ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ АВТОМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ
КАФЕДРА ЭЛЕКТРОННЫХ ВЫЧИСЛИТЕЛЬНЫХ МАШИН

О.В. Караваева

Введение в язык UML

УЧЕБНОЕ ПОСОБИЕ

УЧЕБНОЕ ПОСОБИЕ

Спец. 230105.68

Киров 2010

УДК 004.434(07)
К210

Караваева О.В. Введение в язык UML: учебное пособие – Киров, 2010.- 58 с.

Курс лекций по дисциплине «Разработка и стандартизация программных средств и технологий». Пособие рассчитано на студентов очного и заочного отделений специальности 080801 «Прикладная информатика (в экономике)» и бакалавров по направлению 080800 Прикладная информатика (в экономике)»/ Сост. Караваева О.В. - Киров: КФ МФЮА, 2010.- 58 с.

Рецензент: кандидат технических наук, доцент кафедры ИСЭ ВятГУ В.У.Сысоев

Курс лекций по дисциплине «Разработка и стандартизация программных средств и технологий» охватывает одну, но очень важную тему – проектирование информационных систем с использованием средств объектно-ориентированного подхода к разработке программного продукта. В данном пособии рассматриваются основы языка UML, его свойства, общие механизмы. Здесь же описаны правила построения различных диаграмм для построения объектной модели предметной области.

Оглавление

Введение

Принципиальное различие между структурным и объектно-ориентированным подходом заключается в способе декомпозиции системы. Объектно-ориентированный подход использует объектную декомпозицию, при этом статическая структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениям и между объектами. Каждый объект системы обладает своим собственным поведением, моделирующим поведение объекта реального мира.

Понятие "объект" впервые было использовано около 30 лет назад в технических средствах при попытках отойти от традиционной архитектуры фон Неймана и преодолеть барьер между высоким уровнем программных абстракций и низким уровнем абстрагирования на уровне компьютеров. С объектно-ориентированной архитектурой также тесно связаны объектно-ориентированные операционные системы. Однако наиболее значительный вклад в объектный подход был внесен объектными и объектно-ориентированными языками программирования: Simula, Smalltalk, C++, Object Pascal. На объектный подход оказали влияние также развивавшиеся достаточно независимо методы моделирования баз данных, в особенности подход "сущность-связь".

Концептуальной основой объектно-ориентированного подхода является объектная модель. Основными ее элементами являются:

- абстрагирование (abstraction);
- инкапсуляция (encapsulation);
- модульность (modularity);
- иерархия (hierarchy).

Кроме основных имеются еще три дополнительных элемента, не являющихся в отличие от основных строго обязательными:

- типизация (typing);
- параллелизм (concurrency);

– устойчивость (persistence).

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

UML пригоден для моделирования любых систем: от информационных систем масштаба предприятия до распределенных Web-приложений и даже встроенных систем реального времени. Это очень выразительный язык, позволяющий рассмотреть систему со всех точек зрения, имеющих отношение к ее разработке и последующему развертыванию. Несмотря на обилие выразительных возможностей, этот язык прост для понимания и использования. Изучение UML удобнее всего начать с его концептуальной модели, которая включает в себя три основных элемента: базовые строительные блоки, правила, определяющие, как эти блоки могут сочетаться между собой, и некоторые общие механизмы языка.

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

1. Свойства языка UML

UML - это язык для визуализации, специфицирования, конструирования и документирования артефактов программных систем.

Язык состоит из словаря и правил, позволяющих комбинировать входящие в него слова и получать осмысленные конструкции. В языке моделирования словарь и правила ориентированы на концептуальное и физическое представление системы. Язык моделирования, подобный UML, является стандартным средством для составления "чертежей" программного обеспечения.

Моделирование необходимо для понимания системы. При этом единственной модели никогда не бывает достаточно. Для понимания любой нестандартной системы приходится разрабатывать большое количество взаимосвязанных моделей. В применении к программным системам это означает, что необходим язык, с помощью которого можно с различных точек зрения описать представления архитектуры системы на протяжении цикла ее разработки.

Словарь и правила такого языка, как UML, объясняют, как создавать и читать хорошо определенные модели, но ничего не сообщают о том, какие модели и в каких случаях нужно создавать. Это задача всего процесса разработки программного обеспечения. Хорошо организованный процесс должен подсказать, какие требуются объекты, какие ресурсы необходимы для их создания, как можно использовать эти объекты, чтобы оценить выполненную работу и управлять проектом в целом.

Некоторые особенности системы лучше всего моделировать в виде текста, другие - графически. Во всех интересных системах существуют структуры, которые невозможно представить с помощью одного лишь языка программирования. UML - графический язык, что позволяет решить проблем представления любых объектов и структур.

UML- это не просто набор графических символов. За каждым из них стоит хорошо определенная семантика. Это значит, что модель, написанная одним

разработчиком, может быть однозначно интерпретирована другим - или даже инструментальной программой. Так решается первая из перечисленных выше проблем.

UML позволяет специфицировать все существенные решения, касающиеся анализа, проектирования и реализации, которые должны приниматься в процессе разработки и развертывания системы программного обеспечения.

UML не является языком визуального программирования, но модели, созданные с его помощью, могут быть непосредственно переведены на различные языки программирования. Иными словами, UML-модель можно отобразить на такие языки, как Java, C++, Visual Basic, и даже на таблицы реляционной базы данных или устойчивые объекты объектно-ориентированной базы данных. Те понятия, которые предпочтительно передавать графически, так и представляются в UML; те же, которые лучше описывать в текстовом виде, выражаются с помощью языка программирования.

Такое отображение модели на язык программирования позволяет осуществлять, прямое проектирование: генерацию кода из модели UML в какой-то конкретный; язык. Можно решить и обратную задачу: реконструировать модель по имеющейся; реализации. Обратное проектирование не представляет собой ничего необычного. Если вы не закодировали информацию в реализации, то эта информация теряется! при прямом переходе от моделей к коду. Поэтому для обратного проектирования! необходимы как инструментальные средства, так и вмешательство человека. Сочетание прямой генерации кода и обратного проектирования позволяет работать как в графическом, так и в текстовом представлении, если инструментальные программы обеспечивают согласованность между обоими представлениями.

Помимо прямого отображения в языки программирования UML в силу своей выразительности и однозначности позволяет непосредственно исполнять модели, имитировать поведение систем и контролировать действующие системы.

Компания, выпускающая программные средства, помимо исполняемого кода производит и другие документы, в том числе следующие:

- требования к системе;
- архитектуру;
- исходный код;
- проектные планы;
- тесты;
- прототипы;
- версии, и др.

В зависимости от принятой методики разработки, выполнение одних работ производится более формально, чем других. Упомянутые документы необходимы для управления, для оценки результата, а также в качестве средства общения между членами коллектива во время разработки системы и после ее развертывания.

UML позволяет решить проблему документирования системной архитектуры и всех ее деталей, предлагает язык для формулирования требований к системе и определения тестов и, наконец, предоставляет средства для моделирования работ на этапе планирования проекта и управления версиями.

Язык UML предназначен прежде всего для разработки программных систем. Его использование особенно эффективно в информационных системах масштаба предприятия, в банковских и финансовых услугах; в телекоммуникации, науке, в распределенных Web-системы.

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

2. Концептуальная модель UML

Для понимания UML необходимо усвоить его концептуальную модель, которая включает в себя три составные части: основные строительные блоки языка, правила их сочетания и некоторые общие для всего языка механизмы. Усвоив эти элементы, вы сумеете читать модели на UML и самостоятельно создавать их - вначале, конечно, не очень сложные. По мере приобретения опыта в работе с языком вы научитесь пользоваться и более развитыми его возможностями.

Словарь языка UML включает три вида строительных блоков:

- сущности;
- отношения;
- диаграммы.

Сущности - это абстракции, являющиеся основными элементами модели. Отношения связывают различные сущности; диаграммы группируют представляющие интерес совокупности сущностей.

В UML имеется четыре типа сущностей:

- структурные;
- поведенческие;
- группирующие;
- аннотационные.

Сущности являются основными объектно-ориентированными блоками языка. С их помощью можно создавать корректные модели.

Структурные сущности - это имена существительные в моделях на языке UML Как правило, они представляют собой статические части модели, соответствующие концептуальным или физическим элементам системы. Существует семь разновидностей структурных сущностей.

Класс (Class) - это описание совокупности объектов с общими атрибутами, операциями, отношениями и семантикой Класс реализует один или несколько интерфейсов. Графически класс изображается в виде прямоугольника, в котором обычно записаны его имя, атрибуты и операции, как показано на рис. 2.1.

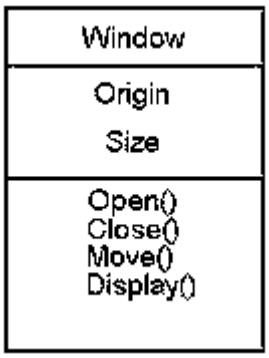


Рис. 2.1 Классы

Интерфейс (Interface) - это совокупность операций, которые определяют сервис (набор услуг), предоставляемый классом или компонентом. Таким образом, интерфейс описывает видимое извне поведение элемента. Интерфейс может представлять поведение класса или компонента полностью или частично; он определяет только спецификации операций (сигнатуры), но никогда - их реализации. Графически интерфейс изображается в виде круга, под которым пишется его имя, как показано на рис. 2.2. Интерфейс редко существует сам по себе - обычно он присоединяется к реализующему его классу или компоненту.

Рис. 2.2 Интерфейсы

Кооперация (Collaboration) определяет взаимодействие; она представляет собой совокупность ролей и других элементов, которые, работая совместно, производят некоторый кооперативный эффект, не сводящийся к простой сумме слагаемых. Кооперация, следовательно, имеет как структурный, так и поведенческий аспект. Один и тот же класс может принимать участие в нескольких кооперациях; таким образом, они являются реализацией образцов поведения, формирующих систему. Графически кооперация изображается в виде эллипса, ограниченного пунктирной линией, в который обычно заключено только имя, как показано на рис. 2.3.

Рис. 2.3 Кооперации

Прецедент (Use case) - это описание последовательности выполняемых системой действий, которая производит наблюдаемый результат, значимый для какого-то определенного ***актера*** (Actor). Прецедент применяется для структурирования поведенческих сущностей модели. Прецеденты реализуются посредством кооперации. Графически прецедент изображается в виде ограниченного непрерывной линией эллипса, обычно содержащего только его имя как показано на рис. 2.4.

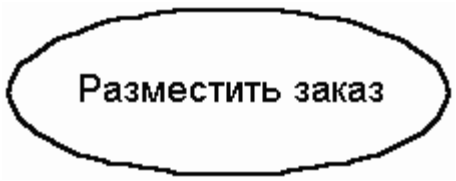


Рис. 2.4 Прецеденты

Три другие сущности - активные классы, компоненты и узлы - подобны классам: они описывают совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. Тем не менее, они в достаточной степени отличаются друг от друга и от классов и, учитывая их важность при моделировании определенных аспектов объектно-ориентированных систем, заслуживают специального рассмотрения.

Активным классом (Active class) называется класс, объекты которого вовлечены в один или несколько процессов, или нитей (Threads), и поэтому могут инициировать управляющее воздействие. Активный класс во всем подобен обычному классу, за исключением того, что его объекты представляют собой элементы, деятельность которых осуществляется одновременно с деятельностью других элементов. Графически активный класс изображается так же, как простой класс, но ограничивающий прямоугольник рисуется жирной линией и обычно включает имя, атрибуты и операции, как показано на рис. 2.5.

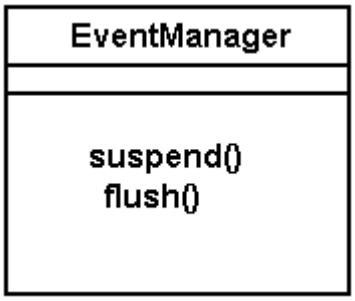


Рис. 2.5 Активные классы

Два оставшихся элемента - компоненты и узлы - также имеют свои особенности. Они соответствуют физическим сущностям системы, в то время как пять предыдущих - концептуальным и логическим сущностям.

