Технологии программирования. Компонентный подход

В. В. Кулямин

Технологии программирования. Компонентный подход

В. В. Кулямин

Аннотация курса

Курс посвящен технологическим проблемам разработки крупномасштабных программных систем и методам решения этих проблем. В нем рассказывается о современных способах организации разработки таких систем на основе компонентных технологий на примере Web-приложений с использованием технологий Java и .NET, а также дается введение в эти технологии. Читатели получат хорошее представление о методах разработки сложного программного обеспечения, об использовании современных подходов к промышленной разработке программ и о способах построения качественных и гибких программ в условиях жестких ограничений на проекты разработки. Читатели также познакомятся с элементами технологий создания распределенных приложений на платформах J2EE и .NET.

Содержание

Список рисунков

Список таблиц

Предисловие

Предлагаемый вниманию читателей курс лекций построен на основе специального курса, читающегося на факультете Вычислительной математики и кибернетики МГУ им. М. В. Ломоносова. Он был задуман как замена и некоторая модернизация курса по технологии программирования, долгое время читавшегося на ВМиК ныне покойным Е. А. Жоголевым и являвшегося введением в инженерию программного обеспечения (ПО).

Но, кроме введения в программную инженерию, автор посчитал необходимым в рамках этого же курса дать слушателям базовый набор знаний по современным компонентным технологиям разработки Web-приложений.

Таким образом, данные лекции преследуют две цели — введение в инженерию ПО в целом как инженерную дисциплину, рассматривающую методы построения сложных программных систем вообще, и введение в современные технологии разработки наиболее широко востребованного сейчас вида компонентных распределенных программных систем. Автор глубоко убежден в необходимости такого объединения. Компонентные технологии, хотя и продолжают активно развиваться, уже сейчас являются стержневым элементом современной программной инженерии. Они не только лежат в основе методов разработки прикладного ПО, но проникают и в системное программирование. Знание основных принципов работы широко используемых систем промежуточного уровня (middleware) и умение строить приложения на их основе в настоящий момент становится таким же необходимыми элементом образования профессионального разработчика программ, как и знания в области операционных систем, компиляторов и систем управления базами данных.

Данный курс не претендует на полное изложение знаний в области программной инженерии. Это скорее попытка обзора основных технологических элементов, из которых складывается процесс промышленной разработки сложных программ в современной практике. Рассматриваются модели жизненного цикла ПО в целом, деятельности, связанные с анализом предметной области и требований, обеспечением качества ПО, разработкой архитектуры ПО и отдельных компонентов, разработкой пользовательского интерфейса, а также современные языки компонентно-ориентированной разработки и проблемы управления проектами разработки ПО.

В каждой из этих областей представлены основные задачи, встающие перед участниками разработки, и одна-две техники, используемые для их решения на практике. Кроме того, рассматриваются такие важные элементы системы знаний опытного разработчика программ, как образцы анализа, проектирования и процессов, компонентные технологии разработки, техники создания распределенных приложений.

В результате кому-то может показаться, что представленный материал чересчур велик, а кому-то — что он слишком поверхностно освещает перечисленные области. И то, и другое может быть близко к истине, поскольку не все задуманное удается. Однако, по мнению автора, такая структура курса хорошо отражает современное состояние инженерии программного обеспечения — бурно развивающейся инженерной дисциплины, активно впитывающей в себя и использующей результаты множества смежных областей, от математики и системного анализа до микроэкономики, когнитивной психологии и социологии малых сообществ. Вместить материал всех ее отдельных областей в одну книгу уже невозможно — любой из них можно посвятить отдельный курс. Читателям, желающим более глубоко ознакомиться с их содержанием, автор советует обратиться к литературе, списками которой он постарался дополнить каждую лекцию.

Схема зависимостей между лекциями курса представлена на Рис. 1. Сплошными стрелками показано, на материал каких лекций опирается данная, а пунктирные стрелки изображают более слабые связи, которые могут быть проигнорированы при первом изучении лекций.

