Содержание

Введение 4
1.Надежное программное средство как продукт технологии программирования. 5
1.1. Программа как формализованное описание процесса обработки данных. 5
1.2. Понятие правильной программы. 6
1.3. Надежность программного средства. 6
1.4. Технология программирования как технология разработки надежных программных средств. 7
2. Источники ошибок в программных средствах 8
2.1. Интеллектуальные возможности человека. 8
2.2. Неправильное преобразование как причина ошибок в программных средствах. 8
3. Общие принципы разработки программных средств 10
3.1. Специфика разработки программных средств. 10
3.2. Жизненный цикл программного средства. 10
3.3. Понятие качества программного средства. 13
3.4. Обеспечение надежности – основа разработки программных средств. 14
3.5. Методы борьбы со сложностью. 15
4. Методология ООАП 15
4.1. Стандарты жизненного цикла 15
4.2. Методология системного анализа и системного моделирования 18
5. История создания языка UML 19
6. Основные понятия языка UML 19
7. Диаграмма вариантов использования (USE CASE DIAGRAM) 24
8. Диаграмма классов (CLASS DIAGRAM) 32
9. Диаграмма компонентов (COMPONENT DIAGRAM) 42
10. Диаграмма развертывания (DEPLOYMENT DIAGRAM) 47
11. Проблемы разработки программного обеспечения информационных систем 51
11.1. Решение проблем разработки информационных систем в .NET 51
11.2. Архитектура платформы .NET 53
11.3. ADO.NET: Провайдеры данных 58
11.4. Место ADO.NET в архитектуре .NET Framework 60
11.6. Основные объекты 62
12. Основы создания удобного пользовательского интерфейса 75
12.1. Факторы удобства использования и принципы создания удобного ПО 82
12.2. Методы разработки удобного программного обеспечения 86
13. Технологии тестирования 87
13.1. Основы тестирования 88
13.2. Фазы тестирования 90
13.3. Типы тестов 92

Введение

Технология программирования - дисциплина, изучающая технологические процессы программирования и порядок их прохождения.

Технологии программирования играло разную роль на разных этапах развития программирования. По мере повышения мощности компьютеров и развития средств и что привело к повышенному вниманию к технологии программирования. Резкое удешевление стоимости компьютеров и, в особенности, стоимости хранения информации на компьютерных носителях привело к широкому внедрению компьютеров практически во все сферы человеческой деятельности, что существенно изменило направленность технологии программирования. Человеческий фактор стал играть в ней решающую роль. Сформировалось достаточно глубокое понятие качества ПС, причем предпочтение стало отдаваться не столько эффективности ПС, сколько удобству работы с ним для пользователей (не говоря уже о его надежности).

В пособии основное внимание уделяется разработке больших систем

Сложные или ***«большие» программы,*** называемые также ***программными системами***, ***программными комплексами***, ***программными продуктами,*** характеризуются наличием факторов, связанных с их востребованностью. и готовностью пользователей платить деньги как за приобретение самой программы, так и за ее сопровождение и даже за специальное обучение работе с ней.

Обычно сложная программа обладает следующими свойствами.

Она решает одну или несколько связанных задач, зачастую сначала не имеющих четкой постановки, настолько важных для каких-либо лиц или организаций, что те приобретают значимые выгоды от ее использования.

Существенно, чтобы она была удобной в использовании. В частности, она должна включать достаточно полную и понятную пользователям документацию, а также набор документов для обучения работе с программой.

Ее низкая производительность на реальных данных приводит к значимым потерям для пользователей.

Ее неправильная работа наносит ощутимый ущерб пользователям и другим организациям и лицам, даже если сбои происходят не слишком часто.

Для выполнения своих задач она должна взаимодействовать с другими программами и программно-аппаратными системами, работать на разных платформах.

Пользователи, работающие с ней, приобретают дополнительные выгоды от того, что программа развивается, в нее вносятся новые функции и устраняются ошибки. Необходимо наличие проектной документации, позволяющей развивать ее, возможно, вовсе не тем разработчикам, которые ее создавали, без больших затрат на обратную разработку.

В ее разработку вовлечено значительное количество людей (более 5-ти человек). «Большую» программу практически невозможно написать с первой попытки, с небольшими усилиями и в одиночку.

Намного больше количество ее возможных пользователей, и еще больше тех лиц, деятельность которых будет так или иначе затронута ее работой и результатами.

1.НАДЕЖНОЕ ПРОГРАММНОЕ СРЕДСТВО КАК ПРОДУКТ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ.

1.1. Программа как формализованное описание процесса обработки данных. Программное средство.

Целью программирования является описание процессов обработки данных (в дальнейшем – просто *процессов*). Согласно ИФИПа *данные* (*data*) – это представление фактов и идей в формализованном виде, пригодном для передачи и переработке в некоем процессе, а *информация* (*information*) – это смысл, который придается данным при их представлении. *Обработка данных* (*data processing*) – это выполнение систематической последовательности действий с данными. Данные представляются и хранятся на т.н. *носителях данных*. Совокупность носителей данных, используемых при какой-либо обработке данных, будем называть *информационной средой* (*data medium*). Набор данных, содержащихся в какой-либо момент в информационной среде, будем называть *состоянием* этой информационной среды. *Процесс* можно определить как последовательность сменяющих друг друга состояний некоторой информационной среды.

Описать процесс – это значит определить последовательность состояний заданной информационной среды. Если мы хотим, чтобы по заданному описанию требуемый процесс порождался *автоматически* на каком-либо компьютере, необходимо, чтобы это описание было *формализованным*. Такое описание называется *программой*. С другой стороны, программа должна быть понятной и человеку, так как и при разработке программ, и при их использовании часто приходится выяснять, какой именно процесс она порождает. Поэтому программа составляется на удобном для человека формализованном *языке программирования*, с которого она автоматически переводится на язык соответствующего компьютера с помощью другой программы, называемой *транслятором*. Человеку (*программисту*), прежде чем составить программу на удобном для него языке программирования, приходится проделывать большую подготовительную работу по уточнению постановки задачи, выбору метода ее решения, выяснению специфики применения требуемой программы, прояснению общей организации разрабатываемой программы и многое другое. Использование этой информации может существенно упростить задачу понимания программы человеком, поэтому весьма полезно ее как-то фиксировать в виде отдельных документов (часто не формализованных, рассчитанных только для восприятия человеком).

Обычно программы разрабатываются в расчете на то, чтобы ими могли пользоваться люди, не участвующие в их разработке (их называют *пользователями*). Для освоения программы пользователем помимо ее текста требуется определенная дополнительная документация. Программа или логически связанная совокупность программ на носителях данных, снабженная программной документацией, называется *программным средством* (*ПС*). Программа позволяет осуществлять некоторую автоматическую обработку данных на компьютере. Программная документация позволяет понять, какие функции выполняет та или иная программа ПС, как подготовить исходные данные и запустить требуемую программу в процесс ее выполнения, а также: что означают получаемые результаты

(или каков эффект выполнения этой программы). Кроме того, программная документация помогает разобраться в самой программе, что необходимо, например, при ее модификации.

1.2. Понятие правильной программы.

Таким образом, можно считать, что продуктом технологии программирования является ПС, содержащее программы, выполняющие требуемые функции. Здесь под «программой» часто понимают правильную программу, т.е. программу, не содержащую ошибок. Однако, понятие ошибки в программе трактуется в среде программистов неоднозначно. Считается, что в программе имеется *ошибка*, если она не выполняет того, что разумно ожидать от нее пользователю. «Разумное ожидание» пользователя формируется на основании документации по применению этой программы. Следовательно, понятие ошибки в программе является существенно не формальным. В ПС программы и документация взаимно увязаны, образуют некоторую целостность. Поэтому правильнее говорить об ошибке не в программе, а в ПС в целом: будем считать, что в ПС имеется *ошибка* (*software error*), если оно не выполняет того, что разумно ожидать от него пользователю. В частности, разновидностью ошибки в ПС является несогласованность между программами ПС и документацией по их применению. В работе [1.3] выделяется в отдельное понятие частный случай ошибки в ПС, когда программа не соответствует своей функциональной спецификации (описанию, разрабатываемому на этапе, предшествующему непосредственному программированию). Такая ошибка в указанной работе называется *дефектом программы*. Однако выделение такой разновидности ошибки в отдельное понятие вряд ли оправданно, так как причиной ошибки может оказаться сама функциональная спецификация, а не программа.

1.3. Надежность программного средства.

Альтернативой правильного ПС является *надежное* ПС. *Надежность* (*reliability*) ПС – это его способность безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью. При этом под *отказом* в ПС понимают проявление в нем ошибки. Таким образом, надежное ПС не исключает наличия в нем ошибок – важно лишь, чтобы эти ошибки при практическом применении этого ПС в заданных условиях проявлялись достаточно редко. Убедиться, что ПС обладает таким свойством можно при его испытании путем тестирования, а также при практическом применении. Таким образом, фактически мы можем разрабатывать лишь надежные, а не правильные ПС.

ПС может обладать различной степенью надежности. Как измерять эту степень? Так же как в технике, степень надежности можно характеризовать вероятностью работы ПС без отказа в течение определенного периода времени. Однако в силу специфических особенностей ПС определение этой вероятности наталкивается на ряд трудностей по сравнению с решением этой задачи в

технике. Позже мы вернемся к более обстоятельному обсуждению этого вопроса.

При оценке степени надежности ПС следует также учитывать последствия каждого отказа. Некоторые ошибки в ПС могут вызывать лишь некоторые неудобства при его применении, тогда как другие ошибки могут иметь катастрофические последствия, например, угрожать человеческой жизни. Поэтому для оценки надежности ПС иногда используют дополнительные показатели, учитывающие стоимость (вред) для пользователя каждого отказа.

1.5. Технология программирования как технология разработки надежных программных средств.

В соответствии с обычным значением слова «технология» под *технологией программирования* (*programming technology*) будем понимать совокупность производственных процессов, приводящую к созданию требуемого ПС, а также описание этой совокупности процессов. Другими словами, технологию программирования мы будем понимать здесь в широком смысле как технологию разработки *программных средств*, включая в нее все процессы, начиная с момента зарождения идеи этого средства, и, в частности, связанные с созданием необходимой программной документации. Каждый процесс этой совокупности базируется на использовании каких-либо методов и средств, например, компьютер (в этом случае будем говорить о *компьютерной* технологии программирования).

Используется в литературе и близкое к технологии программирования понятие *программной инженерии* (s*oftware engineering*), определяемой как систематический подход к разработке, эксплуатации, сопровождению и изъятию из обращения программных средств. Главное различие между технологией программирования и программной инженерией как дисциплинами для изучения заключается в способе рассмотрения и систематизации материала. В технологии программирования акцент делается на изучении процессов разработки ПС (*технологических процессов*) и порядке их прохождения – методы и инструментальные средства разработки ПС *используются* в этих процессах (их применение и образуют технологические процессы). Тогда как в программной инженерии изучаются различные методы и инструментальные средства разработки ПС с точки зрения достижения определенных целей.

Имея ввиду, что надежность является неотъемлемым атрибутом ПС, мы будем рассматривать технологию программирования как технологию разработки *надежных* ПС. Это означает, что

- мы будем рассматривать все процессы разработки ПС, начиная с момента возникновения замысла ПС;
- нас будут интересовать не только вопросы построения программных конструкций, но и вопросы формального описания функций и принимаемых решений с точки зрения их человеческого (неформального) восприятия;

- в качестве продукта технологии принимается надежная (далеко не всегда правильная) ПС.

2. ИСТОЧНИКИ ОШИБОК В ПРОГРАММНЫХ СРЕДСТВАХ

2.3. Интеллектуальные возможности человека.

Теоретик программирования Дейкстра выделяет три интеллектуальные возможности человека, используемые при разработке ПС:

- способность к перебору,
- способность к абстракции,
- способность к математической индукции.

Способность человека к перебору связана с возможностью последовательного переключения внимания с одного предмета на другой, позволяя *узнавать* искомый предмет. Эта способность весьма ограничена - в среднем человек может уверенно (не сбиваясь) перебирать в пределах 1000 предметов (элементов).

При разработке ПС человек имеет дело с системами. Под *системой* будем понимать совокупность взаимодействующих (находящихся в отношениях) друг с другом элементов. ПС можно рассматривать как пример системы. Логически связанный набор программ является другим примером системы. Любая отдельная программа также является системой. Понять систему – значит осмысленно перебрать все пути взаимодействия между ее элементами. В силу ограниченности человека к перебору будем различать простые и сложные системы. Под *простой* будем понимать такую систему, в которой человек может уверенно перебирать все пути взаимодействия между ее элементами, а под *сложной* будем понимать такую систему, в которой он этого делать не в состоянии. Между простыми и сложными системами нет четкой границы, поэтому можно говорить и о промежуточном классе систем: к таким системам относятся программы, о которых программистский фольклор утверждает, что "в каждой отлаженной программе имеется хотя бы одна ошибка".

При разработке ПС мы не всегда можем уверенно знать о всех связях между ее элементами из-за возможных ошибок. Поэтому полезно уметь оценивать сложность системы по числу ее элементов: числом потенциальных путей взаимодействия между ее элементами, т.е. n! , где n – число ее элементов. Систему назовем *малой*, если $n < 7$ ($6! = 720 < 1000$), систему назовем *большой*, если $n > 7$. При $n=7$ имеем промежуточный класс систем. Малая система всегда проста, а большая может быть как простой, так и сложной. Задача технологии программирования – научиться делать большие системы простыми.

2.2. Неправильное преобразование как причина ошибок в программных средствах.

При разработке и использовании ПС мы многократно имеем дело с преобразованием информации из одной формы в другую (см. рис.2.1). Заказчик формулирует свои потребности в ПС в виде некоторых требований. Исходя из этих требований, разработчик создает внешнее описание ПС, используя при

этом спецификацию (описание) заданной аппаратуры и, возможно, спецификацию базового программного обеспечения. На основании внешнего описания и спецификации языка программирования создаются тексты программ ПС на этом языке. По внешнему описанию ПС разрабатывается также и пользовательская документация. Текст каждой программы является исходной информацией при любом ее преобразовании, в частности, при исправлении в ней ошибки. Пользователь на основании документации выполняет ряд действий для применения ПС и осуществляет интерпретацию получаемых результатов. Везде здесь, а также в ряде других процессов разработки ПС, имеет место указанный перевод информации.