Компонент (Component) - это физическая заменяемая часть системы, которая соответствует некоторому набору интерфейсов и обеспечивает его реализацию. В системе можно встретить различные виды устанавливаемых компонентов, такие как COM+ или Java Beans, а также другие компоненты процесса разработки, например файлы исходного кода. Компонент, как правило, представляет собой физическую упаковку логических элементов, таких как классы, интерфейсы и кооперации. Графически компонент изображается в виде прямоугольника с вкладками, содержащего обычно только имя, как показано на рис. 2.6.

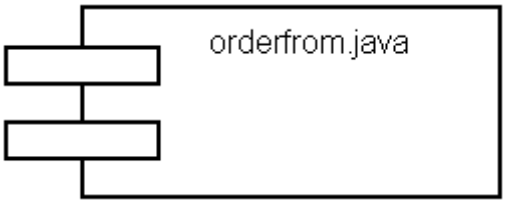


Рис. 2.6. Компоненты

Узел (Node) - это элемент реальной (физической) системы, который существует во время функционирования программного комплекса и представляет собой вычислительный ресурс, обычно обладающий как минимум некоторым объемом памяти, а часто еще и способностью обработки. Совокупность компонентов может размещаться в узле, а также мигрировать с одного узла на другой. Графически узел изображается в виде куба, обычно содержащего только имя, как показано на рис. 2.7.

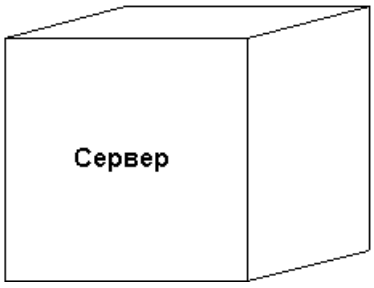


Рис. 2.7. Узлы

Эти семь базовых элементов - классы, интерфейсы, кооперации, прецеденты, активные классы, компоненты узлы - являются основными структурными сущностями, которые могут быть включены в модель UML. Существуют также разновидности этих сущностей: актеры, сигналы, утилиты (виды классов), процессы и нити (виды активных классов), приложения, документы, файлы, библиотеки, страницы и таблицы (виды компонентов).

Поведенческие сущности (Behavioral things) являются динамическими составляющими модели UML. Это глаголы языка: они описывают поведение модели во времени и пространстве. Существует всего два основных типа поведенческих сущностей.

Взаимодействие (Interaction) - это поведение, суть которого заключается в обмене сообщениями (Messages) между объектами в рамках конкретного контекста для достижения определенной цели. С помощью взаимодействия южно описать как отдельную операцию, так и поведение совокупности объектов. взаимодействие предполагает ряд других элементов, таких как сообщения, последовательности действий (поведение, инициированное сообщением) и связи (между объектами). Графически сообщения изображаются в виде стрелки, над которой почти всегда пишется имя соответствующей операции, как показано на рис. 2.8.

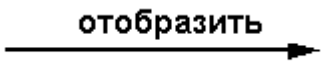


Рис 2.8. Сообщения

Автомат (State machine) - это алгоритм поведения, определяющий последовательность состояний, через которые объект или взаимодействие проходят на протяжении своего жизненного цикла в ответ на различные события, а также реакции на эти события. С помощью автомата можно описать поведение сдельного класса или кооперации классов. С автоматом связан ряд других элементов: состояния, переходы (из одного состояния в другое), события (сущности, инициирующие переходы) и виды действий (реакция на переход). Графически состояние изображается в виде прямоугольника с закругленными углами, содержащего имя (рис. 2.9).

Рис. 2.9 Состояния

Эти два элемента - взаимодействия и автоматы - являются основными поведенческими сущностями, входящими в модель UML. Семантически они часто бывают связаны с различными структурными элементами, в первую очередь - классами, кооперациями и объектами.

Группирующие сущности являются организующими частями модели UML. Это блоки, на которые можно разложить модель. Есть только одна первичная группирующая сущность, а именно пакет.

Пакеты (Packages) представляют собой универсальный механизм организации элементов в группы. В пакет можно поместить структурные, поведенческие и даже другие группирующие сущности. В отличие от компонентов, существующих во время работы программы, пакеты носят чисто концептуальный характер, то есть существуют только во время разработки. Изображается пакет в виде папки с закладкой, содержащей, как правило, только имя и иногда - содержимое (рис. 2.10). Пакеты - это основные группирующие сущности, с помощью которых можно организовать модель UML.

Рис. 2.10. Пакеты

Аннотационные сущности - пояснительные части модели UML. Это комментарии для дополнительного описания, разъяснения или замечания к любому элементу модели. Имеется только один базовый тип аннотационных элементов - примечание (Note***).*** *Примечание* - это просто символ для изображения комментариев или ограничений, присоединенных к элементу или группе элементов. Графически примечание изображается в виде прямоугольника с загнутым краем, содержащим текстовый или графический комментарий, как показано на рис. 2.11.

Рис. 2.11. Примечания

Этот элемент является основной аннотационной сущностью, которую можно включать в модель UML. Чаще всего примечания используются, чтобы снабдить диаграммы комментариями или ограничениями, которые можно выразить в виде неформального или формального текста. Существуют вариации этого элемента, например требования, где описывают некое желательное поведение с точки зрения внешней по отношению к модели.

В языке UML определены четыре типа ***отношений:***

- зависимость;
- ассоциация;
- обобщение;
- реализация.

Эти отношения являются основными связующими блоками в UML и применяются для создания корректных моделей.

Зависимость (Dependency) - это семантическое отношение между двумя сущностями, при котором изменение одной из них, независимой, может повлиять на семантику другой, зависимой. Графически зависимость изображается в виде прямой пунктирной линии, часто со стрелкой, которая может содержать метку (рис. 2.12).

Ассоциация (Association) - структурное отношение, описывающее совокупность связей; связь - это соединение между объектами. Разновидностью ассоциации является *агрегирование* (Aggregation) - так называют структурное отношение между целым и его частями. Графически ассоциация изображается в виде прямой линии (иногда завершающейся стрелкой или содержащей метку), рядом с которой могут присутствовать дополнительные обозначения, на пример кратность и имена ролей. На рис. 2.13 показан пример отношений этого типа.

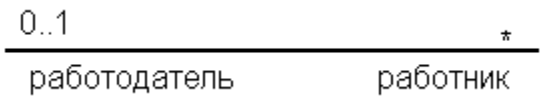


Рис. 2.13. Ассоциации

Обобщение (Generalization) - это отношение "специализация/обобщение", при котором объект специализированного элемента (потомок) может быть подставлен вместо объекта обобщенного элемента (родителя или предка).

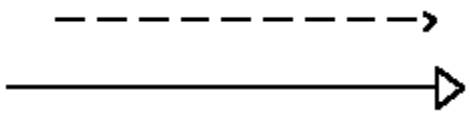


Рис. 2.14. Обобщения

Таким образом, потомок (Child) наследует структуру и поведение своего родителя (Parent). Графически отношение обобщения изображается в виде линии с не закрашенной стрелкой, указывающей на родителя, как показано на рис. 2.14.

Реализация (Realization) - это семантическое отношение между классификаторами, при котором один классификатор определяет "контракт", а другой гарантирует его выполнение. Отношения реализации встречаются в двух случаях: во-первых, между интерфейсами и реализующими их классами или компонентами, а во-вторых, между прецедентами и реализующими их кооперациями. Отношение реализации изображается в виде пунктирной линии с не закрашенной стрелкой, как нечто среднее между отношениями обобщения и зависимости (рис. 2.15).

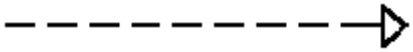


Рис. 2.15. Реализации

Четыре описанных элемента являются основными типами отношений, которые можно включать в модели UML. Существуют также их вариации, например *уточнение* (Refinement), *трассировка* (Trace), *включение* и *расширение* (для зависимостей).

Диаграмма в UML - это графическое представление набора элементов, изображаемое чаще всего в виде связанного графа с вершинами (сущностями) и ребрами (отношениями). Диаграммы рисуют для визуализации системы с разных точек зрения. Диаграмма - в некотором смысле одна из проекций системы. Как правило, за исключением наиболее тривиальных случаев, диаграммы дают свернутое представление элементов, из которых составлена система. Один и тот же элемент может присутствовать во всех диаграммах, или только в нескольких (самый распространенный вариант), или не присутствовать ни в одной (очень редко). Теоретически диаграммы могут содержать любые комбинации сущностей и отношений. На практике, однако, применяется сравнительно небольшое количество типовых комбинаций, соответствующих пяти наиболее употребительным видам,

которые составляют архитектуру программной системы (см. следующий раздел). Таким образом, в UML выделяют девять типов диаграмм:

- диаграммы классов;
- диаграммы объектов;
- диаграммы прецедентов;
- диаграммы последовательностей;
- диаграммы кооперации;
- диаграммы состояний;
- диаграммы действий;
- диаграммы компонентов;

На ***диаграмме классов*** показывают классы, интерфейсы, объекты и кооперации, а также их отношения. При моделировании объектно-ориентированных систем этот тип диаграмм используют чаще всего. Диаграммы классов соответствуют статическому виду системы с точки зрения проектирования. Диаграммы классов, которые включают активные классы, соответствуют статическому виду системы с точки зрения процессов.

На ***диаграмме объектов*** представлены объекты и отношения между ними. Они являются статическими "фотографиями" экземпляров сущностей, показанных на диаграммах классов. Диаграммы объектов, как и диаграммы классов, относятся к статическому виду системы с точки зрения проектирования или процессов, но с расчетом на настоящую или макетную реализацию.

На ***диаграмме прецедентов*** представлены прецеденты и актеры (частный случай классов), а также отношения между ними. Диаграммы прецедентов относятся к статическому виду системы с точки зрения прецедентов использования. Они особенно важны при организации и моделировании поведения системы.

Диаграммы последовательностей и кооперации являются частными случаями диаграмм взаимодействия. На ***диаграммах взаимодействия*** представлены связи между объектами; показаны, в частности, сообщения, которыми объекты могут

обмениваться.Диаграммы взаимодействия относятся к динамическому виду системы. При этом ***диаграммы последовательности*** отражают временную упорядоченность сообщений, а ***диаграммы кооперации*** - структурную организацию обменивающихся сообщениями объектов. Эти диаграммы являются изоморфными, то есть могут быть преобразованы друг в друга.

На диаграммах состояний (Statechart diagrams) представлен автомат, включающий в себя состояния, переходы, события и виды действий. Диаграммы состояний относятся к динамическому виду системы; особенно они важны при моделировании поведения интерфейса, класса или кооперации. Они акцентируют внимание на поведении объекта, зависящем от последовательности событий, что очень полезно для моделирования реактивных систем.

Диаграмма деятельности - это частный случай диаграммы состояний; на ней представлены переходы потока управления от одной деятельности к другой внутри системы. Диаграммы деятельности относятся к динамическому виду системы; они наиболее важны при моделировании ее функционирования и отражают поток управления между объектами.

На ***диаграмме компонентов*** представлена организация совокупности компонентов и существующие между ними зависимости. Диаграммы компонентов относятся к статическому виду системы с точки зрения реализации. Они могут быть соотнесены с диаграммами классов, так как компонент обычно отображается на один или несколько классов, интерфейсов или кооперацией.

Здесь приведен неполный список диаграмм, применяемых в UML. Инструментальные средства позволяют генерировать и другие диаграммы, но девять перечисленных встречаются на практике чаще всего.