Автор желает читателям, приступающим к изучению инженерии ПО для использования полученных знаний на практике, успехов на этом поприще, а тем, кто просто интересуется этим предметом, — чтобы их интерес не был остужен разочарованием, а принес бы достойные плоды в каком угодно виде.

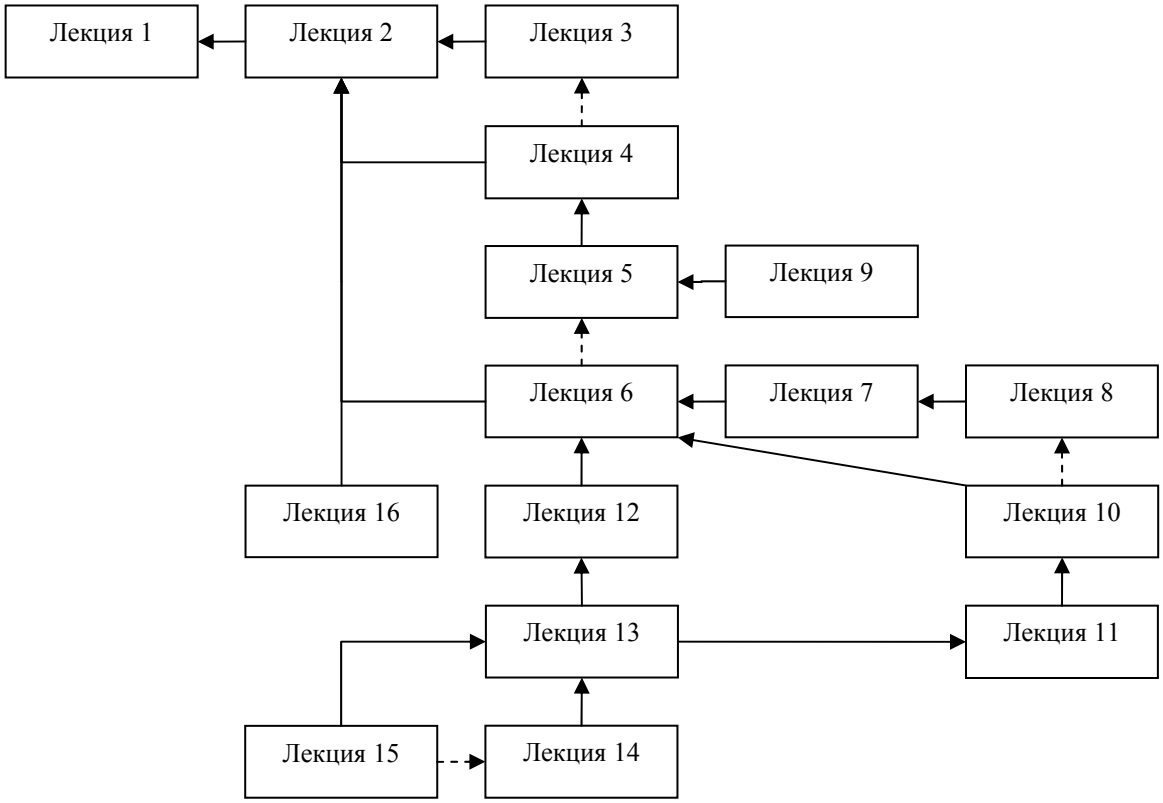


Рисунок 1. Схема зависимостей между лекциями.

Автор хотел бы также поблагодарить за разнообразную помощь в создании этого курса следующих людей: А. К. Петренко за саму идею курса, его же и В. П. Иванникова за предоставленную возможность сделать такой курс и поддержку при работе над ним, А. В. Баранцева за ценные советы по содержанию отдельных лекций, О. Л. Петренко за собранные ею материалы для Лекции 4 и многочисленные полезные замечания по другим лекциям, А. А. Сортова за вычитывание всех лекций, множество замечаний и устранение огромного количества ошибок, представителя издательства А. В. Шкреда за проявленное понимание и терпение, с которым он относился к постоянным задержкам со стороны автора при подготовке курса.