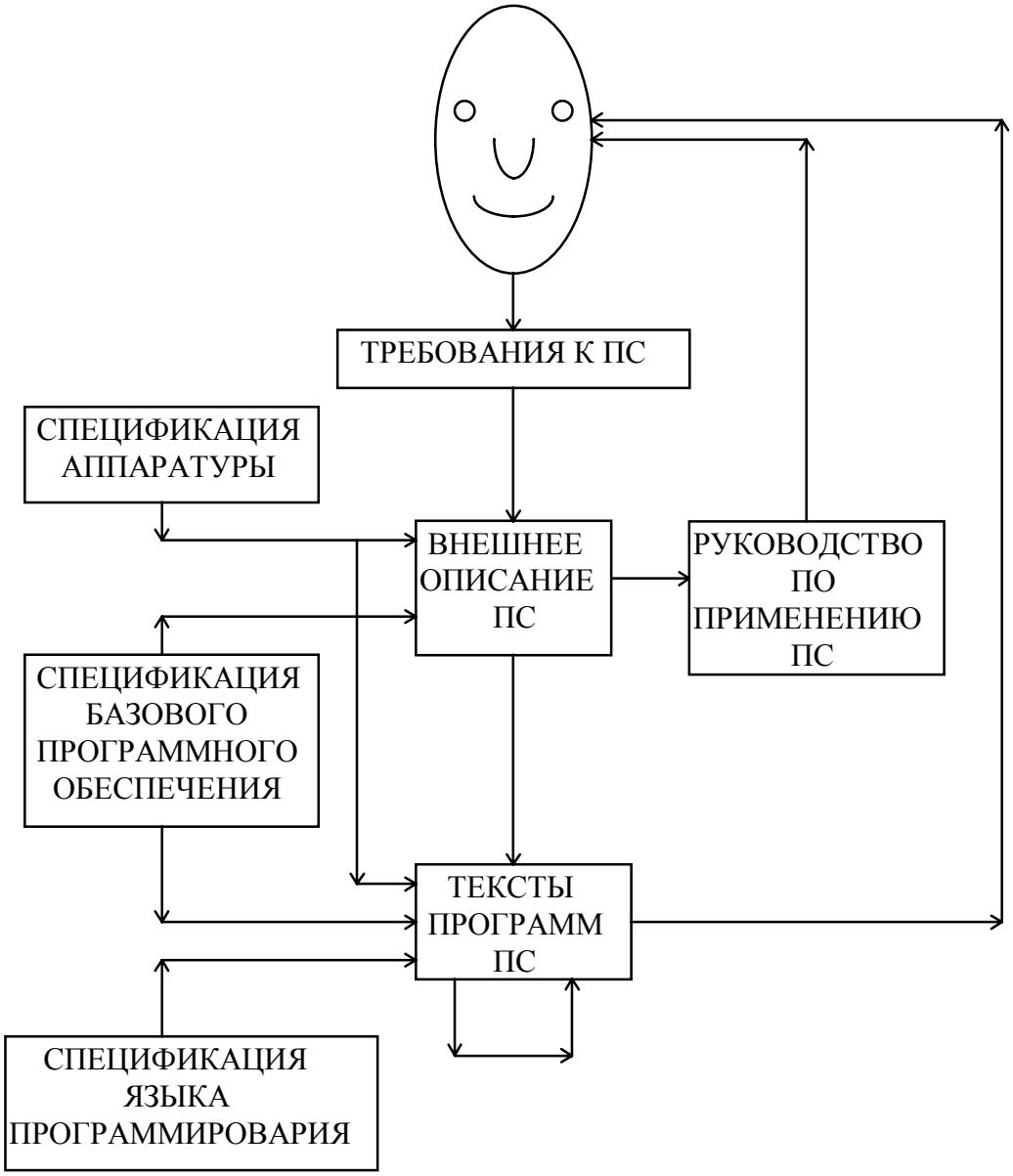


Рис. 2.1. Схема разработки и применения ПС.

На каждом из этих этапов перевод информации может быть осуществлен неправильно, например, из-за неправильного понимания исходного

представления информации. Возникнув на одном из этапов разработки ПС, ошибка в представлении информации преобразуется в новые ошибки результатов, полученных на последующих этапах разработки, и, в конечном счете, окажется в ПС.

Основные пути борьбы с ошибками.

Учитывая рассмотренные особенности действий человека при переводе можно указать следующие пути борьбы с ошибками:

- сужение пространства перебора (упрощение создаваемых систем),
- обеспечение требуемого уровня подготовки разработчика (это функции менеджеров коллектива разработчиков),
- обеспечение однозначности интерпретации представления информации,
- контроль правильности перевода (включая и контроль однозначности интерпретации).

3. ОБЩИЕ ПРИНЦИПЫ РАЗРАБОТКИ ПРОГРАММНЫХ СРЕДСТВ

3.1. Специфика разработки программных средств.

Разработка программных средств имеет ряд специфических особенностей [3.1].

- Прежде всего, следует отметить некоторое противостояние: *неформальный* характер требований к ПС (постановки задачи) и понятия ошибки в нем, но *формализованный* основной объект разработки – программы ПС.
- Разработка ПС носит *творческий характер* (на каждом шаге приходится делать какой-либо выбор, принимать какое-либо решение), а не сводится к выполнению какой-либо последовательности регламентированных действий.
- Следует отметить также особенность продукта разработки. Он представляет собой некоторую совокупность текстов (т.е. *статических* объектов), смысл же (семантика) этих текстов выражается процессами обработки данных и действиями пользователей, запускающих эти процессы (т.е. является *динамическим*).
- Продукт разработки имеет и другую специфическую особенность: ПС при своем использовании (эксплуатации) не расходуется и не расходует используемых ресурсов.

3.2. Жизненный цикл программного средства.

Под *жизненным циклом* ПС (*software life cycle*) понимают весь период его разработки и эксплуатации (использования), начиная от момента возникновения замысла ПС и кончая прекращением всех видов его использования. Жизненный цикл охватывает довольно сложный процесс создания и использования ПС (*software process*). Этот процесс может быть

организован по-разному для разных классов ПС и в зависимости от особенностей коллектива разработчиков.

В настоящее время можно выделить несколько подходов к организации процесса создания и использования ПС.

- Водопадный подход. При таком подходе разработка ПС состоит из цепочки этапов. На каждом этапе создаются документы, используемые на последующем этапе. В исходном документе фиксируются требования к ПС. В конце этой цепочки создаются программы, включаемые в ПС.
- Исследовательское программирование. Этот подход предполагает быструю (насколько это возможно) реализацию рабочих версий программ ПС, выполняющих лишь в первом приближении требуемые функции. После экспериментального применения реализованных программ производится их модификация с целью сделать их более полезными для пользователей. Этот процесс повторяется до тех пор, пока ПС не будет достаточно приемлемо для пользователей.
- Прототипирование. Этот подход моделирует начальную фазу исследовательского программирования вплоть до создания рабочих версий программ, предназначенных для проведения экспериментов с целью установить требования к ПС. В дальнейшем должна последовать разработка ПС по установленным требованиям в рамках какого-либо другого подхода (например, водопадного).
- Формальные преобразования. Этот подход включает разработку формальных спецификаций ПС и превращение их в программы путем корректных преобразований. На этом подходе базируется компьютерная технология (CASE-технология) разработки ПС.
- Сборочное программирование. Этот подход предполагает, что ПС конструируется, главным образом, из компонент, которые уже существуют. Должно быть некоторое хранилище (библиотека) таких компонент, каждая из которых может многократно использоваться в разных ПС.

В нашем курсе лекций мы, в основном, будем рассматривать водопадный подход с некоторыми модификациями. Во-первых, потому, что в этом подходе приходиться иметь дело с большинством процессов программной инженерии, а, во-вторых, потому, что в рамках этого подхода создается большинство больших программных систем. Именно этот подход рассматривается в качестве индустриального подхода разработки программного обеспечения.

В рамках водопадного подхода различают следующие стадии жизненного цикла ПС (см. рис. 3.1): разработку ПС, производство программных изделий (ПИ) и эксплуатацию ПС.

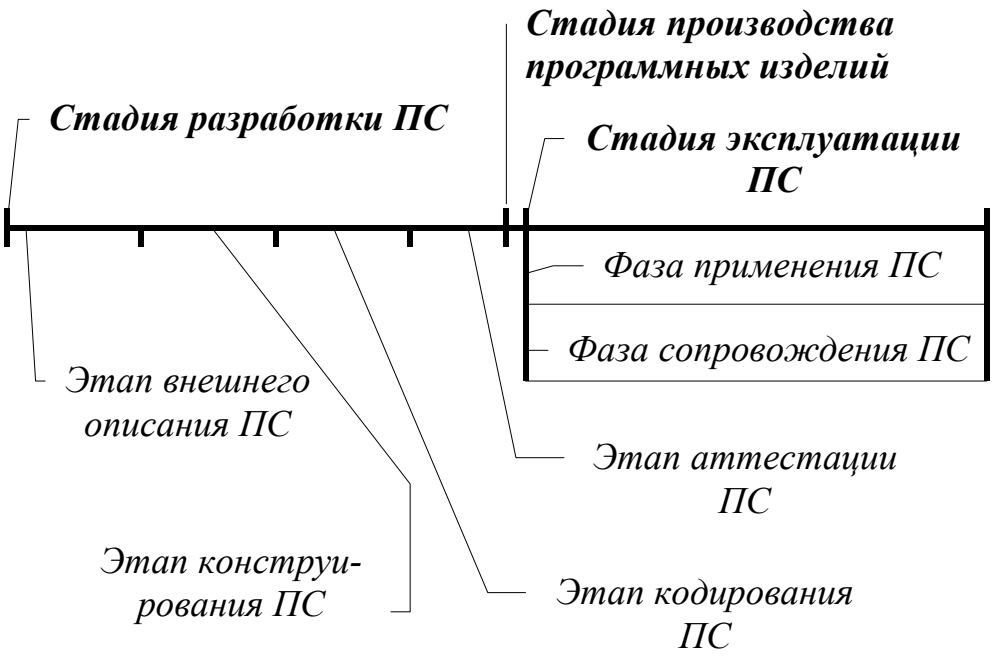


Рис. 3.1. Стадии и фазы жизненного цикла ПС.

Стадия *разработки* (*development*) ПС состоит из этапа его внешнего описания, этапа конструирования ПС, этапа кодирования (программирование в узком смысле) ПС и этапа аттестации ПС. Всем этим этапам сопутствуют процессы документирования и управления (*management*) ПС. Этапы конструирования и кодирования часто перекрываются, иногда довольно сильно. Это означает, что кодирование некоторых частей программного средства может быть начато до завершения этапа конструирования.

Этап *внешнего описания* ПС включает процессы, приводящие к созданию некоторого документа, который мы будем называть *внешним описанием* (*requirements document*) *ПС*. Этот документ является описанием поведения ПС с точки зрения внешнего по отношению к нему наблюдателя с фиксацией требований относительно его качества. Внешнее описание ПС начинается с анализа и определения требований к ПС со стороны пользователей (заказчика), а также включает процессы спецификации этих требований. *Конструирование* (*design*) ПС охватывает процессы: разработку архитектуры ПС, разработку структур программ ПС и их детальную спецификацию.

Кодирование (*coding*) ПС включает процессы создания текстов программ на языках программирование, их отладку с тестированием ПС.

На этапе *аттестации* (*acceptance*) ПС производится оценка качества ПС. Если эта оценка оказывается приемлемой для практического использования ПС, то разработка ПС считается законченной. Это обычно оформляется в виде некоторого документа, фиксирующего решение комиссии, проводящей аттестацию ПС.

Программное изделие (ПИ) – экземпляр или копия разработанного ПС. *Изготовление* ПИ – это процесс генерации и/или воспроизведения (снятия копии) программ и программных документов ПС с целью их поставки

пользователю для применения по назначению. *Производство* ПИ – это совокупность работ по обеспечению изготовления требуемого количества ПИ в установленные сроки.

Стадия *эксплуатации* ПС охватывает процессы хранения, внедрения и сопровождения ПС, а также транспортировки и применения ПИ по своему назначению.

Применение (*operation*) ПС – это использование ПС для решения практических задач на компьютере путем выполнения ее программ.

Сопровождение (*maintenance*) ПС – это процесс сбора информации о качестве ПС в эксплуатации, устранения обнаруженных в нем ошибок, его доработки и модификации, а также извещения пользователей о внесенных в него изменениях.

3.3. Понятие качества программного средства.

Каждое ПС должно выполнять определенные функции, т.е. делать то, что задумано. Хорошее ПС должно обладать еще целым рядом свойств, позволяющим успешно его использовать в течении длительного периода, т.е. обладать определенным качеством. *Качество (quality*) ПС – это совокупность его черт и характеристик, которые влияют на его способность удовлетворять заданные потребности пользователей. Качество ПС является удовлетворительным, когда оно обладает указанными свойствами в такой степени, чтобы гарантировать успешное его использование.

Совокупность свойств ПС, которая образует удовлетворительное для пользователя качество ПС, зависит от условий и характера эксплуатации этого ПС, т.е. от позиции, с которой должно рассматриваться качество этого ПС. Поэтому при описании качества ПС, прежде всего, должны быть фиксированы *критерии* отбора требуемых свойств ПС. В настоящее время *критериями качества ПС* (c*riteria of software quality*) принято считать:

* функциональность,
* надежность,
* легкость применения,
* эффективность,
* сопровождаемость,
* мобильность.

Функциональность – это способность ПС выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей. Набор указанных функций определяется во внешнем описании ПС.

Надежность обсуждалась в первом разделе.

Легкость применения – это характеристики ПС, которые позволяют минимизировать усилия пользователя по подготовке исходных данных, применению ПС и оценке полученных результатов, а также вызывать положительные эмоции определенного или подразумеваемого пользователя.

Эффективность – это отношение уровня услуг, предоставляемых ПС пользователю при заданных условиях, к объему используемых ресурсов.

Сопровождаемость – это характеристики ПС, которые позволяют минимизировать усилия по внесению изменений для устранения в нем ошибок и по его модификации в соответствии с изменяющимися потребностями пользователей.

Мобильность – это способность ПС быть перенесенным из одной среды (окружения) в другую, в частности, с одного компьютера на другой.

Функциональность и надежность являются обязательными критериями качества ПС. Остальные критерии используются в зависимости от потребностей пользователей в соответствии с требованиями к ПС.