3. Правила языка UML

Строительные блоки UML нельзя произвольно объединять друг с другом. Как и любой другой язык, UML характеризуется набором правил, определяющих, как должна выглядеть ***хорошо оформленная модель***, то есть семантически самосогласованная и находящаяся в гармонии со всеми моделями, которые с нею связаны.

В языке UML имеются семантические правила, позволяющие корректно и однозначно определять:

- ***имена,*** которые можно давать сущностям, отношениям и диаграммам;
- ***область действия*** (контекст, в котором имя имеет некоторое значение);
- ***видимость*** (когда имена видимы и могут использоваться другими элементами);
- ***целостность*** (как элементы должны правильно и согласованно соотноситься друг с другом);
- ***выполнение*** (что значит выполнить или имитировать некоторую динамическую модель).

Модели, создаваемые в процессе разработки программных систем, эволюционируют со временем и могут неоднозначно рассматриваться разными участниками проекта в разное время. По этой причине создаются не только хорошо оформленные модели, но и такие, которые:

- содержат скрытые элементы (ряд элементов не показывают, чтобы упростить восприятие);
- неполные (отдельные элементы пропущены);
- несогласованные (целостность модели не гарантируется).

Появление не слишком хорошо оформленных моделей неизбежно в процессе разработки, пока не все детали системы прояснились в полной мере. Правила языка UML побуждают - хотя не требуют - в ходе работы над моделью решать наиболее

важные вопросы анализа, проектирования и реализации, в результате чего модель со временем становится хорошо оформленной.

3.1 Общие механизмы языка UML

Моделирование упрощается и ведется более эффективно, если придерживаться некоторых соглашений. Работу с языком UML существенно облегчает последовательное использование общих механизмов, перечисленных ниже:

- спецификации (Specifications);
- дополнения (Adornments);
- принятые деления (Common Pisions);
- механизмы расширения (Extensibility mechanisms).

UML - это не просто графический язык. За каждой частью его системы графической нотации стоит ***спецификация,*** содержащая текстовое представление синтаксиса и семантики соответствующего строительного блока. Например, пиктограмме класса соответствует спецификация, полностью описывающая его атрибуты, операции (включая полные сигнатуры) и поведение, хотя визуально пиктограмма порой отражает только малую часть этой совокупности. Более того, может существовать другое представление этого класса, отражающее совершенно иные его аспекты, но тем не менее соответствующее все той же спецификации. С помощью графической нотации UML вы визуализируете систему, с помощью спецификаций UML - описываете ее детали. Таким образом, допустимо строить модель инкрементно, то есть пошаговым образом - сначала нарисовать диаграмму, а потом добавить семантику в спецификацию модели, или наоборот - начать со спецификации (возможно, применив обратное проектирование к существующей системе), а потом на ее основе создавать диаграммы.

Спецификации UML создают семантический задний план, который полностью включает в себя составные части всех моделей системы, согласованные между собой. Таким образом, диаграммы UML можно считать визуальными проекциями на этот задний план, при этом каждая из них раскрывает один из значимых аспектов системы.

Почти каждый из элементов UML имеет соответствующее ему уникальное графическое обозначение, которое дает визуальное представление о самых важных аспектах этого элемента. Например, обозначение класса специально придумано так, чтобы его было легко рисовать, поскольку классы - наиболее употребительный элемент при моделировании объектно-ориентированных систем. Нотация класса содержит самые важные его характеристики: имя, атрибуты и операции.

Спецификация класса может содержать и другие детали, например видимость атрибутов и операций или указание на то, что класс является абстрактным. Многие такие детали, можно визуализировать в виде графических или текстовых дополнений к стандартному прямоугольнику, служащему изображением класса. Так, на рис. 3.1 показан класс, в обозначение которого включены сведения о том, что он абстрактный и содержит две открытые, одну защищенную и одну закрытую операцию.

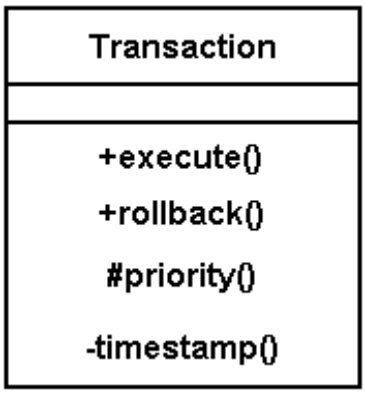


Рис. 3.1 Дополнения

Каждый элемент нотации UML содержит базовый для него символ, к которому можно добавлять разнообразные специфичные для него дополнения.

Принятые деления. При моделировании объектно-ориентированных систем реальность членится с учетом по крайней мере двух подходов.

Прежде всего, существует разделение на классы и объекты. Класс - это абстракция, объект - конкретная материализация этой абстракции. В языке UML можно моделировать и классы, и объекты, как показано на рис. 3.2. На этом рисунке показан один класс *Customer*(Клиент) и три объекта: *Jan*(явно определенный как объект данного класса), :*Customer*(анонимный объект класса *Customer*) и *Elyse*(спецификация которого относит его к классу *Customer*, хотя это и не выражено явно).

Практически все строительные блоки UML характеризуются дихотомией "класс/объект". Так, имеются прецеденты и экземпляры прецедентов, компоненты и экземпляры компонентов, узлы и экземпляры узлов и т.д. В графическом

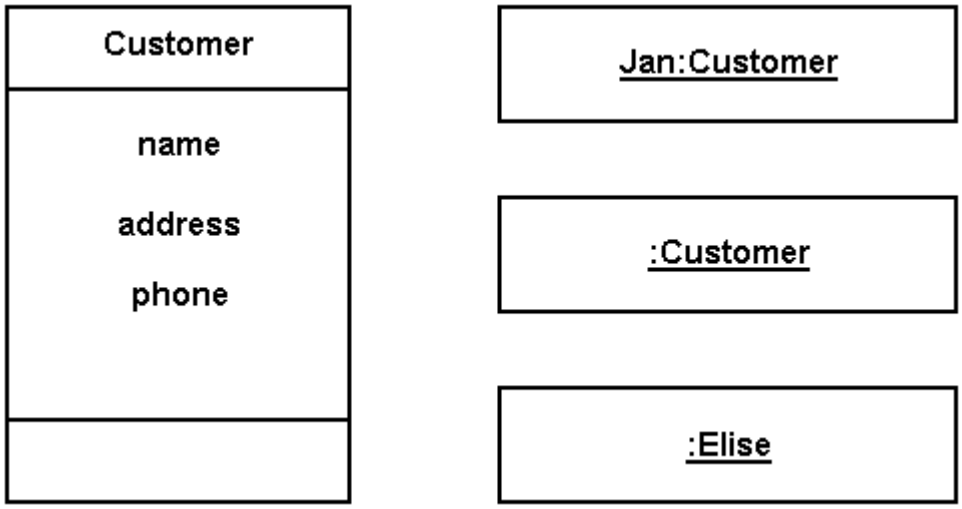


Рис. 3.2. Классы и объекты

представлении для объекта принято использовать тот же символ, что и для его класса, а название объекта подчеркивать.

Еще одним вариантом членения является деление на интерфейс и его реализацию. Интерфейс декларирует контракт, а реализация представляет конкретное воплощение этого контракта и обязуется точно следовать объявленной семантике интерфейса. UML позволяет моделировать обе эти категории, интерфейсы и их реализации, как показано на рис. 3.3: в данном случае один компонент *spellingwizard.dll* реализует два интерфейса *IUnknownu ISpelling*. Почти все строительные блоки UML характеризуются связкой

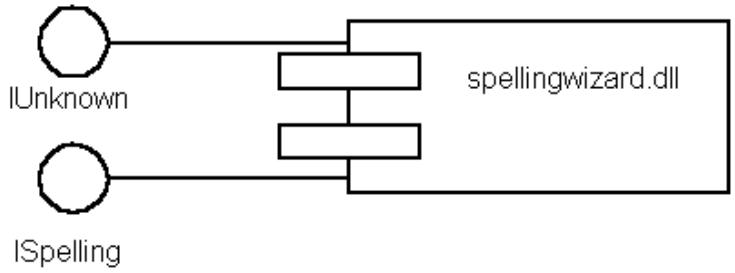


Рис. 3.3. Интерфейсы и реализация

"интерфейс/реализация". Например, прецеденты реализуются кооперациями, а операции - методами.

3.2 Архитектура

Для визуализации, специфицирования, конструирования и документирования программных систем необходимо рассматривать их с различных точек зрения. Все, кто имеет отношение к проекту, - конечные пользователи, аналитики, разработчики, системные интеграторы, тестировщики, технические писатели и менеджеры проектов - преследуют собственные интересы, и каждый смотрит на создаваемую систему по-разному в различные моменты ее жизни. Системная архитектура является, пожалуй, наиболее важным артефактом, который используется для управления всевозможными точками зрения и тем самым способствует итеративной и инкрементной разработке системы на всем протяжении ее жизненного цикла.

Архитектура - это совокупность существенных решений касательно:

– организации программной системы;

– выбора структурных элементов, составляющих систему, и их интерфейсов;

– поведения этих элементов, специфицированного в кооперациях с другими элементами;

– составления из этих структурных и поведенческих элементов все более и более крупных подсистем;

– архитектурного стиля, направляющего и определяющего всю организацию системы: статические и динамические элементы, их интерфейсы, кооперации и способ их объединения.

Архитектура программной системы охватывает не только ее структурные и поведенческие аспекты, но и использование, функциональность, производительность, гибкость, возможности повторного применения, полноту, экономические и технологические ограничения и компромиссы, а также эстетические вопросы.

Как показано на рис. 3.4, архитектура программной системы наиболее оптимально может быть описана с помощью пяти взаимосвязанных видов или представлений, каждый из которых является одной из возможных проекций организации и структуры системы и заостряет внимание на определенном аспекте ее функционирования.

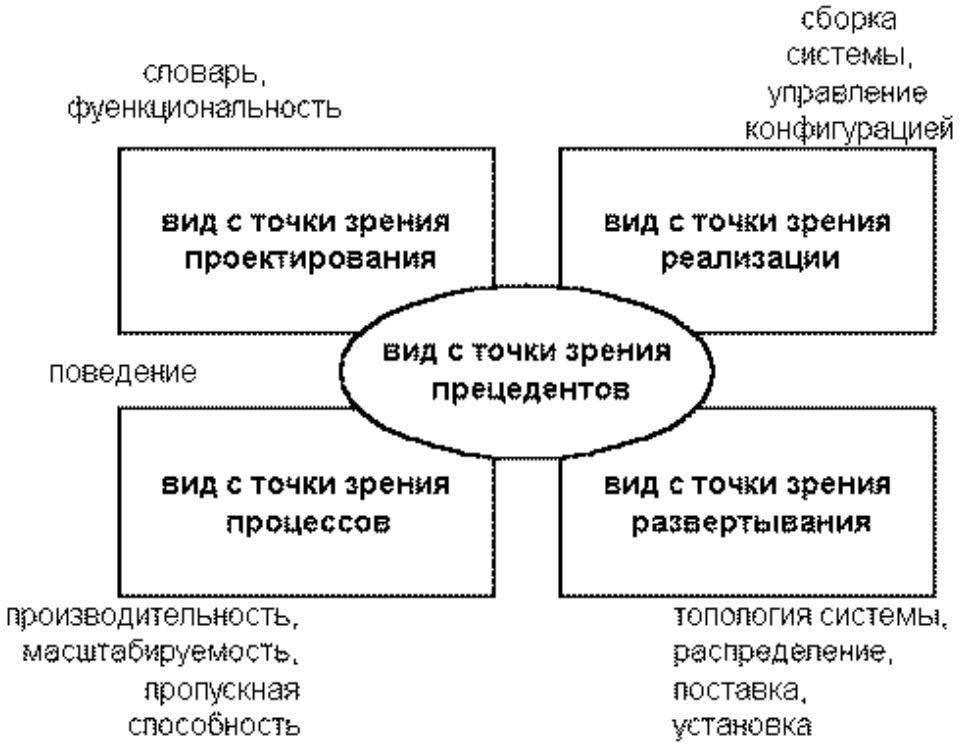


Рис. 3.4. Моделирование системной архитектуры

Вид с точки зрения прецедентов (Use case view) охватывает прецеденты, которые описывают поведение системы, наблюдаемое конечными пользователями, аналитиками и тестировщиками. Этот вид специфицирует не истинную организацию программной системы, а те движущие силы, от которых зависит формирование системной архитектуры. В языке UML статические аспекты этого вида передаются диаграммами прецедентов, а динамические - диаграммами взаимодействия, состояний и действий.

