуживания
Принять заказ на обслуживание
Заказ [подготовка к оформлению]
Определить необходимые виды работ
Определить – возможно ли гарантийное обслуживание
Оформить заказ
Заказ [оформление]
Принять оплату
Оплата [оплата]
Выполнить необходимые виды работ
Заказ [исполнение]
Выдать готовый заказ клиенту

В данном контексте действия рассматриваются как набор операций по изменению состояний целевого объекта.

Переход от действия к состоянию объекта изображается сплошной стрелкой, направленной от действия к состоянию объекта, а переход от состояния объекта к следующему действию изображается пунктирной стрелкой и называется потоком объектов.

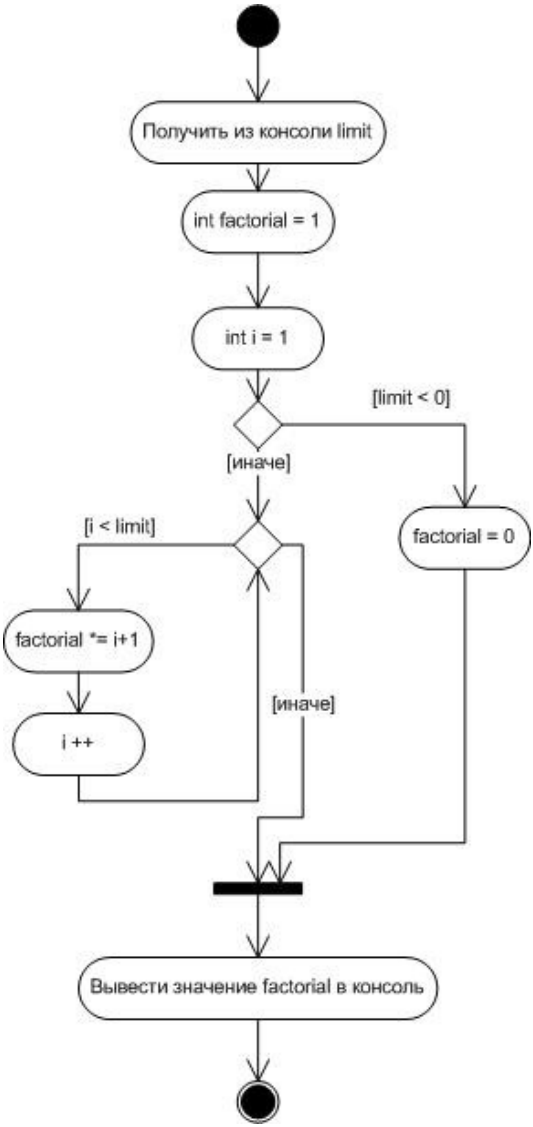
Необходимо отметить ещё один важный нюанс: объект может быть расположен на границе между дорожками. Такое расположение состояния объекта следует понимать как необходимость подготовки некоторых документов, сопровождающих бизнес-процесс, без которых завершения перехода объекта в другое состояние невозможно.

Ниже представлен предыдущий пример, но уже с использование состояний объекта «Заказ».

3.3. Практические примеры построения диаграмм деятельности

В качестве примера диаграммы деятельности мы хотим привести диаграмму, иллюстрирующую алгоритм определения факториала натурального числа. В данной модели представлена итеративная форма всем известного алгоритма.

Внимание следует обратить на то, что при использовании ветвления, в случае, когда определяются только две альтернативные ветви, условие можно указывать только для одной ветви, а во второй использовать утверждение «иначе», которое, в сущности, является условия альтернативной ветви.


Получить из консоли limit
int factorial = 1
int i = 1
[limit < 0]
[иначе]
[i < limit]
factorial = 0
factorial *= i+1
i ++
[иначе]
Вывести значение factorial в консоль