3.4. Обеспечение надежности – основа разработки программных средств.

Рассмотрим теперь общие принципы обеспечения надежности ПС, что, как говорилось, является основным мотивом разработки ПС. В технике известны четыре подхода обеспечению надежности:

- предупреждение ошибок;
- самообнаружение ошибок;
- самоисправление ошибок;
- обеспечение устойчивости к ошибкам.

Целью подхода предупреждения ошибок – не допустить ошибок в готовых продуктах, в нашем случае – в ПС. Проведенное рассмотрение природы ошибок при разработке ПС позволяет для достижения этой цели сконцентрировать внимание на следующих вопросах:

- борьба со сложностью,
- обеспечение точности перевода,
- преодоление барьера между пользователем и разработчиком,
- обеспечение контроля принимаемых решений.

Этот подход связан с организацией процессов разработки ПС, т.е. с технологией программирования. В рамках этого подхода можно достигнуть приемлемого уровня надежности ПС.

Остальные три подхода связаны с организацией самих продуктов технологии, в нашем случае – программ. Самообнаружение ошибки в программе означает, что программа содержит средства обнаружения отказа в процессе ее выполнения. Самоисправление ошибки в программе означает не только обнаружение отказа в процессе ее выполнения, но и исправление последствий этого отказа, для чего в программе должны иметься соответствующие средства. Обеспечение устойчивости программы к ошибкам означает, что в программе содержатся средства, позволяющие локализовать область влияния отказа программы, либо уменьшить его неприятные последствия, а иногда предотвратить катастрофические последствия отказа

3.5. Методы борьбы со сложностью.

Мы уже обсуждали в лекции 2 сущность вопроса борьбы со сложностью при разработке ПС. Известны два общих метода борьбы со сложностью систем:

- обеспечения независимости компонентов системы;
- использование в системах иерархических структур.

Обеспечение независимости компонент означает разбиение системы на такие части, между которыми должны остаться по возможности меньше связей. Одним из воплощений этого метода является модульное программирование. Использование в системах иерархических структур позволяет локализовать связи между компонентами, допуская их лишь между компонентами, принадлежащими смежным уровням иерархии. Этот метод, по существу, означает разбиение большой системы на подсистемы, образующих малую систему.

Как обеспечить, чтобы ПС выполняла то, что пользователю разумно ожидать от нее? Для этого разработчикам необходимо правильно понять, во-первых, чего хочет пользователь, и, во-вторых, его уровень подготовки и окружающую его обстановку. Ясное описание соответствующей сферы деятельности пользователя или интересующей его проблемной области во многом облегчает достижение разработчиками этой цели. При разработке ПС следует привлекать пользователя для участия в процессах принятия решений, а также тщательно освоить особенности его работы (лучше всего – побывать в его "шкуре").

4. Методология ООАП

Как уже говорилось, разделение процесса разработки сложных программных приложений на отдельные этапы способствовало становлению концепции жизненного цикла программы. Под *жизненным циклом* (ЖЦ) программы понимают совокупность взаимосвязанных и следующих во времени этапов, начиная от разработки требований к ней и заканчивая полным отказом от ее использования.

4.1. Стандарты жизненного цикла

Чтобы получить представление о возможной структуре жизненного цикла ПО, обратимся сначала к соответствующим стандартам, описывающим технологические процессы. Международными организациями, такими, как:

- IEEE — читается «ай-трипл-и», Institute of Electrical and Electronic Engineers, Институт инженеров по электротехнике и электронике;
- ISO — International Standards Organization, Международная организация по стандартизации;
- EIA — Electronic Industry Association, Ассоциация электронной промышленности;

- IEC — International Electrotechnical Commission, Международная комиссия по электротехнике;

а также некоторыми национальными и региональными институтами и организациями (в основном, американскими и европейскими, поскольку именно они оказывают наибольшее влияние на развитие технологий разработки ПО во всем мире):

- ANSI — American National Standards Institute, Американский национальный институт стандартов;
- SEI — Software Engineering Institute, Институт программной инженерии;
- ECMA — European Computer Manufactures Association, Европейская ассоциация производителей компьютерного оборудования;

Разработан набор стандартов, регламентирующих различные аспекты жизненного цикла и вовлеченных в него процессов.

Группа стандартов ISO

• ISO/IEC 12207 Standard for Information Technology — Software Life Cycle Processes (процессы жизненного цикла ПО, есть его российский аналог **ГОСТ Р-1999**). Определяет общую структуру жизненного цикла ПО в виде 3-х ступенчатой модели, состоящей из процессов, видов деятельности и задач. Стандарт описывает вводимые элементы в терминах их целей и результатов, тем самым задавая неявно возможные взаимосвязи между ними, но не определяя четко структуру этих связей, возможную организацию элементов в рамках проекта и метрики, по которым можно было бы отслеживать ход работ и их результативность.
Самыми крупными элементами являются *процессы жизненного цикла ПО (lifecycle processes)*. Всего выделено 18 процессов, которые объединены в 4 группы.

Основные процессы	**Поддерживающие процессы**	**Организационные процессы**	**Адаптация**
Приобретение ПО; Передача ПО (в использование); Разработка ПО; Эксплуатация ПО; Поддержка ПО	Документирование; Управление конфигурациями; Обеспечение качества; Верификация; Валидация; Совместные экспертизы; Аудит; Разрешение проблем	Управление проектом; Управление инфраструктурой; Усовершенствование процессов; Управление персоналом	Адаптация описываемых стандартом процессов под нужды конкретного проекта

Стандарт ISO/I EC 12207, хотя и описывает общую структуру ЖЦ программы, не конкретизирует детали выполнения тех или иных этапов. Согласно принятым взглядам ЖЦ программы состоит из следующих этапов:

- Анализа предметной области и формулировки требований к программе
- Проектирования структуры программы
- Реализации программы в кодах (собственно программирования)
- Внедрения программы
- Сопровождения программы
- Отказа от использования программы

На этапе анализа предметной области и формулировки требований осуществляется определение функций, которые должна выполнять разрабатываемая программа, а также концептуализация предметной области. Эту работу выполняют аналитики совместно со специалистами предметной области. Результатом данного этапа должна являться некоторая концептуальная схема, содержащая описание основных компонентов и тех функций, которые они должны выполнять.

Этап проектирования структуры программы заключается в разработке детальной схемы будущей программы, на которой указываются классы, их свойства и методы, а также различные взаимосвязи между ними. Как правило, на этом этапе могут участвовать в работе аналитики, архитекторы и отдельные квалифицированные программисты. Результатом данного этапа должна стать детализированная схема программы, на которой указываются все классы и взаимосвязи между ними в процессе функционирования программы. Согласно методологии ООАП, именно данная схема должна служить исходной информацией для написания программного кода.

Этап программирования вряд ли нуждается в уточнении, поскольку является наиболее традиционным для программистов. Появление инструментариев *быстрой разработки приложений* (Rapid Application Development, RAD) позволило существенно сократить время и затраты на выполнение этого этапа. Результатом данного этапа является программное приложение, которое обладает требуемой функциональностью и способно решать нужные задачи в конкретной предметной области.

Этапы внедрения и сопровождения программы связаны с необходимостью настройки и конфигурирования среды программы, а также с устранением возникших в процессе ее использования ошибок. Иногда в качестве отдельного этапа выделяют *тестирование* программы, под которым понимают проверку работоспособности программы на некоторой совокупности исходных данных или при некоторых специальных режимах эксплуатации. Результатом этих этапов является повышение надежности программного приложения, исключающего возникновение критических ситуаций или нанесение ущерба компании, использующей данное приложение.

Рассматривая различные этапы ЖЦ программы, следует отметить одно важное обстоятельство. А именно, если появление RAD-инструментариев позволило существенно сократить сроки этапа программирования, то отсутствие

соответствующих средств для первых двух этапов долгое время сдерживало процесс разработки приложений. Развитие методологии ООАП было направлено на автоматизацию второго, а затем и первого этапов ЖЦ программы.

Методология ООАП тесно связана с концепцией *автоматизированной разработки программного обеспечения* (Computer Aided Software Engineering, CASE).

Вторая причина имеет более сложную природу, поскольку связана с графической нотацией, реализованной в том или ином CASE-средстве. Если языки программирования имеют строгий синтаксис, то попытки предложить подходящий синтаксис для визуального представления концептуальных схем БД были восприняты далеко неоднозначно. Появилось несколько подходов. Появление ***унифицированного языка моделирования*** (***Unified Modeling Language***, ***UML***), который ориентирован на решение задач первых двух этапов ЖЦ программ, было воспринято с большим оптимизмом всем сообществом программистов.

Последнее, на что следует обратить внимание, это необходимость построения предварительной модели программной системы, которую, согласно современным концепциям ООАП, следует считать результатом первых этапов ЖЦ программы.

4.2. Методология системного анализа и системного моделирования

Центральным понятием системного анализа является понятие *системы,* под которой понимается совокупность объектов, компонентов или элементов произвольной природы, образующих некоторую целостность. Определяющей предпосылкой выделения некоторой совокупности как системы является возникновение у нее новых свойств, которых не имеют составляющие ее элементы. Примеров систем можно привести достаточно много — это персональный компьютер, автомобиль, человек, биосфера, программа и др.

Важнейшими характеристиками любой системы являются ее структура и процесс функционирования. Под *структурой системы* понимают устойчивую во времени совокупность взаимосвязей между ее элементами или компонентами. Именно структура связывает воедино все элементы и препятствует распаду системы на отдельные компоненты. Структура системы может отражать самые различные взаимосвязи, в том числе и вложенность элементов одной системы в другую.

Процесс функционирования системы тесно связан с изменением ее свойств или поведения во времени. При этом важной характеристикой системы является ее *состояние,* под которым понимается совокупность свойств или признаков, которые в каждый момент времени отражают наиболее существенные особенности поведения системы.

Процесс функционирования системы отражает поведение системы во времени и может быть представлен как последовательное изменение ее состояний. Если система изменяет одно свое состояние на другое, то принято говорить, что система *переходит* из одного состояния в другое. Совокупность

признаков или условий изменения состояний системы в этом случае называется *переходом.* Для системы с дискретными состояниями процесс функционирования может быть представлен в виде последовательности состояний с соответствующими переходами.

Рассмотрение особенностей языка UML связано с вопросами логического или информационного моделирования систем.

5. История создания языка UML

Авторами UML являются Грани Буч (Grady Booch), Джеймс Румбах (James Rumbaugh) и Айвар Якобсон (Ivar Jacobson). Известные как "три товарища", в 80-х — начале 90-х годов они работали в разных организациях и независимо друг от друга продумывали методологии объектно-ориентированного анализа и проектирования, которые имели явные преимущества перед всеми остальными известными методами. В середине 90-х годов они стали заимствовать идеи друг друга и поэтому решили объединить свои усилия.

В 1994 году Румбаха пригласили в компанию Rational Software Corporation, где в это же время уже работал Буч. Через год к ним присоединился Якобсон.

Многие корпорации ощутили, что язык UML может оказаться полезен для достижения их стратегических целей. Это привело к возникновению консорциума UML, в который вошли такие компании, как DEC, Hewlett-Packard, Intellicorp, Microsoft, Oracle, Texas Instruments, Rational и другие. В 1997 году консорциум выработал первую версию UML и представил ее на рассмотрение группе OMG (Object Management Group), откликнувшись на ее запрос о подаче предложений по стандартному языку моделирования.

После расширения консорциума вышла версия 1.1 языка UML, которую группа OMG приняла в конце 1997 года. После этого OMG приступила к сопровождению UML и выпустила в 1998 году две его новые версии. Язык UML стал стандартом де-факто в области разработки программного обеспечения.

6. Основные понятия языка UML

Язык UML предназначен для решения следующих задач:

1. ***Предоставить в распоряжение пользователей легко воспринимаемый и выразительный язык визуального моделирования, специально предназначенный для разработки и документирования моделей сложных систем самого различного назначения.***

2. Описание языка UML должно поддерживать такую спецификацию моделей, которая не зависит от конкретных языков программирования и инструментальных средств проектирования программных систем.

3. ***Поощрять развитие рынка объектных инструментальных средств.*** С

4. ***Описание языка UML должно включать в себя семантический базис для понимания общих особенностей ООАП.***

5. ***Способствовать распространению объектных технологий и соответствующих понятий ООАП.***

7. ***Интегрировать в себя новейшие и наилучшие достижения практики ООАП.***

Язык UML включает набор графических элементов, используемых на диаграммах. Будучи языком, UML содержит правила для объединения этих элементов. Перед тем, как изучать эти элементы и правила, рассмотрим диаграммы, используемые при анализе системы.

Диаграммы используются для отображения различных представлений системы. Этот набор различных представлений называется *моделью.* Модель UML системы можно сравнить с художественно оформленной моделью здания. Важно отметить, что модель UML описывает, *что* должна будет делать система. В то же время, ничего не сообщается о том, *как* она будет реализована.

В нашем случае модель является набором диаграмм UML, которые можно исследовать, оценивать и изменять, чтобы понять и спроектировать систему.

Диаграмма классов

Все вещи в окружающем вас мире имеют атрибуты (свойства) и действуют определенным образом. Мы будем считать эти действия набором операций.

Также можно увидеть, что все вещи естественным образом разделяются по категориям (автомобили, мебель, стиральные машины и т.д.). Мы обращаемся к этим категориям как к классам. *Класс* — это категория или группа вещей, которая имеет сходные атрибуты и общие свойства. Например, любая вещь в классе самолет имеет такие атрибуты, как производитель, номер борта и количество мест. Свойства вещей в этом классе включают операции Выполнять рейс, Принять пассажиров, Лететь по трассе, Приземлиться.

На рис. 1.1 представлен пример обозначения UML, где показаны атрибуты и свойства рейса. Класс представляется прямоугольником, разделенным на три области. Самая верхняя область содержит имя, в средней располагаются атрибуты, а в самой нижней — операции. Диаграмма классов состоит из определенного количества таких прямоугольников, соединенных линиями, которые показывают, как классы связаны между собой.

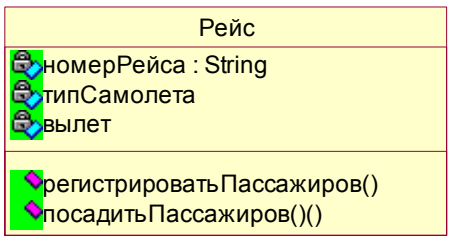


Рис.6. 1. Изображение класса в UML

Диаграммы классов представляют собой отправную точку процесса разработки. Диаграммы классов помогают также при анализе. Они позволяют аналитику общаться с клиентом в его терминологии и стимулируют процесс выявления важных деталей в проблеме, которую требуется решить.