Вид с точки зрения проектирования (Design view) охватывает классы, интерфейсы и кооперации, формирующие словарь задачи и ее решения. Этот вид поддерживает прежде всего функциональные требования, предъявляемые к системе, то есть те услуги, которые она должна предоставлять конечным пользователям. С помощью языка UML статические аспекты этого вида можно передавать диаграммами классов и объектов, а динамические - диаграммами взаимодействия, состояний и действий.

Вид с точки зрения процессов (Process view) охватывает нити и процессы, формирующие механизмы параллелизма и синхронизации в системе. Этот вид описывает главным образом производительность, масштабируемость и пропускную способность системы. В UML его статические и динамические аспекты визуализируются теми же диаграммами, что и для вида с точки зрения проектирования, но особое внимание при этом уделяется активным классам, которые представляют соответствующие нити и процессы.

Вид с точки зрения реализации (Implementation view) охватывает компоненты и файлы, используемые для сборки и выпуска конечного программного продукта. Этот вид предназначен в первую очередь для управления конфигурацией версий системы, составляемых из независимых (до некоторой степени) компонентов и файлов, которые могут по-разному объединяться между собой. В языке UML статические аспекты этого вида передают с помощью диаграмм компонентов, а динамические - с помощью диаграмм взаимодействия, состояний и действий.

Вид с точки зрения развертывания (Deployment view) охватывает узлы, формирующие топологию аппаратных средств системы, на которой она выполняется. В первую очередь он связан с распределением, поставкой и установкой частей, составляющих физическую систему. Его статические аспекты описываются диаграммами развертывания, а динамические - диаграммами взаимодействия, состояний и действий.

Каждый из перечисленных видов может считаться вполне самостоятельным, так что лица, имеющие отношение к разработке системы, могут сосредоточиться на изучении только тех аспектов архитектуры, которые непосредственно их касаются. Но нельзя забывать о том, что эти виды взаимодействуют друг с другом. Например, узлы вида с точки зрения развертывания содержат компоненты, описанные для вида с точки зрения реализации, а те, в свою очередь, представляют собой физическое воплощение классов, интерфейсов, кооперации и активных классов из видов с точки зрения проектирования и процессов. UML позволяет отобразить каждый из пяти перечисленных видов и их взаимодействия.

3.3 Жизненный цикл разработки ПО

Используя UML, вы практически не зависите от организации процесса разработки; он не привязан к какому-либо конкретному циклу изготовления программного продукта. Тем не менее, если вы хотите извлечь из этого языка наибольшую пользу, лучше всего применять процесс, который:

- управляется прецедентами использования;
- основан на архитектуре;
- является итеративным и инкрементным.

Управляемость прецедентами использования означает, что прецеденты должны быть основным артефактом, на основании которого устанавливается желаемое поведение системы, проверяется и подтверждается правильность выбранной системной архитектуры, производится тестирование и осуществляется взаимодействие между участниками проекта.

Процесс называют ***основанным на архитектуре*** (Architecture-centric), когда системная архитектура является решающим фактором при разработке концепций, конструировании, управлении и развитии создаваемой системы.

Итеративным (Iterative) называется процесс, который предполагает управление потоком исполняемых версий системы. ***Инкрементный*** (Incremental) процесс подразумевает постоянное развитие системной архитектуры при выпуске новых версий, причем каждая следующая версия усовершенствована в сравнении с предыдущей. Процесс, являющийся одновременно итеративным и инкрементным, называется ***управляемым рисками*** (Risk-driven), поскольку при этом в каждой новой версии серьезное внимание уделяется выявлению факторов, представляющих наибольший риск для успешного завершения проекта, и сведению их до минимума.

Управляемый прецедентами, основанный на архитектуре, итеративный и инкрементный процесс может быть разбит на фазы. ***Фазами*** (Phase) называют промежутки времени между двумя опорными точками процесса, в которых выполнены хорошо определенные цели, завершено создание артефактов и принимается решение, стоит ли переходить к следующей фазе. Как видно из рис. 3.5, жизненный цикл процесса разработки программного обеспечения состоит из четырех фаз: начало (Inception), исследование (Elaboration), построение

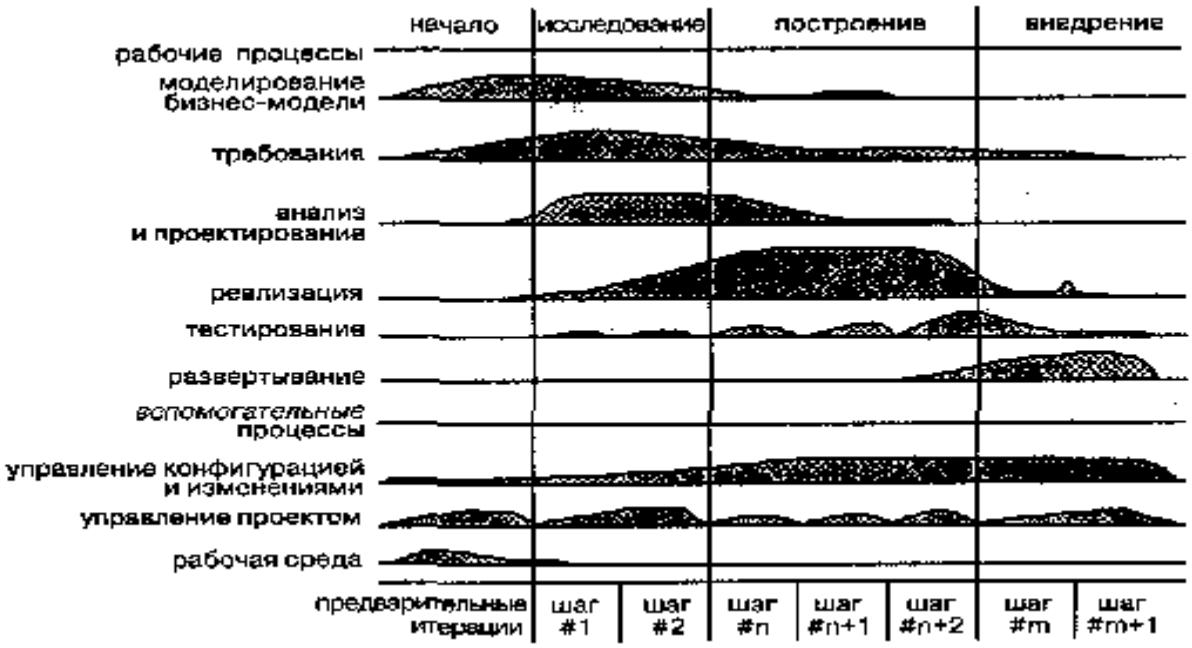


Рис. 3.5. Жизненный цикл процесса разработки программного обеспечения

(Construction) и внедрение (Transition). На этой диаграмме для каждой фазы показаны соответствующие производственные процессы.

Начало - первая стадия процесса, на протяжении которой изначальная идея получает достаточное обоснование, чтобы можно было принять решение о переходе к фазе исследования.

Исследование - это вторая фаза процесса; на этом этапе определяется видение продукта и его архитектура. Основное внимание уделяется конкретизации требований к системе и расстановке приоритетов. Сами требования могут выражаться как в виде общих утверждений, так и в виде четких критериев оценки, каждый из которых определяет функциональное или нефункциональное поведение системы и закладывает основы для тестирования.

Построение является третьей фазой процесса. Исполняемый архитектурный прототип приобретает форму, в которой он может быть представлен пользователям. На этом этапе требования к системе, и в особенности критерии оценки, подвергаются пересмотру в соответствии с изменяющимися потребностями, а для уменьшения риска выделяются необходимые ресурсы.

Внедрение - четвертая стадия процесса разработки программного обеспечения, в ходе которой готовая система передается в руки пользователей. Но разработка на этом, как правило, не заканчивается - ведь даже на протяжении данной фазы система непрерывно совершенствуется, устраняются ошибки и добавляются не вошедшие в ранние версии функциональные возможности.

Во всех четырех фазах присутствует элемент, характерный для описанного способа организации разработки программного обеспечения, - итерация. Итерацией называется четко определенная последовательность действий с ясно сформулированным планом и критериями оценки, которая приводит к появлению новой версии для внешнего или внутреннего использования. Это означает, что жизненный цикл процесса разработки представляет собой непрерывный поток исполняемых версий, реализующих архитектуру системы.

4. Диаграммы

4.1. Диаграммы вариантов использования

Диаграммы вариантов использования играют основную роль в моделировании поведения системы, подсистемы или класса.

Диаграммой вариантов использования называется диаграмма, на которой показана совокупность вариантов использования и актеров, а также отношения между ними.

Вариант использования (use case) – это описание функциональности системы на «высоком уровне». Действующее лицо (actor) – это всё, что взаимодействует с системой.

Варианты использования и действующие лица определяют сферы применения создаваемой системы. При этом варианты использования описывают все то, что происходит внутри системы, а действующие лица – то, что происходит снаружи.

Следует отметить, что нередко название диаграмм *use case* переводится на русский язык как диаграммы прецедентов, а сами варианты использования называют прецедентами.

Вариант использования иллюстрирует, как можно использовать систему.

Например, пусть разрабатывается система Контроля исполнения поручений (КИП), которая предоставляет пользователю базовый набор функциональных возможностей:

- принять новое поручение на контроль;
- разослать напоминания;
- снять поручение с контроля.

Каждая такая функциональная возможность – это самостоятельные варианты использования (или прецеденты).

На языке UML варианты использования изображают так, как показано на рис. 4.1.

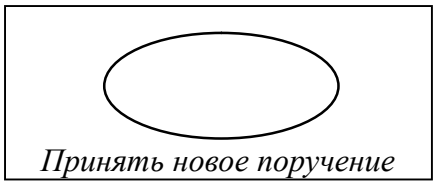


Рис. 4.1. Вариант использования

Названия вариантов использования должны быть деловыми, а не техническими терминами. Их обычно называют глаголами или глагольными фразами, описывая при этом, что пользователь видит в качестве конечного результата процесса.

Преимущество вариантов использования заключается в том, что можно отделить реализацию системы от описания ее принципиальных основ. Они позволяют заострить внимание на наиболее важных вещах – удовлетворении потребностей и ожиданий заказчиков – без необходимости углубления в детали реализации. Взглянув на вариант использования, пользователь сможет понять, что будет делать система, обсудить сферу ее применения в самом начале работы над проектом.

Для того чтобы обнаружить варианты использования, необходимо:

– внимательно прочитать документацию заказчика;

– рассмотреть области использования системы на высоком уровне и документы концептуального характера;

– учесть мнение каждого из заинтересованных лиц проекта.

Следует подумать, чего ожидают от готового продукта. Каждому заинтересованному лицу можно задать следующие вопросы:

– Что он хочет делать с системой?

– Будет ли он с ее помощью работать с информацией (вводить, получать, обновлять, удалять)?

– Нужно ли будет информировать систему о каких-либо внешних событиях?

– Должна ли система в свою очередь информировать пользователя о каких-либо изменениях или событиях?

Для того чтобы убедиться, что вы обнаружили все варианты использования, необходимо ответить на вопросы:

– Присутствует ли каждое функциональное требование хотя бы в одном варианте использования? Если требование не нашло отражение в варианте использования, оно не будет реализовано.