Лекция 1. Проблемы разработки сложных программных систем

Аннотация

Рассматривается понятие сложной программы и отличия сложных программ от простых. Приводятся основные проблемы разработки сложных программ. В приложении к программной инженерии формулируются основные принципы работы со сложными системами, применимые к широкому кругу задач.

Ключевые слова

Сложное программное обеспечение, программная инженерия, компонентная разработка ПО, абстракция и уточнение, выделение модулей, разделение ответственности, переиспользование, адекватность интерфейса, полнота интерфейса, минимальность интерфейса, простота интерфейса.

Текст лекции

Программы «большие» и «маленькие»

Основная тема данного курса — методы разработки *«больших» и сложных программ*.

Каждый человек хоть раз написавший какую-либо программу, достаточно хорошо может представить себе, как разработать «небольшую» программу, решающую обычно одну конкретную несложную задачу и предназначенную, чаще всего, для использования одним человеком или узкой группой людей.

Примером может служить программа, вычисляющая достаточно много (но не слишком, не больше 30000) знаков числа π.

Воспользуемся следующими формулами.

$\arctan(x) = x - x^3/3 + x^5/5 - x^7/7 + \ldots + (-1)^n x^{2n+1}/(2n+1) + O(x^{2n+3})$

$\pi/4 = \arctan(1) = 4*\arctan(1/5) - \arctan(1/239)$

Соответсвующая программа на языке Java может выглядеть примерно так.

```
public class PiCalculator
{
  //Позволяет при вычислениях с повышенной точностью умножать и делить на числа
  //  <= 42949 = ( 2^32 mod CLUSTER_SIZE )
  //Эта константа должна быть степенью 10 для простоты представления чисел.
  private final static long CLUSTER_SIZE = 100000;

  //Определенное значение этого поля позволяет сосчитать
  //  numberOfClusters * lg( CLUSTER_SIZE )
  //точных цифр.
  private static int numberOfClusters;

  private static void print(long a[])
  {
    for(int i = 0; i < numberOfClusters + 1; i++)
    {
      if (i == 0) System.out.print("" + a[i] + '.');
      else
      {
        StringBuffer s = new StringBuffer();
        long z = CLUSTER_SIZE/10;

        while(z > 0)
        {
          if (z > a[i]) { s.append(0); z /= 10; }
          else          break;
        }
        if (z != 0) s.append(a[i]);
        System.out.print(s);
      }
```

```
  }
  System.out.println();
}

private static void lndiv(long a[], int n)
{
  for(int i = 0; i < numberOfClusters + 1; i++)
  {
    if (i != numberOfClusters)
    {
      a[i+1] += (a[i]%n)*CLUSTER_SIZE;
      a[i] /= n;
    }
    else a[i] /= n;
  }
}

private static void lnadd(long a[], long b[])
{
  for(int i = numberOfClusters; i >= 0; i--)
  {
    if (i != 0)
    {
      a[i-1] += (a[i] + b[i])/CLUSTER_SIZE;
      a[i] = (a[i] + b[i])%CLUSTER_SIZE;
    }
    else
      a[i] = (a[i] + b[i])%CLUSTER_SIZE;
  }
}

private static void lnsub(long a[], long b[])
{
  for(int i = numberOfClusters; i >= 0; i--)
  {
    if (i != 0)
    {
      if (a[i] < b[i]) { b[i-1]++; a[i] += CLUSTER_SIZE; }
      a[i] -= b[i];
    }
    else
      a[i] -= b[i];
  }
}

public static void main (String[] args)
{
  int i, j, numberOfDigits = 100, numberOfSteps;

  if (args.length > 0) numberOfDigits = Integer.parseInt(args[0]);

  numberOfSteps = (int)(((numberOfDigits + 1)/(Math.log(5)/Math.log(10)) - 1)/2+1);
  numberOfClusters = (int)(numberOfDigits/(Math.log(CLUSTER_SIZE)/Math.log(10))+1);

  long a1[] = new long[numberOfClusters + 1];
  long b1[] = new long[numberOfClusters + 1];
  long c1[] = new long[numberOfClusters + 1];
  long a2[] = new long[numberOfClusters + 1];
  long b2[] = new long[numberOfClusters + 1];
  long c2[] = new long[numberOfClusters + 1];

  a1[0] = 16;
  a2[0] = 4;
  lndiv(a1, 5);
  lndiv(a2, 239);
```

```
    System.arraycopy(a1, 0, c1, 0, numberOfClusters + 1);
    System.arraycopy(a2, 0, c2, 0, numberOfClusters + 1);

    for(j = 1; j < numberOfSteps; j++)
    {
      lndiv(a1, 25);
      lndiv(a2, 239);
      lndiv(a2, 239);

      System.arraycopy(a1, 0, b1, 0, numberOfClusters + 1);
      System.arraycopy(a2, 0, b2, 0, numberOfClusters + 1);

      lndiv(b1, 2*j+1);
      lndiv(b2, 2*j+1);

      if (j%2 == 0) { lnadd(c1, b1); lnadd(c2, b2); }
      else          { lnsub(c1, b1); lnsub(c2, b2); }
    }

    lndiv(a1, 25);
    lndiv(a1, 2*numberOfSteps + 1);

    System.out.println("Оценка точности результата:");
    print(a1);

    lnsub(c1, c2);

    System.out.println("Результат:");
    print(c1);
  }
}
```