Диаграмма вариантов использования

Вариант использования — это описание поведения системы с точки зрения пользователя. Для разработчиков системы это полезный инструмент,

предоставляющий надежную методику формирования требований к системе с точки зрения пользователя.

Варианты использования будут подробнее описаны позднее. А сейчас рассмотрим простой пример. Пассажир собирается вылететь заданным рейсом. На рис. 1.3 показано, как можно отразить это на диаграмме вариантов использования UML.

Рис. 6.2. Диаграмма прецедентов UML

Небольшая простая фигурка, соответствующая пассажиру, называется *исполнителем* (actor). Эллипс представляет *вариант использования.* Отметим, что исполнитель, инициирующий вариант использования, может быть как человеком, так и другой системой.

Диаграмма состояний

В каждый момент времени объект находится в каком-либо определенном состоянии: Пассажир может заказывать билет на рейс, ехать в аэропорт, регистрироваться на рейс, садиться в самолет. Диаграмма состояний UML, представленная на рис. 1.4, отображает эту сторону реальности. Диаграмма показывает, как стиральная машина переходит из одного состояния в другое.

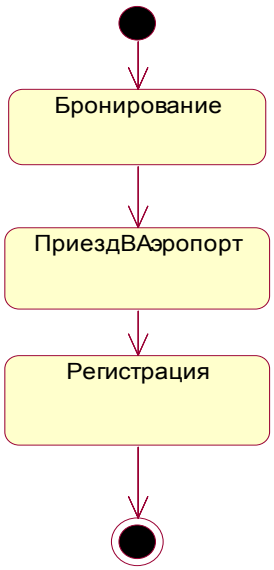


Рис. 6.3. Диаграмма состояний UML

Символ вверху диаграммы представляет начальное состояние, а символ внизу соответствует конечному.

Диаграмма последовательностей

Диаграммы классов и диаграммы объектов дают статическую информацию. Однако во время работы системы объекты взаимодействуют друг с другом, и это взаимодействие происходит во времени. Диаграмма последовательностей UML показывает временную динамику взаимодействия.

На рис. 1.5 показана диаграмма последовательностей, иллюстрирующая процесс действий пассажира для полета на заданном рейсе. На этой диаграмме время изменяется сверху вниз.

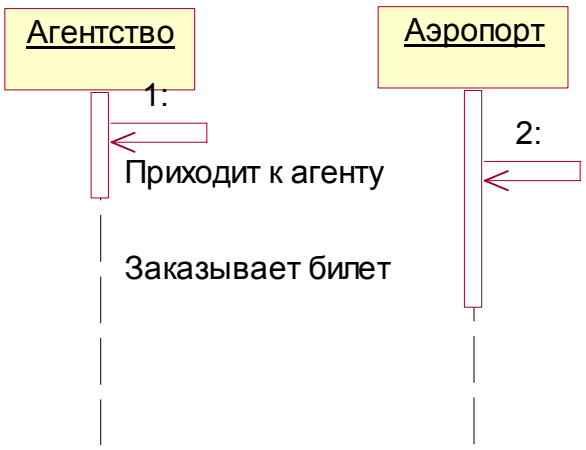


Рис. 6.4. Диаграмма последовательностей UML

Диаграмма видов деятельности

Действия, которые совершаются при выполнении прецедента или во время функционирования объекта, обычно происходят последовательно, как описанные выше шаги работы стиральной машины. На рис. 1.6 показано, как на диаграмме видов деятельности UML представлены шаги 4—6 этой цепочки.

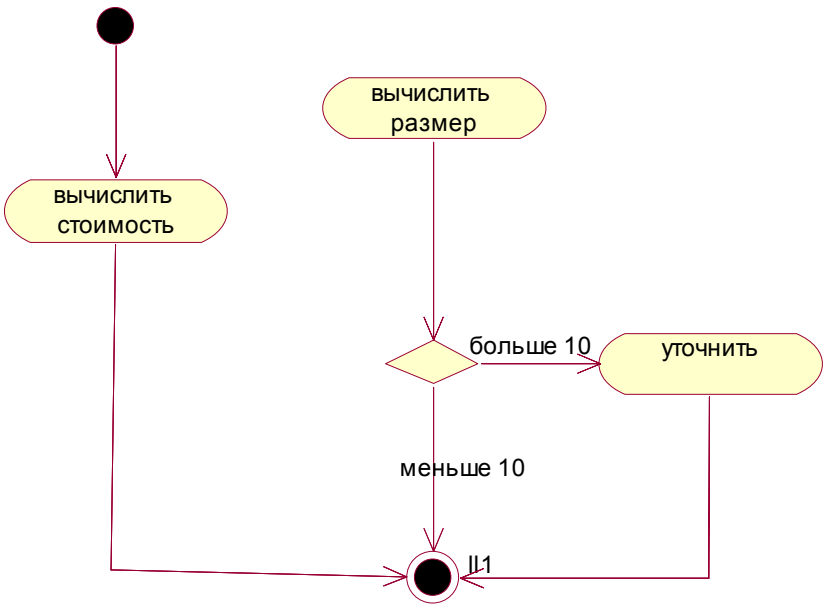


Рис. 6.5. Диаграмма видов деятельности UML

Диаграмма компонентов

Эта и следующая за ней диаграммы не имеют отношения к миру летательных аппаратов, потому что они предназначены только для компьютерных систем.

Проектирование современного программного обеспечения происходит путем разработки компонентов, что очень важно при организации совместных работ коллектива программистов. На рис. 5.6 изображен компонент программного обеспечения.

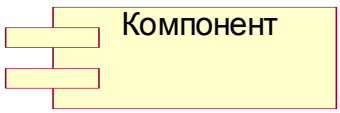


Рис. 6.6. Диаграмма компонента UML

Диаграмма развертывания

Диаграмма развертывания UML показывает физическую архитектуру компьютерной системы. Она представляет компьютеры и устройства, их соединения между собой, а также программное обеспечение, размещенное на каждой машине. Компьютеры изображаются в виде куба, а соединения между ними — в виде линий. Пример диаграммы развертывания показан на рис. 1.9.

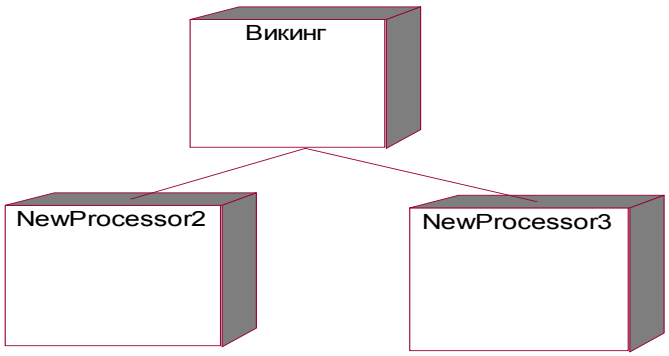


Рис. 6.7.Диаграмма развертывания

Иногда необходимо организовать элементы диаграмм в группы. Возможно, потребуется показать, что некоторые классы или компоненты являются частью отдельной подсистемы. Чтобы сделать это, их нужно сгруппировать в *пакет,* который представляется папкой с закладкой, как на рис. 1.10.

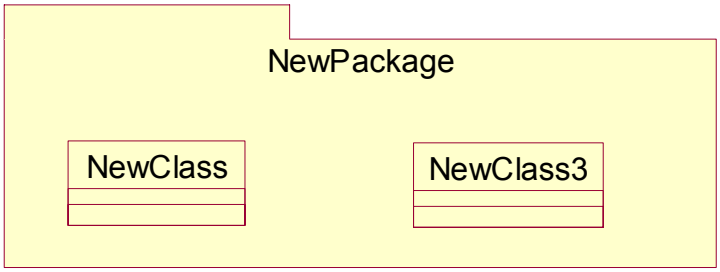


Рис. 6.8. Пакет UML

Рассмотрим более подробно базовые аспекты языка UML.

7. ДИАГРАММА ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ (USE CASE DIAGRAM)

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

Разработка диаграммы вариантов использования преследует цели:

- Определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы.
- Сформулировать общие требования к функциональному поведению проектируемой системы.
- Разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей.
- Подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

Суть данной диаграммы состоит в следующем: проектируемая система представляется в виде множества сущностей или актеров, взаимодействующих с системой с помощью так называемых вариантов использования. При этом актером (actor) или действующим лицом называется любая сущность, взаимодействующая с системой извне. Это может быть человек, техническое устройство, программа или любая другая система, которая может служить источником воздействия на моделируемую систему так, как определит сам разработчик. В свою очередь, вариант использования (use case) служит для описания сервисов, которые система предоставляет актеру. При этом ничего не говори о том, каким образом будет реализовано взаимодействие актеров с системой.

В самом общем случае, диаграмма вариантов использования представляет собой граф специального вида, который является графической нотацией, представления конкретных вариантов использования, актеров, возможно некоторых интерфейсов, и отношений между этими элементами. При этом отдельные компоненты диаграммы могут быть заключены в прямоугольник, который обозначает проектируемую систему в целом.

Вариант использования

Конструкция или стандартный элемент языка UML *вариант использования* применяется для спецификации общих особенностей поведения системы или любой другой сущности предметной области без рассмотрения внутренней структуры этой сущности. Диаграмма вариантов может дополняться пояснительным текстом, который раскрывает смысл или семантику составляющих ее компонентов. Такой пояснительный текст получил название *примечания* или *сценария.*

Отдельный вариант использования обозначается на диаграмме эллипсом, под которым содержится его краткое название или имя в форме глагола с пояснительными словами (рис. 6.1).

Рис.7.1.. Графическое обозначение варианта использования

Цель варианта использования заключается в том, чтобы определить законченный аспект или фрагмент поведения некоторой сущности без раскрытия внутренней структуры этой сущности

Применение вариантов использования на всех уровнях диаграммы позволяет достичь требуемого уровня унификации обозначений для представления функциональности подсистем и системы в целом. Примерами вариантов использования могут являться следующие действия: бронирование билета на рейс, оформление заказа на покупку товара, регистрация пассажира в аэропорту, отображение графической формы на экране монитора и другие действия.

Актеры

Актер представляет собой любую внешнюю по отношению к моделируемой системе сущность, которая взаимодействует с системой и использует ее функциональные возможности для достижения определенных целей или решения частных задач. Стандартным графическим обозначением актера на диаграммах является фигурка "человечка", под которой записывается конкретное имя актера (рис. 6.2).

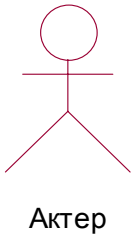


Рис. 7.2.. Графическое обозначение актера

В некоторых случаях актер может обозначаться в виде прямоугольника класса с ключевым словом "актер" и обычными составляющими элементами класса. Имена актеров должны записываться заглавными буквами и следовать рекомендациям использования имен для типов и классов модели. При этом символ отдельного актера связывает соответствующее описание актера с конкретным именем..

Имя актера должно быть достаточно информативным с точки зрения семантики. Вполне подходят для этой цели наименования должностей в компании (например, продавец, кассир, менеджер, президент). Не рекомендуется давать актерам имена собственные или моделей конкретных устройств (например, "маршрутизатор Cisco 3640"), Примерами актеров могут быть: клиент банка, банковский служащий, продавец магазина, менеджер отдела продаж, пассажир авиарейса, водитель автомобиля, администратор гостиницы, сотовый телефон и другие сущности, имеющие отношение к концептуальной модели соответствующей предметной области.

Интерфейсы

Интерфейс (interface) служит для спецификации параметров модели, которые видимы извне без указания их внутренней структуры. В языке UML интерфейс является классификатором и характеризует только ограниченную часть поведения моделируемой сущности. интерфейс эквивалентен абстрактному классу без атрибутов и методов с наличием только абстрактных операций.

На диаграмме вариантов использования интерфейс изображается в виде маленького круга, рядом с которым записывается его имя (рис. 6.3, а). В качестве имени может быть существительное, которое характеризует соответствующую информацию или сервис (например, "датчик", "сирена", "видеокамера"), но чаще строка текста (например, "запрос к базе данных", "форма ввода", "устройство подачи звукового сигнала"). Если имя записывается на английском, то оно должно начинаться с заглавной буквы I, например, ISecure Information, ISensor (рис. 6.3, б).

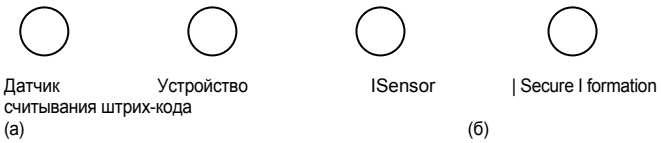


Рис. 7.3. *Графическое изображение интерфейсов на диаграммах вариантов использования*

Графический символ отдельного интерфейса может соединяться на диаграмме сплошной линией с тем вариантом использования, который его поддерживает. Сплошная линия в этом случае указывает на тот факт, что связанный с интерфейсом вариант использования должен реализовывать все операции, необходимые для данного интерфейса, а возможно и больше {рис. 6.4, а). Кроме этого, интерфейсы могут соединяться с вариантами использования пунктирной линией со стрелкой (рис. 4.4, б), означающей, что вариант использования предназначен для спецификации только того сервиса, который необходим для реализации данного интерфейса.

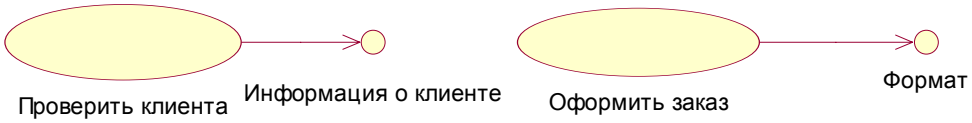


Рис. 7.4. *Графическое изображение взаимосвязей интерфейсов с вариантами использования*

Важность интерфейсов заключается в том, что они определяют стыковочные узлы в проектируемой системе, что совершенно необходимо для организации коллективной работы над проектом. Более того, спецификация интерфейсов способствует "безболезненной" модификации уже существующей системы при переходе на новые технологические решения. В этом случае изменению подвергается только реализация операций, но никак не функциональность самой системы. А это обеспечивает совместимость последующих версий программ с первоначальными при спиральной технологии разработки программных систем.