– Учтено ли, как с системой будет работать каждое заинтересованное лицо?

– Какую информацию будет передавать системе каждое заинтересованное лицо?

– Какую информацию будет получать от системы каждое заинтересованное лицо?

– Учтены ли проблемы, связанные с эксплуатацией? Кто-то должен будет запускать готовую систему и выключать ее.

– Учтены ли все внешние системы, с которыми будет взаимодействовать данная?

– Какой информацией каждая внешняя система будет обмениваться данной

Действующее лицо (actor) – это то, что взаимодействует с создаваемой системой. Если варианты использования описывают все, что происходит внутри области действия системы, действующие лица определяют все, что находится вне системы.

На языке UML действующие лица называют актерами и обозначают следующим образом (рис. 4.2).

Рис.4.2. Действующее лицо

Первый тип действующих лиц – это физические личности. Они наиболее типичны и имеются практически в каждой системе. Например, в системе «Стол заказов» к этому типу будут относиться оператор и клиент.

Вторым типом действующих лиц является другая система. Нужно иметь в виду, что, когда делают систему действующим лицом, то предполагают, что она не будет изменяться вообще.

Следует помнить, что действующие лица находятся вне сферы действия того, что разрабатывается и, следовательно, не подлежат контролю со стороны разработчика.

На диаграмме вариантов использования могут использоваться несколько видов связей.

Связь коммуникации (communicates relationship) – это связь между вариантом использования и актером. Изображается в виде стрелки, направление которой показывает, кто инициирует коммуникацию. Каждый вариант использования должен быть инициирован действующим лицом; исключения составляют лишь варианты использования в связях использования и расширения.

Связь использования (uses relationship) позволяет одному варианту использования задействовать функциональность другого. С помощью таких связей обычно моделируют многократно применяемую функциональность, используемую двумя или более другими вариантами использования.

Например, пусть в системе КИП варианты использования «Распечатать поручение или группу поручений» и «Редактировать поручение» должны будут выделять цветом поручения, у которых подходит срок исполнения. Для этого заведём один вариант использования «Выделить поручения». Когда какому-нибудь варианту использования потребуется выполнить эти действия, он сможет воспользоваться функциональностью созданного варианта использования «Выделить поручения».

Связь использования изображается в UML с помощью стрелок и слова «использование» (uses), как показано на рис. 4.3.

Связь расширения (extends relationship) позволяет варианту использования только при необходимости применять функциональные возможности, предоставляемые другим вариантом использования. Она напоминает связь использования: в обоих типах отношений некоторая общая функциональность выделяется в отдельный Вариант Использования.

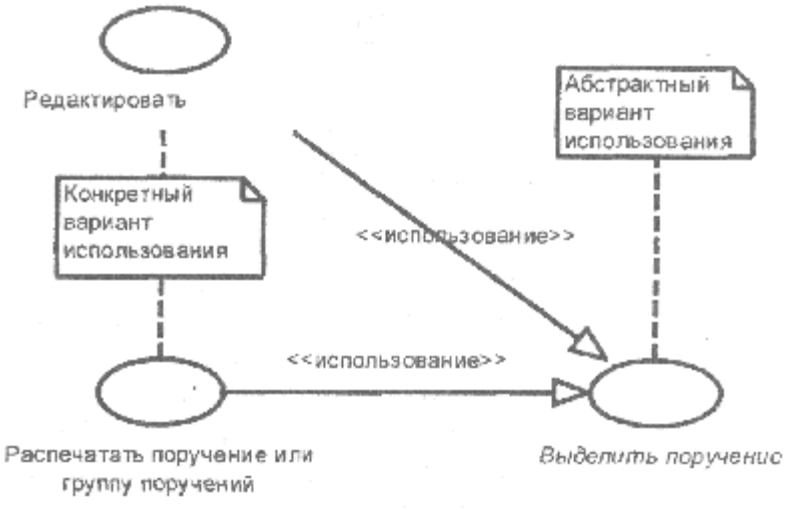


Рис.4.3. Связь использования

Связь использования предполагает, что один вариант использования всегда применяет функциональные возможности другого. В свою очередь, связи расширения (extends relationships) позволяют варианту использования только при необходимости применять функциональные возможности другого.

На языке UML связи расширения изображают в виде стрелки с надписью «расширение» (extends), как показано на рис. 4.4. .

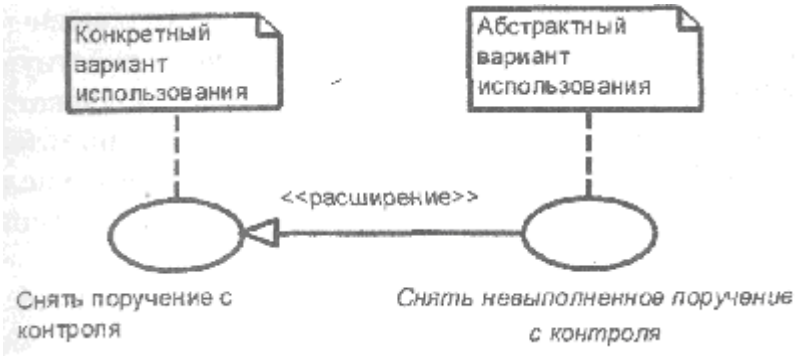


Рис.4.4. Связь расширения

На примере показано, что вариант использования «Снять поручение с контроля» иногда применяет функциональные возможности, предоставляемые вариантом использования «Снять невыполненное поручение с контроля».

Предоставляющий дополнительные возможности вариант использования «Снять невыполненное поручение с контроля» является абстрактным. Вариант использования «Снять поручение с контроля» – конкретный.

Разрабатывая диаграммы Вариантов Использования, следует придерживаться некоторых правил.

– Не нужно моделировать связи между действующими лицами. Действующие лица по определению находятся вне сферы действия системы. Это означает, что связи между ними также не относятся к компетенции системы.

– Не следует соединять стрелкой непосредственно два варианта использования (кроме случаев связей использования и расширения). На данных диаграммах описываются только сами варианты использования, доступные системе, а не порядок их выполнения. Для отображения порядка выполнения вариантов использования применяются диаграммы Деятельности.

– Каждый вариант использования должен быть инициирован действующим лицом, т.е. стрелка всегда должна начинаться на действующем лице и заканчивающаяся на варианте использования (исключения – связи использования и расширения).

– О базе данных следует думать как о слое, находящемся под диаграммой. С помощью одного варианта использования можно вводить данные в базу, а получать их с помощью другого. Для изображения потока информации не нужно рисовать стрелки от одного варианта использования к другому.

4.2. Диаграммы взаимодействия

Диаграммы Взаимодействия моделируют взаимодействия между объектами системы.

На диаграмме Взаимодействия (Interaction) отображают один из процессов обработки информации в варианте использования. Если у варианта использования имеется несколько альтернативных потоков, то для данного варианта использования нужно создать несколько диаграмм Взаимодействия.

Существуют два типа диаграмм Взаимодействия: диаграммы Последовательности (Sequence) и Кооперативные диаграммы (Collaboration). Обе диаграммы отображают события, участвующие в процессе обработки информации варианта использования, и сообщения, которыми обмениваются объекты.

События на диаграмме Последовательности упорядочены по времени. Они заостряют внимание на управлении. Графически такая диаграмма представляет собой таблицу, объекты в которой располагаются вдоль оси X, а сообщения в порядке возрастания времени – вдоль оси Y. Пример такой диаграммы представлен на рис. 4.5.а.

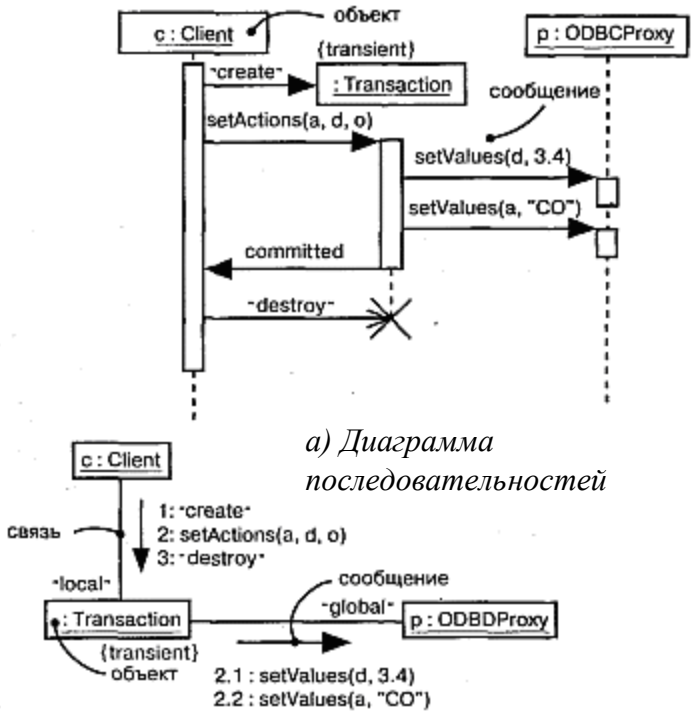
Кооперативная диаграмма организована вокруг самих объектов. Здесь отображается поток данных. Графически такая диаграмма представляет собой граф из вершин и ребер. Пример такой диаграммы представлен на рис. 4.5.б.

Диаграммы Взаимодействия визуализируют практически те же детали, что уже были описаны в потоке событий, однако представляют их в форме, более удобной для разработчика. Главное здесь – объекты, которые должны быть созданы для реализации функциональных возможностей, заложенных в вариант использования. На диаграммах Последовательности и Кооперативных диаграммах могут быть показаны объекты, классы или то и другое вместе.

С помощью диаграмм Взаимодействия проектировщики и разработчики системы могут определить классы, которые нужно создать, связи между ними, а также операции и ответственности каждого класса.

Диаграммы Последовательности полезны для того, кто хочет понять логическую последовательность событий в сценарии. Хотя информация о последовательности входит и в Кооперативные диаграммы, она лучше воспринимается на диаграмме Последовательности.

Кооперативные диаграммы полезны в тех случаях, когда нужно оценить последствия сделанных изменений. Она показывает, какие объекты взаимодействуют друг с другом.



а) Диаграмма последовательностей

б) Диаграмма кооперации

Рис. 4.5. Диаграммы Взаимодействия

При внесении изменений в объект вы сразу поймете, на какие другие объекты это повлияет.

Диаграммы Взаимодействия содержат объекты и сообщения.

Для объектов можно использовать имена как объектов, так и классов, или того и другого.

С помощью сообщения один объект или класс запрашивает у другого выполнения какой-то конкретной функции.

Создавая диаграммы Взаимодействия, следует помнить, что таким образом назначаются объектам определенные ответственности. При помещении на диаграмму сообщения, назначается ответственность получающему его объекту. Необходимо следить за тем, чтобы объекты и их ответственности соответствовали друг другу. Например, в большинстве приложений экраны и формы не должны реализовывать никаких бизнес-процессов. С их помощью следует только вводить и просматривать информацию. Тогда внесение изменений в бизнес-логику не затронет интерфейс. С другой стороны, при изменении формата одного или двух экранов, хотя бы бизнес-логика останется неизменной.

Диаграмма Последовательности – это упорядоченная по времени диаграмма Взаимодействия, читать которою следует сверху вниз.

Как упоминалось раньше, у каждого варианта использования имеется большое количество альтернативных потоков. Каждая диаграмма Последовательности описывает один из потоков варианта использования.

Участвующие в потоке объекты нарисованы в прямоугольниках в верхней части диаграммы (рис. 2). У каждого объекта имеется линия жизни (lifeline), изображаемая в виде вертикальной штриховой линии под объектом. Обычно объект существует на протяжении всего взаимодействия и их линии жизни прорисованы донизу. Объекты могут также уничтожаться и во время взаимодействий: в таком случае их линии жизни заканчиваются получением сообщения со стереотипом

«destroy», а в качестве визуального образа используется большая буква X, обозначающая конец жизни объекта (рис. 4.5.а).