Данная программа — «небольшая», как по размерам (~150 строк), так и по другим признакам:

- Она решает одну четко поставленную задачу (выдает десятичные цифры числа π) в хорошо известных ограничениях (не более 30000 цифр), к тому же, не очень существенную для какой-либо практической или исследовательской деятельности.
- Неважно, насколько быстро она работает — на вычисление 30000 цифр уходит не более получаса даже на устаревших компьютерах, и этого вполне достаточно.
- Ущерб от неправильной работы программы практически нулевой (за исключением возможности обрушения ею системы, в которой выполняются и другие, более важные задачи).
- Не требуется дополнять программу новыми возможностями, практически никому не нужно разрабатывать ее новые версии или исправлять найденные ошибки.
- В связи со сказанным выше не очень нужно прилагать к программе подробную и понятную документацию — для человека, который ею заинтересуется, не составит большого труда понять, как ею пользоваться, просто по исходному коду.

Сложные или ***«большие» программы,*** называемые также ***программными системами***, ***программными комплексами***, ***программными продуктами,*** отличаются от «небольших» не столько по размерам (хотя обычно они значительно больше), сколько по наличию дополнительных 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, возможно, также

специальную документацию для администраторов, 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 (реинжиниринг).
- 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.

Примером «большой» программы может служить стандартная библиотека классов Java, входящая в Java Development Kit [1].

Строго говоря, ни одно из указанных свойств не является обязательным для того, чтобы программу можно было считать «большой», но при наличии двух-трех из них достаточно уверенно можно утверждать, что она «большая».

На основании некоторых из перечисленных свойств можно сделать вывод, что «большая» программа или программная система чаще всего представляет собой не просто код или исполняемый файл, а включает еще и набор проектной и пользовательской документации.

Для разработки программных систем требуются особые методы — как уже говорилось, их нельзя написать «нахрапом». Изучением организационных, инженерных и технических аспектов создания ПО, включая методы разработки, занимается дисциплина, называемая ***программной инженерией***. Большая часть трудностей при разработке программных систем связана с организацией экономически эффективной совместной работы многих людей, приводящей к практически полезному результату. Это требует рассмотрения следующих аспектов.