Примечания

Примечания (notes) в языке UML предназначены для включения в модель произвольной текстовой информации, имеющей непосредственное отношение к контексту разрабатываемого проекта. В качестве такой информации могут быть комментарии разработчика (например, дата и версия разработки диаграммы или ее отдельных компонентов), ограничения (например, на значения отдельных связей или экземпляры сущностей) и помеченные значения.

Графически примечания обозначаются прямоугольником с "загнутым" верхним правым уголком (рис. 4.5). Внутри прямоугольника содержится текст примечания. Примечание может относиться к любому элементу диаграммы, в этом случае их соединяет пунктирная линия. Если примечание относится к нескольким элементам, то от него проводятся, соответственно, несколько линий. Разумеется, примечания могут присутствовать не только на диаграмме вариантов использования, но и на других канонических диаграммах.

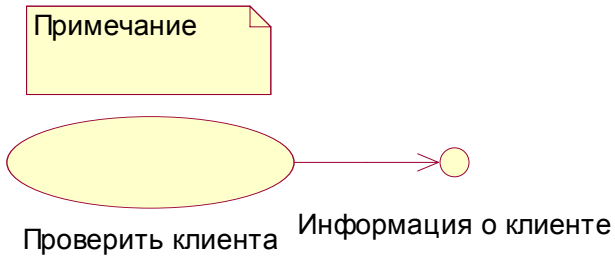


Рис. 7.5. *Примеры примечаний в языке UML*

Если в примечании указывается ключевое слово "constraint", то данное примечание является ограничением, налагаемым на соответствующий элемент модели, но не на саму диаграмму. При этом запись ограничения заключается в фигурные скобки и должна соответствовать правилам правильного построения выражений языка OCL.

Отношения на диаграмме вариантов использования

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

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

- Отношение ассоциации (association relationship)

- Отношение расширения (extend relationship)
- Отношение обобщения (generalization relationship)
- Отношение включения (include relationship)

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

Отношение ассоциации

Отношение ассоциации является одним из фундаментальных понятий в языке UML и в той или иной степени используется при построении всех графических моделей систем в форме канонических диаграмм. Применительно к диаграммам вариантов использования оно служит для обозначения специфической роли актера в отдельном варианте использования. Другими словами, ассоциация специфицирует семантические особенности взаимодействия актеров и вариантов использования в графической модели системы. Таким образом, это отношение устанавливает, какую конкретную роль играет актер при взаимодействии с экземпляром варианта использования. На диаграмме вариантов использования, так же как и на других диаграммах, отношение ассоциации обозначается сплошной линией между актером и вариантом использования. Эта линия может иметь дополнительные условные обозначения, такие, например, как имя и кратность (рис. 6.6).

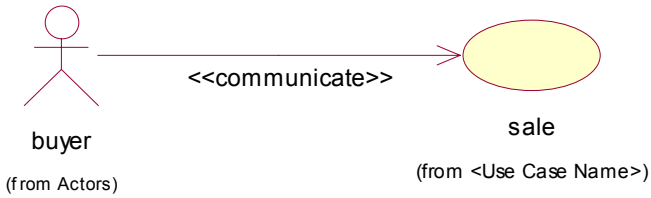


Рис. 7.6. Пример графического представления отношения ассоциации между актером и вариантом использования

Кратность (multiplicity) ассоциации указывается рядом с обозначением компонента диаграммы, который является участником данной ассоциации. Кратность характеризует общее количество конкретных экземпляров данного компонента, которые могут выступать в качестве элементов данной ассоциации. Применительно к диаграммам вариантов использования кратность имеет специальное обозначение в форме одной или нескольких цифр и, возможно, специального символа "*" (звездочка).

Для диаграмм вариантов использования наиболее распространенными являются четыре основные формы записи кратности отношения ассоциации:

- Целое неотрицательное число (включая цифру 0). Примером этой формы записи кратности ассоциации является указам кратности "1" для актера "Клиент банка" (рис. 6.6).
- Два целых неотрицательных числа, разделенные двумя точками и записанные в виде: *"первое число .. второе число"*. Данная запись в языке

UML соответствует нотации для множества или интервала целых чисел, которая применяется в некоторых языках программирования для обозначения границ массива элементов. Пример такой формы записи кратности ассоциации - *"1..5"*..

- Два символа, разделенные двумя точками. При этом первый из них является целым неотрицательным числом или 0, а второй — специальным символом "*". Здесь символ "*"обозначает произвольное конечное целое неотрицательное число, значение которого неизвестно на момент задания соответствующего отношения ассоциации.
- Единственный символ "*", который является сокращением записи интервала *"0..*"*. В этом случае количество отдельных экземпляров данного компонента отношения ассоциации может быть любым целым неотрицательным числом. При этом 0 означает, что для некоторых экземпляров соответствующего компонента данное отношение ассоциации может вовсе не иметь места.

Если кратность отношения ассоциации не указана, то по умолчанию принимается ее значение, равное I.

Отношение расширения

Отношение расширения определяет взаимосвязь экземпляров отдельного варианта использования с более общим вариантом, свойства которого определяются на основе способа совместного объединения данных экземпляров. В метамодели отношение расширения является направленным и указывает, что применительно к отдельным примерам некоторого варианта использования должны быть выполнены конкретные условия, определенные для расширения данного варианта использования. Так, если имеет место отношение расширения от варианта использования *Л* к варианту использования *B,* то это означает, что свойства экземпляра варианта использования *B* могут быть дополнены благодаря наличию свойств у расширенного варианта использования *A*.

Отношение расширения между вариантами использования обозначается пунктирной линией со стрелкой (вариант отношения зависимости), направленной от того варианта использования, который является расширением для исходного варианта использования. Данная линия со стрелкой помечается ключевым словом "extend" ("расширяет"), как показано на рис. 4.7.

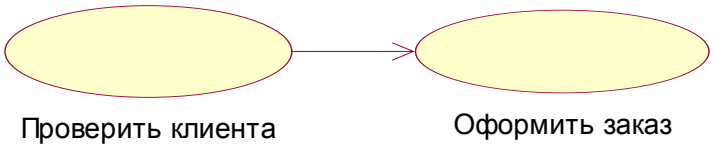


Рис. 7.7. Пример графического изображения отношения расширения между вариантами использования

Отношение расширения отмечает тот факт, что один из вариантов использования может присоединять к своему поведению некоторое дополнительное поведение, определенное для другого варианта использования.

Данное отношение включает в себя некоторое условие и ссылки на точки расширения в базовом варианте использования. Чтобы расширение имело место, должно быть выполнено определенное условие данного отношения. Ссылки на точки расширения определяют те места в базовом варианте использования, в которые должно быть помещено соответствующее расширение при выполнении условия.

Один из вариантов использования может быть расширением для нескольких базовых вариантов, а также иметь в качестве собственных расширений несколько других вариантов. Базовый вариант использования может дополнительно никак не зависеть от своих расширений.

В представленном выше примере (рис. 6.7) при оформлении заказа на приобретение товара только в некоторых случаях может потребоваться предоставление клиенту каталога всех товаров. При этом условием расширения является запрос от клиента на получение каталога товаров. Очевидно, что после получения каталога клиенту необходимо некоторое время на его изучение, в течение которого оформление заказа приостанавливается. После ознакомления с каталогом клиент решает либо в пользу выбора отдельного товара, либо отказа от покупки вообще. Сервис или вариант использования "Оформить заказ на приобретение товара" может отреагировать на выбор клиента уже *после* того, как клиент получит для ознакомления каталог товаров.

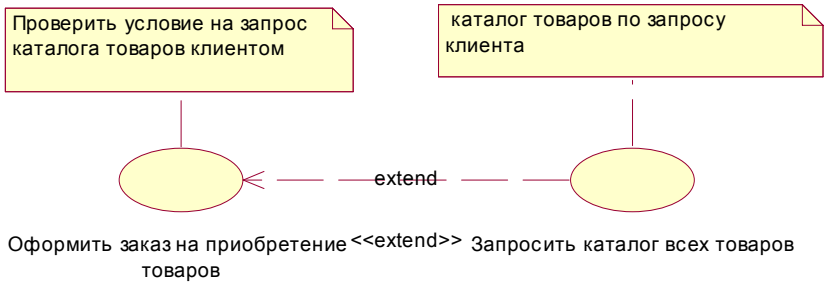


Рис. 7.8. *Графическое изображение отношения расширения с примечаниями условий выполнения вариантов использования*

Ссылки на расположение точек расширения могут быть представлены различными способами, например, с помощью текста примечания на естественном языке (рис. 4.8).

Отношение обобщения

Отношение обобщения служит для указания того факта, что некоторый вариант использования *A* может быть обобщен до варианта использования *B*. В этом случае вариант *A* будет являться специализацией варианта *B*. При этом *B* называется ***предком*** или родителем по отношению *A,* а вариант *A* ***-потомком*** по отношению к варианту использования *B*. Следует подчеркнуть, что потомок наследует все свойства и поведение своего родителя, а также может быть дополнен новыми свойствами и особенностями поведения. Графически данное отношение обозначается сплошной линией со стрелкой в форме незакрашенного треугольника, которая указывает на родительский вариант использования (рис. 6.9). Эта линия со стрелкой имеет специальное название — *стрелка "обобщение"*.

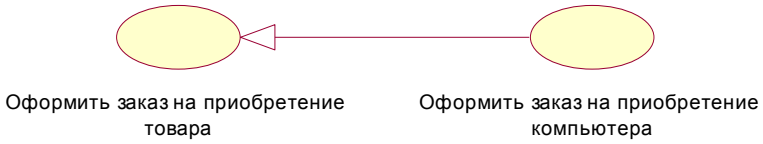


Рис. 7.9. Пример графического изображения отношения обобщения между вариантами использования

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

Между отдельными актерами также может существовать отношение обобщения. Данное отношение является направленным и указывает на факт специализации одних актеров относительно других. Например, отношение обобщения от актера *A* к актеру *B* отмечает тот факт, что каждый экземпляр; актера *A* является одновременно экземпляром актера *B* и обладает всеми • свойствами. В этом случае актер *B* является родителем по отношению к актеру *A,* а актер *A.* соответственно, потомком актера *B.* При этом актер *A об*ладает способностью играть такое же множество ролей, что и актер *B.* Графически данное отношение также обозначается стрелкой обобщения, т. е. сплошной линией со стрелкой в форме незакрашенного треугольника, которая указывает на родительского актера (рис.6.10).

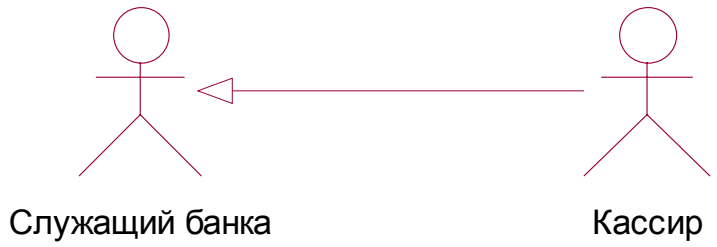


Рис. 7.10. *.Пример графического изображения отношения обобщения между актерами*

Отношение включения

Отношение включения между двумя вариантами использования указывает, что некоторое заданное поведение для одного варианта использования включается в качестве составного компонента в последовательность поведения другого варианта использования. Данное отношение является направленным бинарным отношением в том смысле, что пара экземпляров вариантов использования всегда упорядочена в отношении включения.

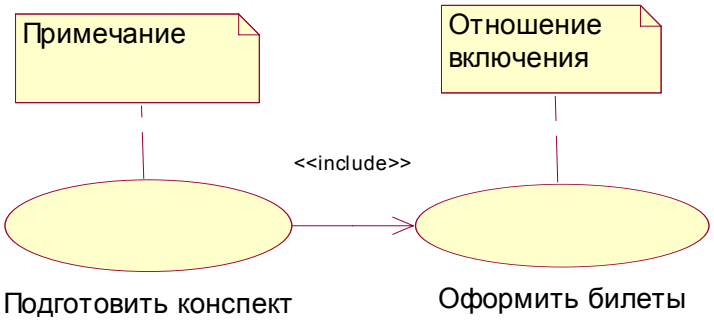


Рис.7.11 Отношение включения

Пример построения диаграммы вариантов использования

В качестве примера рассмотрим процесс моделирования системы приема экзамена преподавателем у студента.

использования разрабатываемой диаграммы, первоначальная структура которой может включать в себя только двух указанных актеров и единственный вариант использования (рис. 6.12).

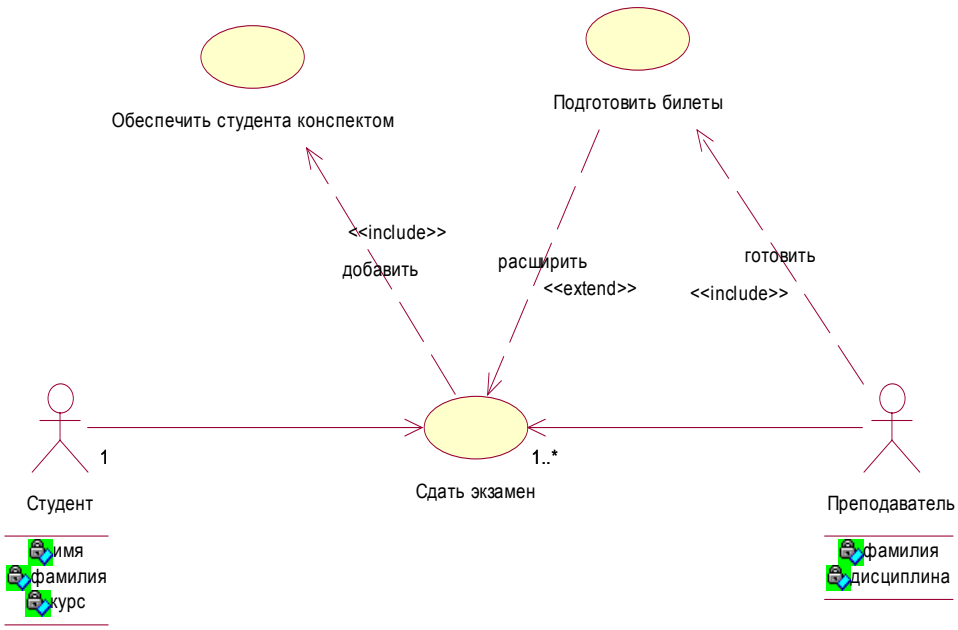


Рис.7.12. Диаграмма вариантов использования

Построение диаграммы вариантов использования является самым первым этапом процесса объектно-ориентированного анализа и проектирования, цель которого — представить совокупность требований к поведению проектируемой системы.