Сообщение (message) – это связь между объектами, в которой один из них (клиент) требует от другого (сервера) выполнения каких-то действий. Сообщения показывают, что один объект вызывает функцию другого. Их изображают в виде стрелки, которая проводится между линиями жизни двух объектов или линии жизни объекта к самой себе (рефлексивное сообщение). Например, на диаграмме последовательности для варианта использования «Принять новое поручение» (см. рис. 2) объект «Поручение» обращается сам к себе, чтобы присвоить вновь введённому поручению идентификационный номер.

Сообщения располагают хронологическом порядке сверху вниз.

Также на диаграммах Последовательности изображается фокус управления – вытянутый прямоугольник, показывающий промежуток времени, в течение которого объект выполняет какое-либо действие (рис. 4.5.а). Верхняя грань прямоугольника выравнивается по временной оси с моментом начала действия, нижняя – с моментом его завершения. Вложенность фокуса управления, вызванную рекурсией, можно показать, расположив другой фокус управления чуть правее своего родителя.

На любой диаграмме Последовательности должен быть объект–действующее лицо. Он является внешним стимулом, дающим системе команду на выполнение какой-то функции. Объекты–действующие лица на диаграмме Взаимодействия содержат действующих лиц, взаимодействующих с вариантом использования в диаграмме Вариантов Использования. Подобно диаграммам Последовательности, Кооперативные диаграммы (Collaboration) отображают поток событий в конкретном сценарии варианта использования. Кооперативные диаграммы больше внимания заостряют на связях между объектами.

В отличие от диаграмм Последовательности, **на кооперативных диаграммах** объекты располагаются произвольным образом, отсутствуют линии жизни.

Перед тем как поместить сообщение на Кооперативную диаграмму, необходимо установить путь коммуникации (связь) между объектами (линия, соединяющая объекты).

Для обозначения временной последовательности перед сообщением можно поставить номер (нумерация начинается с единицы), который должен постепенно возрастать для каждого нового сообщения (2, 3 и т.д.). Для обозначения вложенности используется десятичная нотация Дьюи (1 – первое сообщение; 1.1 – первое сообщение, вложенное в сообщение 1; 1.2 – второе сообщение, вложенное в сообщение 1 и т.д.) (рис. 4.5.б). Уровень вложенности не ограничен.

На Кооперативных диаграммах, также как и на диаграммах Последовательности, должны присутствовать действующие лица, дающие команду на выполнение какой-либо функции.

Кооперативные диаграммы должны иметь такой же состав объектов, как и соответствующая диаграмма

4.3. Диаграмма деятельности

При моделировании поведения проектируемой или анализируемой системы возникает необходимость не только представить производимые изменения ее состояний, но и детализировать особенности алгоритмической и логической реализации выполняемых системой операций. Для этой цели могли использоваться блок-схемы, которые акцентируют внимание на последовательности выполнения определенных действий или элементарных операций, которые в совокупности приводят к получению желаемого результата.

Диаграммы деятельности в языке UML используются для моделирования процесса выполнения операций. Каждое состояние на диаграмме деятельности соответствует выполнению некоторой элементарной операции, а переход в следующее состояние срабатывает только при завершении этой операции в предыдущем состоянии (рис. 4.6).

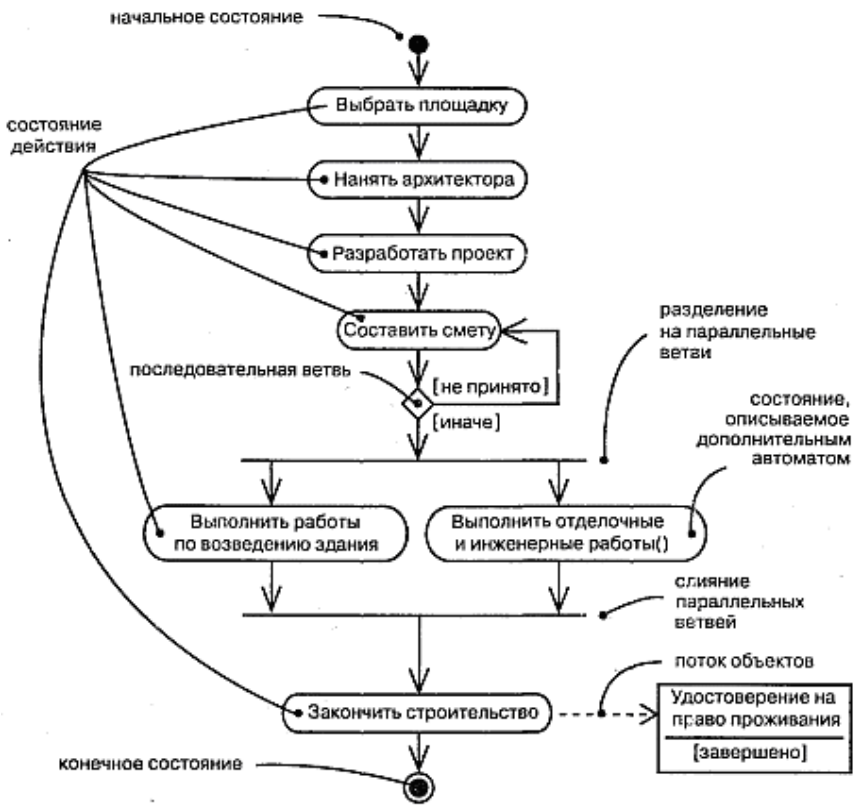


Рис. 4.6. Диаграммы деятельности

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

На диаграмме деятельности отображается логика или последовательность перехода от одной деятельности к другой, при этом внимание фиксируется на результате деятельности. Сам же результат может привести к изменению состояния системы или возвращению некоторого значения.

Диаграммы Деятельности целесообразно использовать для описания сложных базовых сценариев Вариантов использования, а также связи между сценариями, где прослеживается параллельность выполняемых операций.

Действия на диаграмме деятельности могут производиться над теми или иными объектами. Эти объекты либо инициируют выполнение действий, либо определяют результат этих действий. При этом действия специфицируют вызовы, которые передаются от одного объекта графа деятельности другому. Поскольку в таком ракурсе объекты играют определенную роль в понимании процесса деятельности, иногда возникает необходимость явно указать их на диаграмме деятельности.

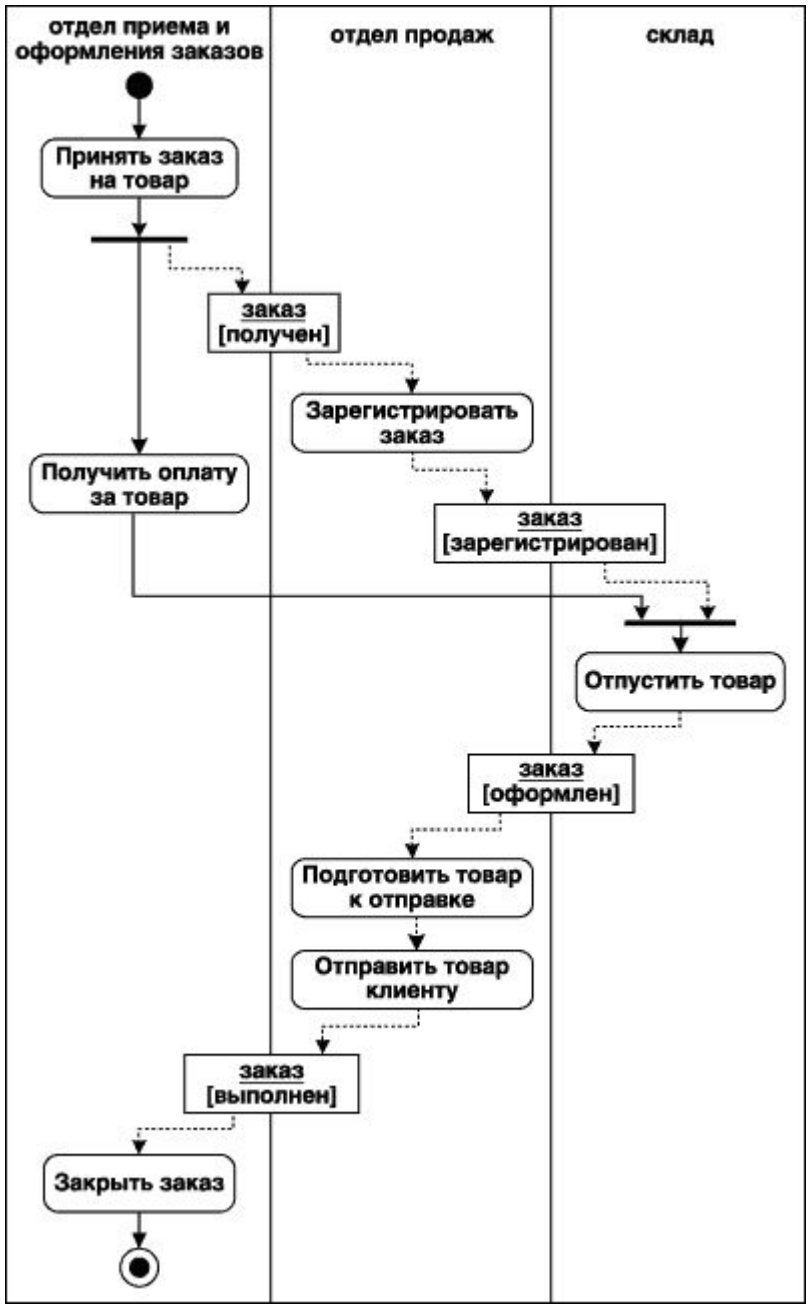


Рис.4.7. Действия на диаграмме деятельности

Базовым графическим представлением объекта в нотации языка UML является прямоугольник класса, с тем отличием, что имя объекта подчеркивается. На диаграммах деятельности после имени может указываться характеристика состояния объекта в прямых скобках. Такие прямоугольники объектов присоединяются к состояниям действия отношением зависимости пунктирной линией со стрелкой. Соответствующая зависимость определяет состояние конкретного объекта после выполнения предшествующего действия (рис.4.7).

На диаграмме деятельности с дорожками расположение объекта может иметь дополнительный смысл:

– если объект расположен на границе двух дорожек, то это может означать, что переход к следующему состоянию действия в соседней дорожке ассоциирован с нахождением документа в некотором состоянии;

– если объект расположен внутри дорожки, то и состояние этого объекта целиком определяется действиями данной дорожки.

Полная модель системы может содержать одну или несколько диаграмм деятельности, каждая из которых описывает последовательность реализации либо наиболее важных вариантов использования (типичный ход событий и все исключения), либо нетривиальных операций классов.

4.4 Диаграмма классов

Одно из центральным мест в объектно-ориентированном проектировании занимает разработка логической модели системы в виде диаграммы Классов.

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

Обычно для описания системы создают несколько диаграмм Классов. На одних показывают некоторое подмножество классов отношения между классами подмножества. На других отображают подмножество, но вместе с атрибутами и операциями классов. Третьи соответствуют только пакетам классов и отношениям

между ними. Для представления полной картины системы можно разработать столько диаграмм Классов, сколько требуется.

С помощью диаграмм Классов разработчики могут видеть и планировать структуру системы еще до фактического написания кода, благодаря чему с самого начала могут понять, хорошо ли спроектирована система

Класс (class) – это абстрактное описание множества однородных объектов, имеющих одинаковые атрибуты, операции и отношения с объектами других классов. Класс – это некоторая сущность, инкапсулирующая данные и поведение.

Можно сказать, что класс – это тип данных. В языке Паскаль он описывается в разделе type вместе с другими структурными типами, такими как массивы, записи и т.д.