- Над программой обычно работает много людей, иногда географически удаленных друг от друга и из различных организаций. Их работа должна быть организована так, чтобы затраты на разработку были бы покрыты доходами от продаж и предоставления услуг, связанных с полученной программой. В затраты входят зарплаты разработчиков, затраты на закупленное оборудование и программные инструменты разработки, на приобретение лицензий и патентование собственных решений, часто еще и затраты на исследование потребностей клиентов, проведение рекламы и другой маркетинговой деятельности.
- Значимые доходы могут быть получены, только если программа будет предоставлять пользователям в реальных условиях их работы такие возможности, что они готовы будут заплатить за это деньги (которым, заметим, без труда можно найти другие полезные применения). Для этого нужно учесть множество аспектов. Доходы от продаж значительно снизятся, если многие из пользователей не смогут воспользоваться программой только потому, что в их компьютерах процессоры слишком медленные или мало оперативной памяти, или потому что данные к системе часто поступают в искаженном виде и она не может их обработать, или потому что они привыкли работать с графическим интерфейсом, а система требует ввода из командной строки, и т.п.

Важно отметить, что *практически полезная* сложная программная система не обязательно является *«правильной»*.

Большинство опытных разработчиков и исследователей считают, что практически значимые программные системы всегда содержат ошибки. При переходе от «небольших» программ к «большим» понятие «правильной» программы становится практически бессмысленным. Про программную систему, в отличие от приведенной выше программы вычисления числа π, нельзя утверждать, что она «правильная», т.е. всегда правильно решает все поставленные перед ней задачи. Этот факт связан как с практической невозможностью полного доказательства или проверки этого, так и с тем, что смысл существования программной системы — удовлетворение потребностей и запросов большого количества различных заинтересованных лиц. А эти потребности не только нечетко определены, различны для разных групп пользователей и иногда противоречивы, но и значительно изменяются с течением времени.

В связи с этим, вместо рассмотрения *«правильных»* и *«неправильных»* программных систем, в силу практического отсутствия первых, рассматривают *«достаточно качественные»* и *«недостаточно качественные»*.

Поэтому и основные проблемы разработки сложных программных систем связаны с нахождением разумного компромисса между затратами на разработку и качеством ее результата. В затраты входят все виды используемых ресурсов, из которых наиболее важны затрачиваемое время, бюджет проекта и используемый персонал. Удовлетворение пользователей от работы с программой (а, следовательно, доходы от ее продаж и предоставления дополнительных услуг) и удовлетворение разработчиков от ее создания определяются качеством программы, которое включает в себя такие аспекты, как набор предоставляемых возможностей, надежность, удобство использования, гибкость, удобство внесения изменений и исправления ошибок. Более подробно понятие качественного программного обеспечения будет обсуждаться в одной из следующих лекций.

Часто программное обеспечение (ПО) нельзя рассматривать отдельно от программно-аппаратной системы, куда оно входит в качестве составной части. Изучением вопросов, связанных с разработкой и эксплуатацией программно-аппаратных систем занимается ***системная инженерия***. В ее рамки попадает огромное количество проблем, связанных с аппаратной частью систем и обеспечением нужного уровня интеграции программной и аппаратной составляющих. Мы только изредка будем затрагивать вопросы, касающиеся системной инженерии в целом, в основном ограничиваясь аспектами, относящимися непосредственно к ПО.

В данном курсе будут рассматриваться различные подходы к решению проблем разработки, связанных с обеими составляющими дилеммы «ресурсы-качество» при создании сложных программ. Для изложения этих подходов вводится система понятий, относящихся к программным системам и процессам их создания и позволяющих эффективно разрабатывать такие системы, оценивать и планировать их свойства. В их число входят такие понятия, как *жизненный цикл ПО*, *качество ПО*, *процесс разработки ПО*, *требования к ПО*, *архитектура ПО*, *образцы проектирования* и пр.

Кроме того, особое внимание в курсе уделяется одному из подходов к разработке сложного ПО, ***компонентной разработке***, предлагающей строить такие системы последовательно из отдельных элементов — компонентов, каждый из которых, в свою очередь, может рассматриваться как отдельная программная система. Курс дает введение в современные компонентные технологии разработки программных систем на основе платформ J2EE и .NET.

Проблемы, связанные с управлением ресурсами разработки, в том числе — планированием отдельных действий во времени, созданием эффективных команд разработчиков, относятся к *управлению проектами*, которому будет посвящена последняя лекция курса.