8. ДИАГРАММА КЛАССОВ (CLASS DIAGRAM)

Центральное место в ООАП занимает разработка логической модели системы в виде диаграммы классов.

Диаграмма классов (class diagram) служит для представления статической структуры модели системы в терминологии классов объектно-ориентированной программирования. Диаграмма классов может отражать, в частности, различные взаимосвязи между отдельными сущностями предметной области такими как объекты и подсистемы, а

также описывает их внутреннюю структуру и типы отношений. На данной диаграмме не указывается информация о временных аспектах функционирования системы. С этой точки зрения диаграмма классов является дальнейшим развитием концептуальной модели проектируемой системы.

Диаграмма классов представляет собой некоторый граф, вершинами которого являются элементы типа "классификатор", которые связаны различии ми типами структурных отношений. Следует заметить, что диаграмма классов может также содержать интерфейсы, пакеты, отношения и даже отдельные экземпляры, такие как объекты и связи. Когда говорят о данной диаграмме, имеют в виду статическую структурную модель проектируемой системы. Поэтому диаграмму классов принято считать графическим представлением таких структурных взаимосвязей логической модели системы, которые не зависят или инвариантны от времени.

Класс

Класс (class) в языке UML служит для обозначения множества объектов, которые обладают одинаковой структурой, поведением и отношениями с объектами из других классов. Графически класс изображается в виде прямоугольника, который дополнительно может быть разделен горизонтальными линиями на разделы или секции (рис. 7.1). В этих разделах могут указываться имя класса, атрибуты (переменные) и операции (методы).

Рис. 8.1. Графическое изображение класса на диаграмме классов

Обязательным элементов обозначения класса является его имя. На начальных этапах разработки диаграммы отдельные классы могут обозначаться простым прямоугольником с указанием только имени соответствующего класса (рис. 7.1). По мере проработки отдельных компонентов диаграммы описания классов дополняются атрибутами и операциями.

Предполагается, что окончательный вариант диаграммы содержит наиболее полное описание классов, которые состоят из трех разделов или секций. Иногда в обозначениях классов используется дополнительный четвертый раздел, в котором приводится семантическая информация справочного характера или явно указываются исключительные ситуации.

Даже если секция атрибутов и операций является пустой, в обозначении класса она выделяется горизонтальной линией, чтобы сразу отличить класс от других элементов языка UML. Примеры графического изображения классов на диаграмме классов приведены на рис. 5.2. В первом случае для класса "Прямоугольник" (рис. 5.2, а) указаны только его атрибуты — точки на координатной плоскости, которые

определяют его расположение. Для класса "Окно" (рис. 5.2, б) указаны только его операции, секция атрибутов оставлена пустой. Для класса "Счет" (рис. 5.2, в) дополнительно изображена четвертая секция, в которой указано исключение — отказ от обработки просроченной кредитной карточки.

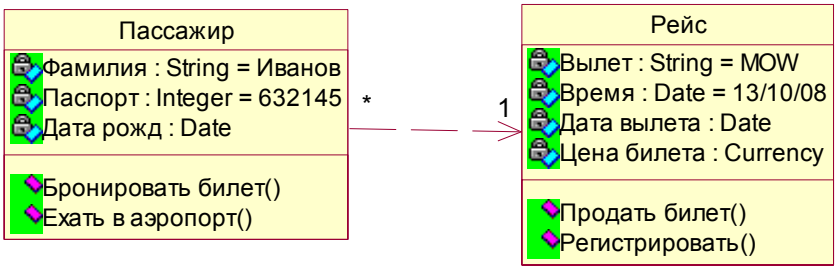


Рис. 8.2. Примеры графического изображения класса на диаграмме классов

Имя класса

Имя класса должно быть уникальным в пределах пакета, который описывается некоторой совокупностью диаграмм классов (возможно, одной диаграммой). Оно указывается в первой верхней секции прямоугольника. Рекомендуется в качестве имен классов использовать существительные, записанные по практическим соображениям без пробелов.

Примерами имен классов могут быть такие существительные, как "Сотрудник", "Компания", "Руководитель", "Пассажир", "Рейс", "Менеджер", "Офис" и другие, имеющие непосредственное отношение к моделируемой предметной области и функциональному назначению проектируемой системы.

Класс может не иметь экземпляров или объектов. В этом случае он называется *абстрактным* классом, а для обозначения его имени используется наклонный шрифт.

Атрибуты класса

Во второй сверху секции прямоугольника класса записываются его *атрибуты* (attributes) или свойства. В языке UML принята определенная стандартизация записи атрибутов класса, которая подчиняется некоторым синтаксическим правилам. Каждому атрибуту класса соответствует отдельная строка текста, которая состоит из квантора видимости атрибута, имени атрибута, его кратности, типа значений атрибута и, возможно, его исходного значения:

<квантор видимости><имя атрибута>[кратность]:

<тип атрибута> = <исходное значение>{строка-свойство}

Квантор видимости может принимать одно из трех возможных значений и. соответственно, отображается при помощи специальных символов:

- Символ "+" обозначает атрибут с областью видимости типа общедоступный (public).
- Символ "#" обозначает атрибут с областью видимости типа защищенный

(protected). Атрибут с этой областью видимости недоступен или невиден для всех классов, за исключением подклассов данного класса.

- Символ "-" обозначает атрибут с областью видимости типа закрытый (private). Атрибут с этой областью видимости недоступен или невиден для всех классов без исключения.

Квантор видимости может быть опущен. В этом случае его отсутствие просто означает, что видимость атрибута не указывается.

Имя атрибута представляет собой строку текста, которая используется в качестве идентификатора соответствующего атрибута и поэтому должна быть уникальной в пределах /данного класса. Имя атрибута является единственным обязательным элементом синтаксического обозначения атрибута.

Кратность атрибута характеризует общее количество конкретных атрибутов данного типа, входящих в состав отдельного класса. В общем случае кратность записывается в форме строки текста в квадратных скобках после имени соответствующего атрибута:

В качестве примера рассмотрим следующие варианты задания кратности атрибутов.

- *[0.. 1]* означает, что кратность атрибута может принимать значение 0 или 1. При этом 0 означает отсутствие значения для данного атрибута.
- *[0.. *]*означает, что кратность атрибута может принимать любое положительное целое значение большее или равное 0. Эта кратность может быть записана короче в виде простого символа - [*].
- *[1..*]* означает, что кратность атрибута может принимать любое положительное целое значение большее или равное 1.
- *[1..5]* означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 4, *5*.
- *[1..3,5,7]* означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 5, *1*.
- *[1..3,7..10]* означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 7, 8, 9, 10.
- *[1..3,7..*]* означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, а также любое положительное целое значение большее или равное 7.

Если кратность атрибута не указана, то по умолчанию принимается ее значение равное 1..1, т. е. в точности 1.

Тип атрибута представляет собой выражение, семантика которого определяется языком спецификации соответствующей модели (Рис.7.2).

Операция

В третьей сверху секции прямоугольника записываются операции или методы класса. ***Операция*** (operation) представляет собой некоторый сервис, представляющий каждый экземпляр класса по определенному требованию, совокупность операций характеризует функциональный аспект поведения

класса. Запись операций класса в языке UML также стандартизована и подчиняется определенным синтаксическим правилам. При этом каждой операции класса соответствует отдельная строка, которая состоит из квантора видимости операции, имени операции, выражения типа возвращаемого операцией значения и, возможно, строка-свойство данной операции:

<квантор **видимости> <имя** операции>(список параметров):

<выражение типа возвращаемого значения>{строка-свойство}

Квантор видимости, как и в случае атрибутов класса, может принимать одно из трех возможных значений и, соответственно, отображается при помощи специального символа. Символ "+" обозначает операцию типа общедоступный (public). Символ "#" обозначает операцию типа защищенный (protected). И, наконец, символ "-" используется для обозначения операции с областью видимости типа закрытый (private).

Квантор видимости для операции может быть опущен. В этом случае его отсутствие просто означает, что видимость операции не указывается.

Имя операции представляет собой строку текста, которая используется в качестве идентификатора соответствующей операции и поэтому должна быть уникальной в пределах данного класса.

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

<вид параметра><имя параметра>:<выражение типа>=<значение параметра ло умодчаниюХ

Здесь ***вид параметра*** — есть одно из ключевых слов in, out или inout со значением in по умолчанию, в случае если вид параметра не указывается. *Имя параметра* есть идентификатор соответствующего формального параметра. ***Выражение типа*** является зависимой от конкретного языка программирования спецификацией типа возвращаемого значения для соответствующего формального параметра. Наконец, ***значение по умолчанию*** в общем случае представляет собой выражение для значения формального параметра, синтаксис которого зависит от конкретного языка программирования и подчиняется принятым в нем ограничениям.

Для повышения производительности системы одни операции могут выполняться параллельно или одновременно, а другие — только последовательно. В этом случае для указания параллельности выполнения операции используется строка-свойство вида "{concurrency = *имя}",* где *имя* может принимать одно из следующих значений: последовательная (sequential), параллельная (concurrent), охраняемая (guarded). При этом придерживаются следующей семантики для данных значений:

- *последовательная* (sequential) — для данной операции необходимо обеспечить ее единственное выполнение в системе, одновременное выполнение других операций может привести к ошибкам или нарушениям целостности объектов класса.
- *параллельная* (concurrent) — данная операция в силу своих особенностей может выполняться параллельно с другими операциями в системе, при этом

параллельность должна поддерживаться на уровне реализации модели.

- *охраняемая* (guarded) — все обращения к данной операции должны быть строго упорядочены во времени с целью сохранения целостности объектов данного класса, при этом могут быть приняты дополнительные меры по контролю исключительных ситуаций на этапе ее выполнения.

В качестве примеров записи операций можно привести следующие обозначения отдельных операций:

- +создать() — может обозначать абстрактную операцию по созданию отдельного объекта класса, которая является общедоступной и не содержит формальных параметров. Эта операция не возвращает никакого значения после своего выполнения.
- +нарисовать(форма: Многоугольник = прямоугольник, цвет_заливки: Color — (0, 0, 255)) — может обозначать операцию по изображению на экране монитора прямоугольной области синего цвета, если не указываются другие значения в качестве аргументов данной операции.
- запросить_счет_клиента(номер_счета: Integer):Currency — обозначает операцию по установлению наличия средств на текущем счете клиента банка. При этом аргументом данной операции является номер счета клиента, который записывается в виде целого числа (например, "123456"). Результатом выполнения этой операции является некоторое число, записанное в принятом денежном формате (например, S 1,500.00).
- выдать_сообщение():{"Ошибка деления на ноль"} — смысл данной операции не требует пояснения, поскольку содержится в строке-свойстве операции. Данное сообщение может появиться на экране монитора в случае попытки деления некоторого числа на ноль, что недопустимо.

Отношения между классами

Кроме внутреннего устройства или структуры классов на соответствующей диаграмме указываются различные отношения между классами. При этом совокупность типов таких отношений фиксирована в языке UML и предопределена семантикой этих типов отношений. Базовыми отношениями или связями в языке UML являются:

- Отношение зависимости (*dependency relationship*)
- Отношение ассоциации (*association relationship*)
- Отношение обобщения (*generalization relationship*)
- Отношение реализации (*realization relationship*)

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

Отношение зависимости

Отношение зависимости в общем случае указывает некоторое семантическое отношение между двумя элементами модели или двумя множествами таких элементов, которое не является отношением ассоциации, обобщения или

реализации. Оно касается только самих элементов модели и не требует множества отдельных примеров для пояснения своего смысла. Отношение зависимости используется в такой ситуации, когда некоторое изменение одного элемента модели может потребовать изменения другого зависимого от него элемента модели.

Отношение зависимости графически изображается пунктирной линией между соответствующими элементами со стрелкой на одном из ее концов. На диаграмме классов данное отношение связывает отдельные классы между собой, при этом стрелка направлена от класса-клиента зависимости к независимому классу или классу-источнику (рис. 7.3). На данном рисунке изображены два класса: Класс_А и Класс_В, при этом Класс_В является источником некоторой зависимости, а Класс_А - клиентом этой зависимости.

Рис. 8.3. *Графическое изображение отношения зависимости на диаграмме классов*

Стрелка может помечаться необязательным, но стандартным ключевым словом в кавычках и необязательным индивидуальным именем. Для отношения зависимости предопределены ключевые слова, которые обозначают некоторые специальные виды зависимостей. Эти ключевые слова (стереотипы) записываются в кавычках рядом со стрелкой, которая соответствует данной зависимости. Примеры стереотипов для отношения зависимости представлены ниже:

- "access"-- служит для обозначения доступности открытых атрибутов и операций класса-источника для классов-клиентов;
- "bind" -- класс-клиент может использовать некоторый шаблон для своей последующей параметризации;
- "derive" -- атрибуты класса-клиента могут быть вычислены по атрибутам класса-источника;
- "import"-- открытые атрибуты и операции класса-источника становятся частью класса-клиента, как если бы они были объявлены непосредственно в нем;
- "refine" - указывает, что класс-клиент служит уточнением класса-источника в силу причин исторического характера, когда появляется дополнительная информация в ходе работы над проектом.

Отношение зависимости является наиболее общей формой отношения в языке UML. Все другие типы рассматриваемых отношений можно считать частным случаем данного отношения.

Отношение ассоциации

Отношение ассоциации соответствует наличию некоторого отношения между классами. Данное отношение обозначается сплошной линией с дополнительными специальными символами, которые характеризуют отдельные свойства конкретной ассоциации. В качестве дополнительных специальных символов могут использоваться имя ассоциации, а также имена и кратность классов-ролей ассоциации. Имя ассоциации является необязательным элементом ее обозначения. Если оно задано, то записывается с заглавной (большой) буквы рядом с линией соответствующей ассоциации.

В качестве простого примера отношения бинарной ассоциации рассмотрим отношение между двумя классами — классом "Компания" и классом "Сотрудник" (рис. 7.4). Они связаны между собой бинарной ассоциацией Работа, имя которой указано на рисунке рядом с линией ассоциации.