Переменная типа класс называется объектом. Все объекты одного класса имеют одинаковый набор атрибутов, описанный в классе, в которых хранятся конкретные значения, и умеют выполнять одни и те же операции. В языке Паскаль все объекты описываются в разделе var вместе с другими переменными, и для каждого объекта нужно указать тип, т.е. класс, к которому он относиться.

Например, «Студент вуза» – это класс. Он описывает набор атрибутов и операций, свойственные для всех студентов вуза. Это могут быть атрибуты «Курс», «Специальность» и «Вуз», а также операции «Сдать зачет» и «Сдать экзамен».

Объектами будут, например:

– «Студент Иванов», Курс – 3, Специальность – Геология, Вуз – МГУ;
– «Студент Петров», Курс – 1, Специальность – Математика, Вуз – ТГУ;
– «Студент Рожков», Курс – 5, Специальность – ИСТ, Вуз – ТГНГУ.

Как видно из примера, и Иванов, и Петров, и Рожков имеют атрибуты Курс, Специальность и Вуз (эти и только эти), а вот значения в этих полях у каждого объекта свои (Иванов – 1-ый курс, Петров – 3-ий, и т.д.).

Кроме того, все трое студентов имеют общее поведение – они выполняют операции «Сдать зачет» и «Сдать экзамен».

Еще один пример: класс для работы с данными о поручениях Task. Он содержит такие данные, как:

- идентификационный номер поручения (taskNumber);
- его содержание (taskText);
- дата исполнения поручения (executionDate);
- исходящий номер (sourceNumber);
- исходящая дата (sourceDate);
- дата выдачи поручения (taskDate).

Класс Task имеет также свое поведение:

- создать поручение (create);
- заполнить реквизиты (setInfo);
- извлечь информацию (getInfo);
- сделать пометку о снятии поручения с контроля (closeTask).

Графически класс в нотации языка UML изображается в виде прямоугольника, который дополнительно может быть разделен горизонтальными линиями на разделы или секции (рис. 4.8).

Рис.4.8. Графическое представление классов

На начальных этапах разработки диаграммы отдельные классы могут обозначаться простым прямоугольником, в котором должно быть указано имя класса (Рис.4.8 а).

По мере проработки отдельных компонентов диаграммы описание классов дополняется атрибутами (Рис.4.8 б) и операциями (Рис.4.8 в). Четвертая секция (Рис.4.8 г) не обязательна и служит для размещения дополнительной информации справочного характера, например, об исключениях или ограничениях класса, сведения о разработчике или языке реализации.

Окончательный вариант диаграммы обычно содержит наиболее полное описание классов, которые состоят из трех или четырех секций.

Следует отметить, что даже если секции атрибутов и операций пусты, в обозначении класса они должна быть выделены горизонтальной линией, с тем чтобы отличить класс от других элементов языка UML.

На рис. 4.9 представлены некоторые примеры графического представления классов на диаграмме Классов.

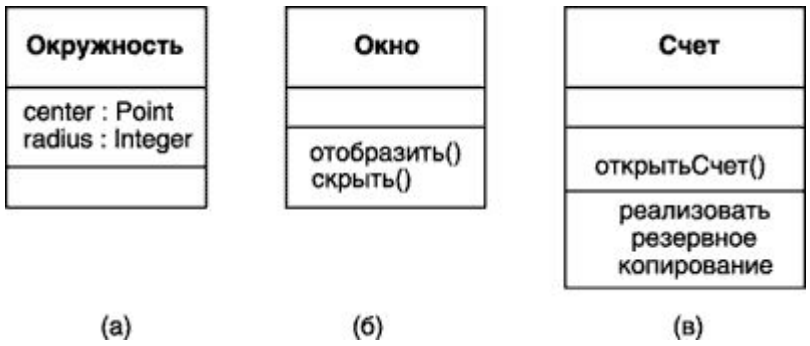


Рис.4.9. Графическое представление классов (примеры)

Выявление классов можно начать с изучения потока событий сценария. Имена существительные в описании этого потока дадут понять, что может являться классом. В общем случае существительные могут соответствовать таким элементам, как:

- действующее лицо;
- класс;
- атрибут класса;
- выражение, не являющееся действующим лицом, классом или атрибутом.

Другим способом является анализ диаграмм Последовательности и Кооперативных диаграмм. Для похожих объектов на этих диаграммах следует создать один класс, который будет шаблоном для этих объектов.

Следует также отметить, что в потоке событий и на диаграммах Взаимодействия можно не найти некоторые классы.

Имя класса должно быть уникальным в пределах пакета, который может содержать одну или несколько диаграмм Классов. Оно записывается по центру самой верхней секции.

Большинство организаций имеет собственные соглашения по именованию классов. В общем случае используются существительные в единственном числе. Обычно имена классов не содержат пробелов. Используемый для именования классов регистр символов определяется обычно организацией (обычно имена классов записывают с заглавной буквы). Важно только, чтобы принятый подход применялся ко всем классам модели.

Атрибут – это фрагмент информации, связанный с классом. Все атрибуты записываются во второй сверху секции в графическом представлении класса.

Общий формат записи отдельного атрибута класса следующий:

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

Нередко атрибуты содержатся внутри класса и скрыты от других классов. При указывается, какие классы имеют право читать и изменять атрибуты. Это свойство называется видимостью атрибута (visibility).

Допустимы четыре значения квантора видимости:

1) Символ "+" – Public (общий). Атрибут виден всем остальным классам. Любой класс может просмотреть или изменить значение атрибута.

2) Символ "-" – Private (закрытый). Атрибут не виден никаким другим классам. Доступ к таким атрибутам обычно делается с помощью общих операций.

3) Символ "#" – Protected (защищенный). Атрибут доступен только самому классу и его потомкам.

4) Символ "~" – Package or Implementation (пакетный). Атрибут является общим, но только в пределах своего пакета, т.е. он может быть изменен из класса, находящегося в том же пакете.

Если квантор видимости не указан, то, в отличие от большинства языков программирования, считается, что видимость атрибута не определена.

В общем случае атрибуты рекомендуется делать закрытыми или защищенными. Это позволяет лучше контролировать сам атрибут и код, т.к. удается избежать ситуации, когда значение атрибута может изменяться всеми классами системы. Вместо этого логика изменения атрибута будет заключена в том же классе, что и сам атрибут (например, атрибут «цвет формы»).

Стоит отметить, что в различных средах могут поддерживаться и другие нотации обозначения видимости.

Имя атрибута представляет собой строку текста, которая обычно начинается с маленькой буквы и не содержит пробелов. Имя атрибута используется в качестве его идентификатора и должно быть уникальным в пределах классах. Единственный обязательный элемент.

Кратность атрибута характеризует общее количество конкретных атрибутов данного типа, входящих в состав отдельного класса. В общем случае кратность записывается в форме [нижняя граница .. верхняя граница]. В качестве верхней границы может использоваться символ «*». Также здесь может быть указано единственное число – просто кратность. Если кратность не указана, то по умолчанию она принимается как 1.

Одной из главных характеристик атрибута является его тип данных. Он специфичен для используемого языка программирования.

Исходное значение служит для задания начального значения соответствующего атрибута в момент создания отдельного экземпляра класса.

Строка-свойство служит для указания дополнительных свойств атрибута (если имеется знак равенства). Фигурные скобки обозначают фиксированное значение соответствующего атрибута, которое должны принимать все создаваемые экземпляры класса без исключения. Это значение не может быть переопределено в последующем. Отсутствие строки-свойства по умолчанию трактуется так, что значение соответствующего атрибута может быть изменено в программе.

При добавлении атрибута к классу каждый экземпляр класса получит свою собственную копию этого атрибута (значение «Курс» у студентов). Однако существуют и общие для всех объектов атрибуты.

Статичный атрибут (static) – это такой атрибут, который используется всеми экземплярами класса, т.е. его значение у всех создаваемых объектов одинаковое.

Такой атрибут помечают символом "$" или подчеркиванием строки атрибута.

Существует также понятие производного атрибута. Производным (derived) называется атрибут, созданный из одного или нескольких других атрибутов. Например, если у класса Студент существует атрибут «Дата рождения», то можно создать производный атрибут «Возраст», который будет вычисляться как текущая дата минус дата рождения.

В UML производные атрибуты помечают символом «/».

Атрибуты можно найти:

– в описании варианта использования. Здесь одни имена существительные в потоке событий будут классами или объектами, другие – действующими лицами и, наконец, последняя группа – атрибутами;

– в документации, описывающей требования к системе (любой элемент собираемой информации может быть атрибутом класса);

– и т.д

Операцией называется связанное с классом поведение.

Операция – 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.

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: Public, Private, Protected или Package or Implementation (пакетная).

Имя операции представляет собой строку текста, которая используется в качестве идентификатора операции и поэтому должна быть уникальной в пределах класса. Имя операции – единственный обязательный элемент, обычно начинаться с маленькой буквы и записываться без пробелов.

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

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

Направление параметра – это одно из ключевых слов in, out или inout со значением in по умолчанию, в случае если вид параметра не указывается.

Имя параметра – это идентификатор соответствующего формального параметра, при записи которого следуют правилам задания имен атрибутов.

Выражение типа определяет тип данных для значений формального параметра. Наконец, значение по умолчанию в общем случае представляет собой некоторое конкретное значение для этого формального параметра.

Список параметров не обязателен, однако круглые скобки должны присутствовать после имени операции всегда.

Выражение типа возвращаемого значения указывает на тип данных значения, которое возвращается объектом после выполнения соответствующей операции. Может быть опущено, если операция не возвращает никакого значения.

Строка-свойство служит для указания значений свойств, которые могут быть применены к данной операции. Может отсутствовать.

При выявлении операций следует рассмотреть четыре различных типа операций:

1) Операции реализации – реализуют некоторую бизнес-функциональность. Можно найти, исследуя диаграммы Взаимодействия. Диаграммы этого типа фокусируются на бизнес-функциональности, и каждое сообщение диаграммы скорее всего, можно соотнести с операцией реализации.

2) Операции управления (manager operations) управляют созданием и разрушением объектов. В эту категорию попадают конструкторы и деструкторы классов.

3) Операции доступа. Атрибуты обычно бывают закрытыми или защищенными. Тем не менее, другие классы иногда должны просматривать или изменять их значения. Для этого предназначены операции доступа (access operations).

4) Вспомогательные (helper operations) операции, которые необходимы для выполнения ответственностей, но о которых другие классы не должны ничего знать. Это закрытые и защищенные операции класса

4.5 Диаграммы состояний

Элементы на диаграмме Состояний отображают жизненный цикл одного объекта начиная с момента его создания и заканчивая разрушением.

Главное назначение диаграммы состояний – описать возможные последовательности состояний и переходов, которые в совокупности характеризуют поведение моделируемой системы в течение всего ее жизненного цикла.

Диаграммы состояний чаще всего используются для описания поведения отдельных систем и подсистем. Также они могут быть применены для спецификации функциональности экземпляров отдельных классов, т.е. для моделирования всех возможных изменений состояний конкретных объектов. Как правило, диаграммы Состояний не требуется создавать для каждого класса.

Сложные классы имеют много различных состояний (рис. 1 – пример диаграммы Состояний для класса «Заказ», экземпляры которого могут пребывать в трех состояниях). В каждом из состояний объект может себя вести по-разному.