Рис8.4. Графическое изображение отношения бинарной ассоциации между классами

Следующий элемент обозначений — *кратность* отдельных классов, являющихся концами ассоциации. Кратность отдельного класса обозначается в виде интервала целых чисел, аналогично кратности атрибутов и операций классов. Интервал записывается рядом с концом ассоциации и для N-арной ассоциации означает потенциальное число отдельных экземпляров или значений кортежей этой ассоциации, которые могут иметь место, когда остальные N-1 экземпляров или значений классов фиксированы.

Так, для рассмотренного ранее примера (см. рис. 7.4) кратность "1" для класса "Компания" означает, что каждый сотрудник может работать только в одной компании. Кратность "1..*" для класса "Сотрудник" означает, что в каждой компании могут работать несколько сотрудников, общее число которых заранее неизвестно и ничем не ограничено.

Специальной формой или частным случаем отношения ассоциации является отношение агрегации, которое, в свою очередь, тоже имеет специальную форму — отношение композиции. Поскольку эти отношения имеют свои специальные обозначения и относятся к базовым понятиям языка UML, рассмотрим их последовательно.

Отношение агрегации

Отношение агрегации имеет место между несколькими классами в том случае, если один из классов представляет собой некоторую сущность, включающую в себя в качестве составных частей другие сущности.

Данное отношение имеет фундаментальное значение для описания структуры сложных систем, поскольку применяется для представления системных взаимосвязей типа "часть-целое". Раскрывая внутреннюю структуру системы, отношение агрегации показывает, из каких компонентов состоит система и как они связаны между собой. С точки зрения модели отдельные части системы могут выступать как в виде элементов,

так и в виде подсистем, которые, в свою очередь, тоже могут образовывать составные компоненты или подсистемы. Это отношение по своей сути описывает декомпозицию или разбиение сложной системы на более простые составные части, которые также могут быть подвергнуты декомпозиции, если в этом возникнет необходимость в последующем.

Графически отношение агрегации изображается сплошной линией, один из концов которой представляет собой незакрашенный внутри ромб. Этот ромб указывает на тот из классов, который представляет собой "целое". Остальные классы являются его "частями" (рис. 5.8).

Рис. 8.5.. *Графическое изображение отношения агрегации в языке UML*

Еще одним примером отношения агрегации может служить известное каждому деление персонального компьютера на составные части: системный блок, монитор, клавиатуру и мышь. Используя обозначения языка UML, компонентный состав ПК можно представить в виде соответствующей диаграммы классов (рис. 5.9), которая в данном случае иллюстрирует отношение агрегации.

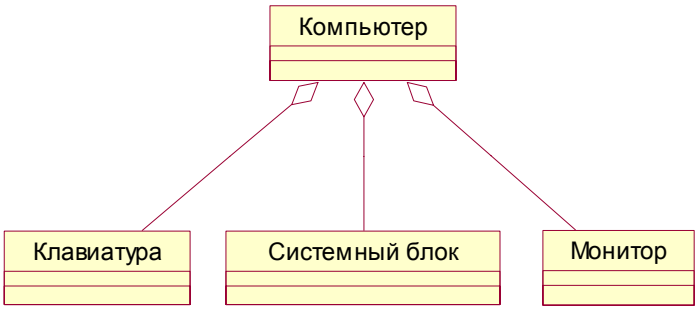


Рис. 8.6. *Диаграмма классов для иллюстрации отношения агрегации на примере ПК*

Пример определения отношений в диаграмме классов

Пример содержит три класса: Автор, страна и издательство.

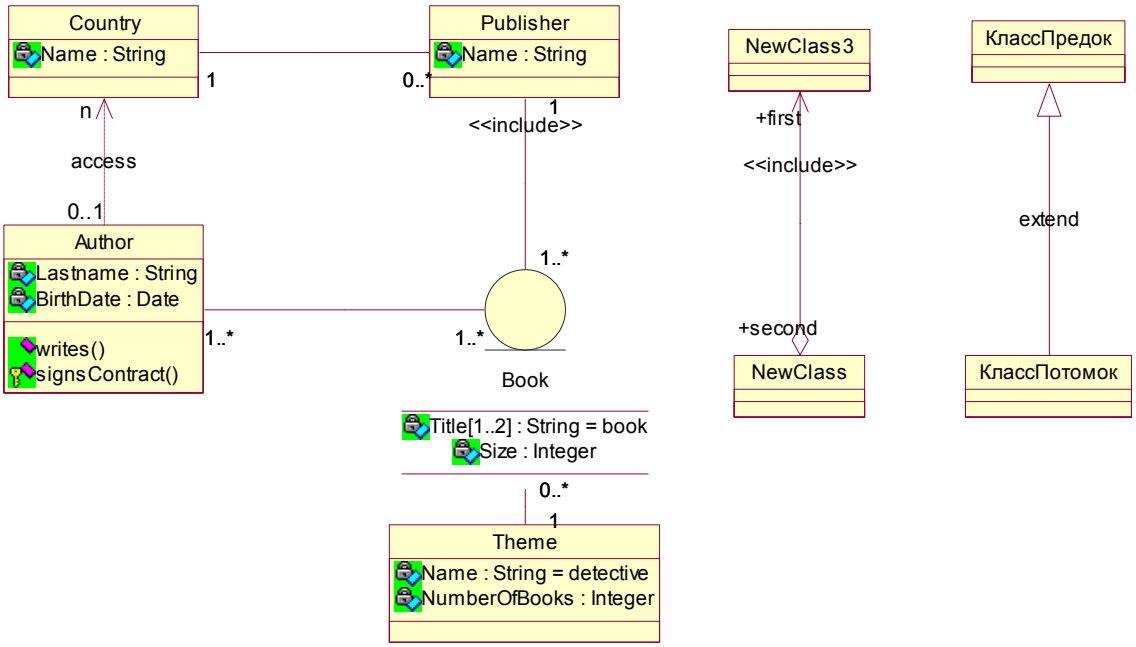


Рис.8.7. Примеры отношений между классами

Например, ассоциация, соединяющая классы "Преподаватель" и "Дисциплина", свидетельствует о том, что объекты одного класса связаны с объектами другого. Количество соединяемых объектов зависит от значения признака множественности ассоциации. Как показано на рис. 7.8. , в UML ассоциация помечается отрезком прямой, связывающим два класса.

Рис. 8.8. Обозначение связи ассоциации в UML

Как создать отношение ассоциации между классами

В системе Rational Rose:

1. Щелкнуть на пиктограмме Association панели инструментов Diagram. (Если пиктограмма на панели не отображается, расположить курсор мыши над панелью, щелкнуть правой кнопкой, выбрать элемент Customize контекстного меню и настроить панель инструментов средствами одноименного диалогового окна.)
2. В окне диаграммы щелкнуть на символе одного из ассоциируемых классов и, не отпуская кнопку мыши, построить линию связи, направленную к символу другого класса.

Отношение агрегирования

Чтобы выяснить, должна ли та или иная ассоциация трактоваться как связь агрегирования, можно воспользоваться следующими тестами.

- Применяется ли для описания отношения выражение "является частью"?
- Действительно ли некоторые операции над "целым" автоматически переносятся на его "части"? (Примером может служить операция удаления объекта"курс",

которая влечет необходимость изъятия всех соответствующих объектов "предложение курса".)

- Обладает ли отношение внутренне присущим свойством асимметрии, когда один соединяемый ею класс "подчиняется" другому?

Как создать отношение агрегирования классов

1. Щелкнуть на пиктограмме Aggregation панели инструментов Diagram. (Если пиктограмма на панели не отображается, расположить курсор мыши над панелью, щелкнуть правой кнопкой, выбрать элемент Customize контекстного меню и настроить панель инструментов средствами одноименного диалогового окна.)
2. В окне диаграммы щелкнуть на символе класса, представляющего "целое", и, не отпуская кнопку мыши, построить линию связи агрегирования, направленную к символу класса, служащего "частью целого".

Возвратные связи

Возможна ситуация, когда взаимодействовать приходится нескольким объектам одного и того же класса. Для отображения подобного требования на диаграмме классов используются возвратные связи (reflexive relationships) ассоциации или агрегирования. Возвратные связи обычно снабжаются названиями ролей, а не именами ассоциаций.

Как создать возвратную связь классов

1. Щелкнуть на пиктограмме Association (или Aggregation) панели инструментов Diagram.
2. В окне диаграммы классов щелкнуть на символе соответствующего класса и, не отпуская кнопку мыши, построить отрезок линии связи ассоциации (агрегирования).
3. Освободить кнопку мыши.
4. Щелкнуть на конце отрезка и построить очередной отрезок (возможно, замыкающий связь).
5. Обозначить имена ролей и признаки множественности для обоих концов связи.

Пример возвратной связи приведен на рис. 6.8.

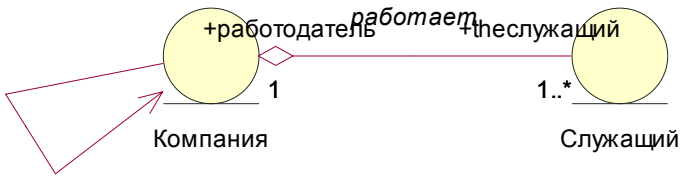


Рис. 8.9. Возвратная связь

9. ДИАГРАММА КОМПОНЕНТОВ (COMPONENT DIAGRAM)

Все рассмотренные ранее диаграммы отражали концептуальные аспекты построения модели системы и относились к логическому уровню представления.

Основное назначение логического представления состоит в анализе структурных и функциональных отношений между элементами модели системы. Однако для создания конкретной физической системы необходимо некоторым образом реализовать все элементы логического представления в конкретные материальные сущности. Для описания таких реальных сущностей предназначен другой аспект модельного представления, а именно физическое представление модели.

Чтобы пояснить отличие логического и физического представлений, рассмотрим в общих чертах процесс разработки некоторой программной системы. Ее исходным логическим представлением могут служить структурные схемы алгоритмов и процедур, описания интерфейсов и концептуальные схемы баз данных. Однако для реализации этой системы необходимо разработать исходный текст программы на некотором языке программирования (C++, Pascal, Basic.NET, Java). При этом уже в тексте программы предполагается такая организация программного кода, которая предполагает его разбиение на отдельные модули.

Таким образом, полный проект программной системы представляет собой совокупность моделей логического и физического представлений, которые должны быть согласованы между собой. В языке UML для физического представления моделей систем используются так называемые *диаграммы реализации* (*implementation diagrams*), которые включают в себя две отдельные канонические диаграммы: диаграмму компонентов и диаграмму развертывания.

Диаграмма компонентов, в отличие от ранее рассмотренных диаграмм, описывает особенности физического представления системы. Диаграмма компонентов позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых может выступать исходный, бинарный и исполняемый код. Во многих средах разработки модуль или компонент соответствует файлу. Пунктирные стрелки, соединяющие модули, показывают отношения взаимозависимости, аналогичные тем, которые имеют место при компиляции исходных текстов программ. Основными графическими элементами диаграммы компонентов являются компоненты, интерфейсы и зависимости между ними.

Диаграмма компонентов разрабатывается для следующих целей:

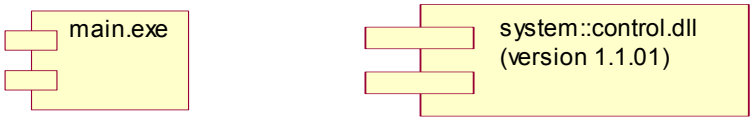
- Визуализации обшей структуры исходного кода программной системы.
- Спецификации исполнимого варианта программной системы.
- Обеспечения многократного использования отдельных фрагментов программного кода.
- Представления концептуальной и физической схем баз данных.

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. Диаграмма компонентов отражает общие зависимости между компонентами, рассматривая последние в качестве классификаторов.

Компоненты

Для представления физических сущностей в языке UML применяется специальный термин — ***компонент*** (component). Компонент реализует некоторый набор интерфейсов и служит для общего обозначения элементов физического представления модели. Для графического представления компонента может использоваться специальный символ — прямоугольник со вставленными слева двумя более мелкими прямоугольниками (рис. .1). Внутри объемлющего прямоугольника записывается имя компонента и, возможно, некоторая дополнительная информация. Изображение этого символа может незначительно варьироваться в зависимости от характера ассоциируемой с компонентом информации.

Как классификатор, компонент может иметь также свои собственные свойства, такие как атрибуты и операции.



а в

Рис. 9.1. Варианты графического изображения компонентов на диаграмме компонентов

Так, в первом случае (рис. 8.1, а) с компонентом уровня экземпляра связывается только его имя, а во втором (рис. 8.1, б) — дополнительно имя пакета и помеченное значение.

Имя компонента

Имя компонента подчиняется общим правилам именования элементов модели в языке UML и может состоять из любого числа букв, цифр и некоторых знаков препинания.

Если же компонент представляется на уровне экземпляра, то в качестве его имени записывается <имя компонента ':' имя типа>.

В качестве простых имен принято использовать имена исполняемых файлов (с указанием расширения exe после точки-разделителя), имена динамических библиотек (расширение dll), имена Web-страниц (расширение html), имена текстовых файлов (расширения txt или doc) или файлов справки (hlp), имена файлов баз данных (DB) или имена файлов с исходными текстами программ (расширения h, cpp для языка C++, расширение Java для языка Java), скрипты (pl, asp) и др.

Виды компонентов

Поскольку компонент как элемент физической реализации модели представляет отдельный модуль кода, иногда его комментируют с указанием дополнительных графических символов, иллюстрирующих конкретные особенности его реализации. Строго говоря, эти дополнительные обозначения для примечаний не специфицированы в языке UML. Однако их применение упрощает понимание диаграммы компонентов,

существенно повышая наглядность физического представления. Некоторые из таких общепринятых обозначений для компонентов изображены ниже (рис. 10.2).

В языке UML выделяют три вида компонентов.

- Во-первых, компоненты развертывания, которые обеспечивают непосредственное выполнение системой своих функций. Такими компонентами могут быть динамически подключаемые библиотеки с расширением dll (рис. 10.2, а), Web-страницы на языке разметки гипертекста с расширением html (рис. 10.2, б) и файлы справки с расширением hip (рис. 10.2, в).
- Во-вторых, компоненты-рабочие продукты. Как правило — это файлы с исходными текстами программ, например, с расширениями h или cpp для языка C++ (рис. 10.2, г).
- В-третьих, компоненты исполнения, представляющие исполнимые модули — файлы с расширением exe. Они обозначаются обычным образом.

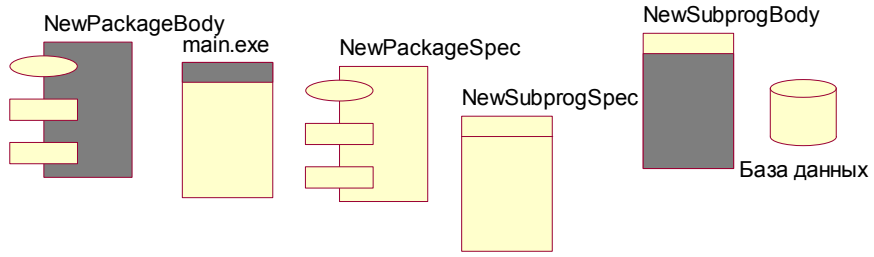


Рис. 9.2. *Варианты графического изображения компонентов на диаграмме компонентов*

Эти элементы иногда называют ***артефактами***, подчеркивая при этом их законченное информационное содержание, зависящее от конкретной технологии реализации соответствующих компонентов. Более того, разработчики могут для этой цели использовать самостоятельные обозначения, поскольку в языке UML нет строгой нотации для графического представления примечаний.

Другой способ спецификации различных видов компонентов — явное указание стереотипа компонента перед его именем. В языке UML для компонентов определены следующие стереотипы:

- ***Библиотека*** (library) — определяет первую разновидность компонента, который представляется в форме динамической или статической библиотеки.
- ***Таблица*** (table) — также определяет первую разновидность компонента, который представляется в форме таблицы базы данных.
- ***Файл*** (file) -- определяет вторую разновидность компонента, который представляется в виде файлов с исходными текстами программ.
- ***Документ*** (document) — определяет вторую разновидность компонента, который представляется в форме документа.
- Исполнимый (executable) — определяет третий вид компонента, который может исполняться в узле.

Интерфейсы

Следующим элементом диаграммы компонентов являются интерфейсы. Последние уже неоднократно рассматривались ранее, поэтому здесь будут отмечены те их

особенности, которые характерны для представления на диаграммах компонентов. Напомним, что в общем случае интерфейс графически изображается окружностью, которая соединяется с компонентом отрезком линии без стрелок (рис. 8.3, а). При этом имя интерфейса, которое обязательно должно начинаться с заглавной буквы "I", записывается рядом с окружностью. Семантически линия означает реализацию интерфейса, а наличие интерфейсов у компонента означает, что данный компонент реализует соответствующий набор интерфейсов.

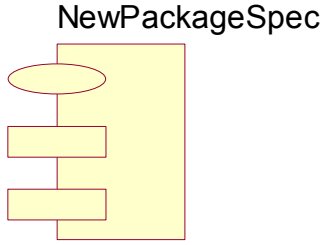


Рис 9.3 Графическое изображение интерфейсов на диаграмме компонентов

Другим способом представления интерфейса на диаграмме компонентов является его изображение в виде прямоугольника класса со стереотипом "интерфейс" и возможными секциями атрибутов и операций (рис. 10.3, б). Как правило, этот вариант обозначения используется для представления внутренней структуры интерфейса, которая может быть важна для реализации.

При разработке программных систем интерфейсы обеспечивают не только совместимость различных версий, но и возможность вносить существенные изменения в одни части программы, не изменяя другие ее части. Таким образом, назначение интерфейсов существенно шире, чем спецификация взаимодействия с пользователями системы (актерами).

Зависимости

Отношение зависимости на диаграмме компонентов изображается пунктирной линией со стрелкой, направленной от клиента (зависимого элемента) к источнику (независимому элементу).

Зависимости могут отражать связи модулей программы на этапе компиляции и генерации объектного кода. В другом случае зависимость может отражать наличие в независимом компоненте описаний классов, которые используются в зависимом компоненте для создания соответствующих объектов. Применительно к диаграмме компонентов зависимости могут связывать компоненты и импортируемые этим компонентом интерфейсы, а также различные виды компонентов между собой.

В первом случае рисуют стрелку от компонента-клиента к импортируемому интерфейсу (рис. 10.4). Наличие такой стрелки означает, что компонент не реализует соответствующий интерфейс, а использует его в процессе своего выполнения. Причем на этой же диаграмме может присутствовать и другой компонент, который реализует этот интерфейс. Так, например, изображенный ниже фрагмент диаграммы компонентов представляет информацию о том, что компонент с именем "main.exe" зависит от импортируемого интерфейса I Dialog, который, в свою очередь,

реализуется компонентом с именем "image.java". Для второго компонента этот же интерфейс является экспортируемым.

***Рис.9.4**. Фрагмент диаграммы компонентов с отношением зависимости*

Заметим, что изобразить второй компонент с именем "image.java" в форме варианта примечания нельзя именно в силу того факта, что этот компонент реализует интерфейс.

Другим случаем отношения зависимости на диаграмме компонентов является отношение между различными видами компонентов (рис. 10.5). Наличие подобной зависимости означает, что внесение изменений в исходные тексты программ или динамические библиотеки приводит к изменениям самого компонента. При этом характер изменений может быть отмечен дополнительно.

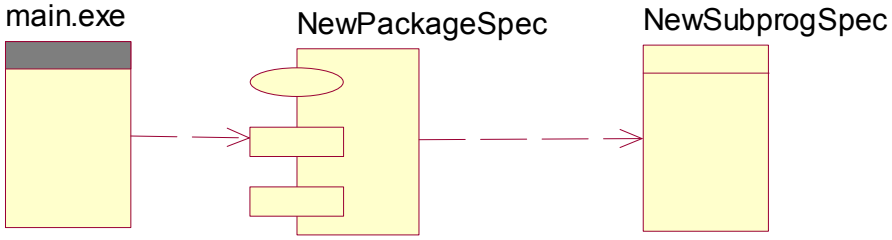


Рис. 9.5. Графическое изображение отношения зависимости между компонентами

Наконец, на диаграмме компонентов могут быть представлены отношения зависимости между компонентами и реализованными в них классами. Эта информация имеет важное значение для обеспечения согласования логического и физического представлений модели системы. Разумеется, изменения в структуре описаний классов могут привести к изменению компонента. Ниже приводится фрагмент зависимости подобного рода, когда некоторый компонент зависит от соответствующих классов.

UML позволяет генерировать код на выбранном языке, например, на C++. Как уже говорилось логическая модель системы отображается на диаграмме классов. Физическая модель программного продукта отображается на диаграмме компонентов. Система Rational Rose, предназначенная для реализации языка UML, имеет средства для преобразования графических диаграмм, описывающих структуру системы, в программный код. Технология получения программного кода подробно описана в лабораторных работах.

10. ДИАГРАММА РАЗВЕРТЫВАНИЯ (DEPLOYMENT DIAGRAM)

Физическое представление программной системы не может быть полным, если отсутствует информация о том, на какой платформе и на каких вычислительных средствах она реализована. Конечно, если разрабатывается простая программа, которая может выполняться локально на компьютере пользователя, не задействуя

никаких периферийных устройств и ресурсов, то в этом случае нет необходимости в разработке дополнительных диаграмм. Однако при разработке корпоративных приложений ситуация представляется совсем по-другому.

Как было отмечено *в разделе 8,* первой из диаграмм физического представления является диаграмма компонентов. Второй формой физического представления программной системы является *диаграмма развертывания* (синоним — диаграмма размещения). Она применяется для представления общей конфигурации и топологии распределенной программной системы и содержит распределение компонентов по отдельным узлам системы. Кроме того, диаграмма развертывания показывает наличие физических соединений маршрутов передачи информации между аппаратными устройствами, задействованными в реализации системы.

Диаграмма развертывания предназначена для визуализации элементов и компонентов программы, существующих лишь на этапе ее исполнения (runtime). При этом представляются только компоненты-экземпляры программы, являющиеся исполнимыми файлами или динамическими библиотеками. Те компоненты, которые не используются на этапе исполнения, на диаграмме развертывания не показываются. Так, компоненты с исходными текстами программ могут присутствовать только на диаграмме компонентов. На диаграмме развертывания они не указываются.

Диаграмма развертывания содержит графические изображения процессоров, устройств, процессов и связей между ними. В отличие от диаграмм логического представления, диаграмма развертывания является единой для системы в целом, поскольку должна всецело отражать особенности ее реализации. Эта диаграмма, по сути, завершает процесс ООАП для конкретной программной системы и ее разработка, как правило, является последним этапом спецификации модели.

Итак, перечислим цели, преследуемые при разработке диаграммы развертывания:

- Определить распределение компонентов системы по ее физическим узлам.
- Показать физические связи между всеми узлами реализации системы на этапе ее исполнения.
- Выявить узкие места системы и реконфигурировать ее топологию для достижения требуемой производительности.

Для обеспечения этих требований диаграмма развертывания разрабатывается совместно системными аналитиками, сетевыми инженерами и системотехниками.

Узел

Узел (node) представляет собой некоторый физически существующий элемент системы, обладающий некоторым вычислительным ресурсом, сканеры и манипуляторы.

Графически на диаграмме развертывания узел изображается в форме куба. Узел имеет собственное имя, которое указывается внутри этого графического символа. Сами узлы могут представляться как в качестве типов (рис. 11.1, а), так и в качестве экземпляров (рис. 11.1, б).

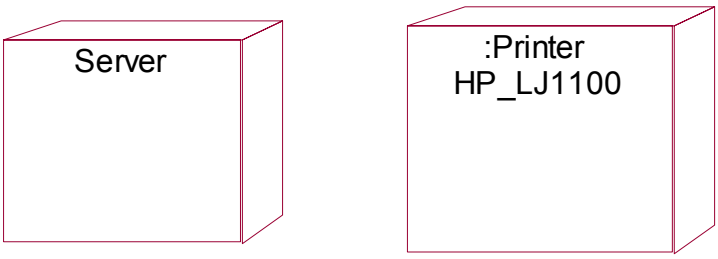


Рис. 10.1. Графическое изображение узла на диаграмме развертывания

Так же, как и на диаграмме компонентов, изображения узлов могут расширяться, чтобы включить некоторую дополнительную информацию о спецификации узла. Если дополнительная информация относится к имени узла, то она записывается под этим именем в форме помеченного значения (рис. 9.2).

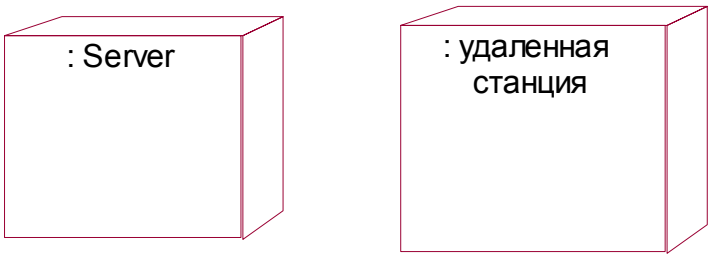


Рис. 10.2. Графическое изображение узла-экземпляра с дополнительной информацией в форме

Если необходимо явно указать компоненты, которые размещаются на отдельном узле, то это можно сделать двумя способами. Первый из них позволяет разделить графический символ узла на две секции горизонтальной линией. В верхней секции записывают имя узла, а в нижней секции — размещенные на этом узле компоненты (рис. П.3, а).

Второй способ разрешает показывать на диаграмме развертывания узлы с вложенными изображениями компонентов (рис. 9.3. б). Важно помнить, что в качестве таких вложенных компонентов могут выступать только исполняемые компоненты

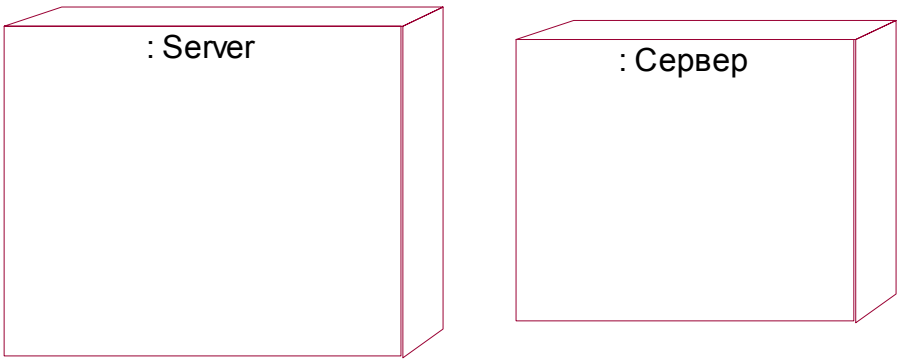


Рис. 10.3. *Варианты графического изображения узлов-экземпляров с размещаемыми на них компонентами*

Соединения

Кроме собственно изображений узлов на диаграмме развертывания указываются отношения между ними. В качестве отношений выступают физические соединения между узлами и зависимости между узлами и компонентами, изображения которых тоже могут присутствовать на диаграммах развертывания.

Соединения являются разновидностью ассоциации и изображаются отрезками линий без стрелок

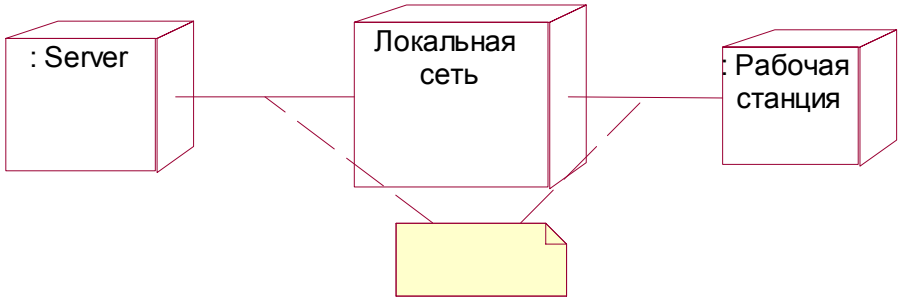


Рис. 10.4. *Фрагмент диаграммы развертывания с соединениями между узлами*

Кроме соединений на диаграмме развертывания могут присутствовать отношения зависимости между узлом и развернутыми на нем компонентами.

Итоги

Язык UML является удобным средством для описания процессов на первом и втором этапах жизненного цикла программного средства. Модели, построенные на UML, являются исходными данными для начала работ на третьем этапе – кодировании, то есть программирования.

Все современные системы разработки программного обеспечения, включая одну из последних разработок Microsoft Visual Studio.NET, имеют интерфейс с языком UML.