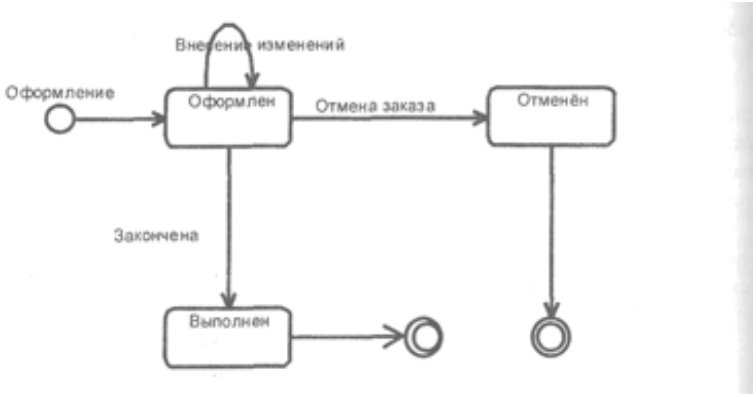


Рис.4.10. Диаграмма Состояний для класса «Заказ»

Для анализа динамики поведения класса необходимо рассмотреть его атрибуты. Экземпляр класса может вести себя по-разному в зависимости от их значений. Полезно исследовать также связи между классами. Можно рассмотреть все связи, множественность которых может принимать нулевое значение. Нули указывают, что данная связь не является обязательной. Экземпляр класса может себя вести по-разному при наличии и отсутствии связи – в этом случае он имеет несколько состояний.

Диаграммы состояний нужны для того, чтобы документировать динамику поведения класса, благодаря чему аналитики и разработчики получат о нем четкое представление. Реализацией заложенной в эти диаграммы логики будут заниматься разработчики. Как и в случае других диаграмм UML, диаграммы Состояний дают команде возможность обсудить и документировать логику перед началом процесса кодирования.

4.6. Диаграммы компонентов

Для создания конкретной физической системы необходимо реализовать все элементы логического представления в конкретные материальные сущности. Для описания таких реальных сущностей предназначено физическое представление модели. В контексте языка UML это означает совокупность связанных физических сущностей, включая программное и аппаратное обеспечение, а также персонал, которые организованы для выполнения специальных задач.

Одной из диаграмм, позволяющей представить физическую модель, является диаграмма компонентов (рис.4.10).

Диаграмма компонентов (component diagram) – диаграмма, представляющая состав и зависимости между компонентами программной системы.

На такой диаграмме можно видеть исходный код и исполняемые компоненты системы. С ее помощью отвечающий за компиляцию и размещение системы персонал выясняет, какие библиотеки кода существуют, и какие исполняемые файлы будут созданы при его компиляции. Разработчики узнают, какие библиотеки кода существуют и каковы связи между ними. Зависимости между компонентами отражают порядок их компиляции.

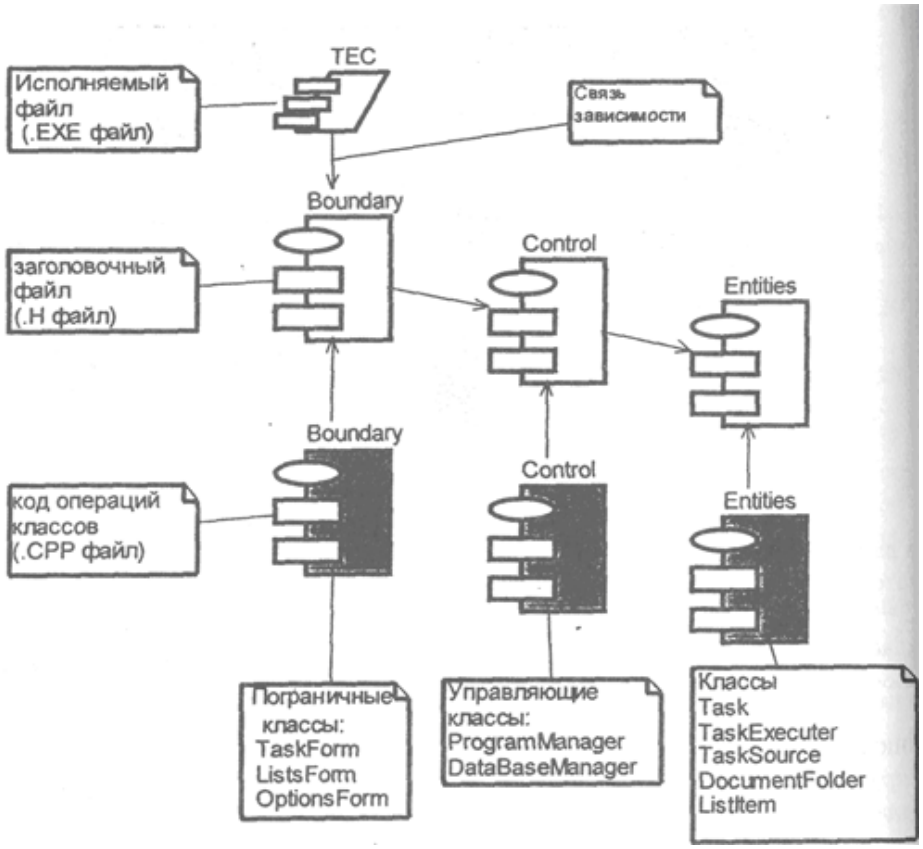


Рис.4.10. Диаграмма Компонентов

В разработке диаграмм компонентов участвуют как системные аналитики и архитекторы, так и программисты. Диаграмма компонентов обеспечивает согласованный переход от логического представления к конкретной реализации проекта в форме программного кода. Одни компоненты могут существовать только на этапе компиляции программного кода, другие – на этапе его исполнения.

4.7. Диаграммы представления размещений

Физическое представление программной системы не может быть полным, если отсутствует информация о том, на какой платформе и на каких вычислительных средствах она реализована. Если создается простая программа, которая может выполняться локально на компьютере пользователя, не используя никаких распределенных устройств и сетевых ресурсов, то необходимости в разработке дополнительных диаграмм нет. Однако при создании корпоративных или распределенных приложений требуется визуализировать сетевую инфраструктуру программной системы.

Диаграмма Размещения применяется для представления общей конфигурации и топологии распределённой программной системы и содержит распределение компонентов по отдельным узлам системы. Кроме того, диаграмма Размещения показывает наличие физических соединений – маршрутов передачи информации между аппаратными устройствами, задействованными в реализации системы.

Диаграмма Размещения предназначена для визуализации элементов и компонентов программы, существующих лишь на этапе её исполнения. При этом представляются только те компоненты, которые являются исполнимыми файлами или динамическими библиотеками. Компоненты, которые не используются на этапе исполнения, на диаграмме Размещения не показываются (например, компоненты с исходными текстами программ).

Диаграмма Размещения (рис.4.11) содержит графические изображения процессоров, устройств, процессов и связей между ними. В отличие от диаграмм логического представления, диаграмма Размещения является целостной для системы в целом, поскольку должна всецело отражать особенности её реализации. Эта диаграмма завершает процесс объектно-ориентированного анализа и проектирования для конкретной программной системы.

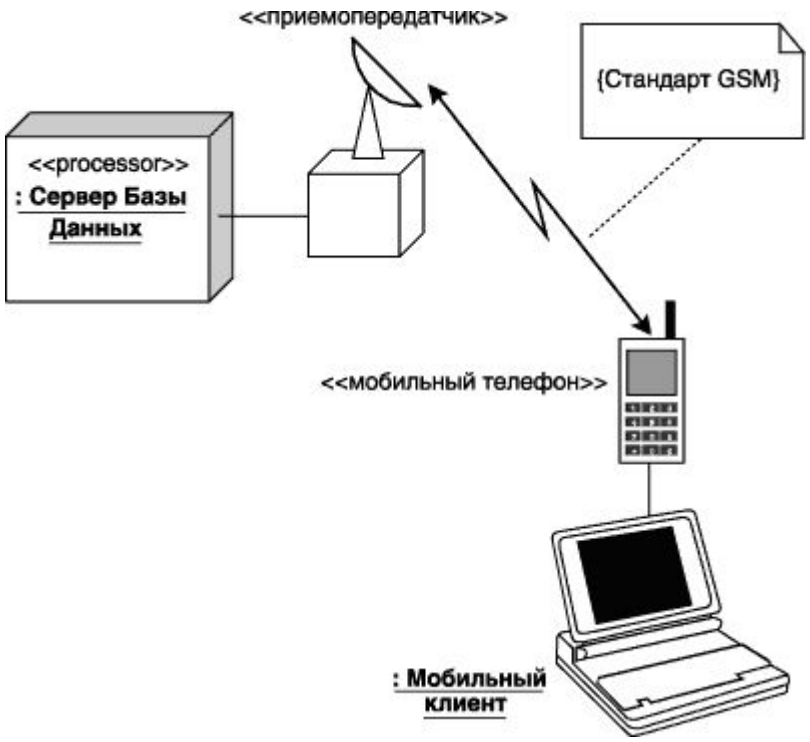


Рис.4.11. Диаграмма Размещения

При разработке диаграммы Размещения преследуются цели:

• определить распределение компонентов системы по её физическим узлам;

• показать физические связи между всеми узлами реализации системы на этапе её исполнения;

• выявить узкие места системы и реконфигурировать её топологию для достижения требуемой производительности.

Диаграмма Размещения разрабатывается совместно системными аналитиками, сетевыми инженерами и системотехниками.

Заключение

Перспективы дальнейшего развития UML связаны со становлением и интенсивным развитием новой парадигмы объектно-ориентированного анализа - компонентной разработки приложений (Component-Based Development - CBD). В этой связи развернута работа над дополнительной спецификацией языка UML применительно к технологиям CORBA и COM+. Речь идет о разработке так называемых профилей, содержащих нотацию всех необходимых элементов для представления в языке UML компонентов соответствующих технологий. При этом интенсивно используется механизм расширения языка UML за счет добавления новых стереотипов, помеченных значений и ограничений.

Язык UML уже сейчас находит широкое применение в качестве неофициального стандарта в процессе разработки программных систем, связанных с такими областями, как моделирование бизнеса, управление требованиями, анализ и проектирование, программирование и тестирование. Применительно к этим процессам в языке UML унифицированы стандартные обозначения основных элементов соответствующих предметных областей.

Разработав модель и специфицировав ее на языке UML, разработчик имеет все основания быть понятым и по достоинству оцененным своими коллегами. При этом могут быть исключены ситуации, когда тот или иной разработчик применяет свою собственную графическую нотацию для представления тех или иных аспектов модели, что практически исключает ее понимание другими специалистами в случае нетривиальности исходной модели.

Не менее важный аспект применения языка UML связан с профессиональной подготовкой соответствующих специалистов. Речь идет о том, что знания различных научных дисциплин характеризуют различные аспекты реального мира. При этом принципы системного анализа позволяют рассматривать те или иные объекты в качестве систем.

Последующая разработка модели системы, направленная на решение определенных проблем, может потребовать привлечения знаний из различных дисциплин. С этой точки зрения язык UML может быть использован не только для унификации представлений этих знаний, но что не менее важно - для их интеграции, направленной на повышение адекватности много-модельных представлений сложных систем.

Литература

1. Вендров А. М. Проектирование программного обеспечения экономических информационных систем: учебник – 2-е изд., перераб. и доп. – М.: Финансы и статистика, 2006. – 544 с.: ил.
2. Леоненков. Самоучитель UML. [Электронный ресурс] http://khpi-iip.mipk.kharkiv.edu/library/case
3. Моделирование корпоративных информационных систем [Электронный ресурс] http://pogo-tech.com
4. "ПРОИНФОТЕХ" [Электронный ресурс] http://www.proinfotech.ru/uml8.htm
5. Язык UML. Руководство пользователя [Электронный ресурс] http://libyr.narod.ru/

61

Караваева Ольга Владимировна

Введение в язык UML

Учебное пособие

Ответственный за выпуск Ростовцев В.С.

Технический редактор Караваева О.В.

Корректор

Подписано в печать «» ноября 2008 г.

Заказ № _____

Усл. печ.л. 1,5 Тираж 50

Текст напечатан с оригинал-макета, предоставленного составителем.

Издательский орган Кировского филиала МФА
610000, Киров, ул. Ленина, 104
Тел./факс 37-18-78

Отпечатано на ризографе.