На основании опыта конструирования больших систем разработаны так называемые *технологические процессы*, содержащие достаточно детальные описания разных аспектов их создания и эксплуатации. Эти описания дают ответы на вопросы о том, как должна вестись разработка, какие лица должны в ней участвовать и на каких этапах, какие виды деятельности и в какой последовательности должны выполняться, какие документы являются их входными данными и какие документы, модели, другие части программной системы должны быть подготовлены в результате каждой отдельной работы. Элементы таких методик будут

упоминаться на всем протяжении курса. Также будут рассматриваться отраслевые стандарты, содержащие описание выработанных на основе большого количества реальных проектов подходов к построению сложных программных систем.

При практической разработке больших систем, однако, стоит помнить, что все общеметодические рекомендации имеют границы применимости, и чем детальнее они определяют действия разработчиков, тем вероятнее, что что-то пойдет не так, как это предусматривается авторами методик. Кроме того, огромное количество вспомогательных по сути документов, оформление которых часто требуется подобными методиками, иногда затрудняет понимание основных целей проекта, принимаемых в его ходе решений и сути происходящего в нем. Оно также может приводить к имитации усердной работы при отсутствии реальных продвижений к нужным результатам.

Протест сообщества разработчиков против подобной бюрократизации разработки программ и попыток механического использования теоретических рекомендаций вылился в популярное сейчас движение *живой разработки ПО* (Agile Software Development). Одним из примеров «живого» процесса разработки является набор техник, известный как *экстремальное программирование* (Extreme Programming, XP). Некоторые аспекты этих подходов также будут рассмотрены в данном курсе.

Принципы работы со сложными системами

Помимо методических рекомендаций, при конструировании больших систем часто используются прагматические принципы работы со сложными системами вообще. Они играют значительную роль в выработке качественных технических решений в достаточно широком контексте. Эти принципы позволяют распределять работы между участвующими в проектах людьми с меньшими затратами на обеспечение их взаимодействия и акцентировать внимание каждого из участников на наиболее существенных для его части работы характеристиках системы. К таким принципам относятся использование *абстракции* и *уточнения*, *модульная разработка* и *переиспользование*.

- **Абстракция (abstraction) и уточнение (refinement).**
  Абстракция является универсальным подходом к рассмотрению сложных вещей. Интеллект одного человека достаточно ограничен и просто не в силах иметь дело сразу со всеми элементами и свойствами систем большой сложности. Известно, что человеку крайне тяжело держать в голове одновременно десяток-полтора различных мыслей, а в современных системах число различных существенных аспектов доходит до сотен. Для того чтобы как-то все-таки работать с такими системами, мы пользуемся своей возможностью *абстрагироваться*, т.е. отвлекаться от всего, что несущественно для достижения поставленной в данной момент частной цели и не влияет на те аспекты рассматриваемого предмета, которые для этой цели важны.

  Чтобы перейти от абстрактного представления к более конкретному, используется обратный процесс последовательного *уточнения.* Рассмотрев систему в каждом аспекте в отдельности, мы пытаемся объединить результаты анализа, добавляя аспекты по одному и обращая при этом внимание на возможные взаимные влияния и возникающие связи между элементами, выявленными при анализе отдельных аспектов.
  Абстракция и уточнение используются, прежде всего, для получения работоспособных решений, гарантирующих нужные свойства результирующей системы.

  Пример абстракции и уточнения.
  Систему хранения идентификаторов пользователей Интернет-магазина можно представить как множество целых чисел, забыв о том, что эти числа — идентификаторы пользователей, и о том, что все это как-то связано с Интернет-магазином.
  Затем описанную модель системы хранения идентификаторов пользователей Интернет-магазина можно уточнить, определив конкретную реализацию множества чисел, например, на основе сбалансированных красно-черных деревьев (см. [2], раздел 14, глава III и JDK классы `java.util.TreeSet` и `java.util.TreeMap`).

Другой пример.
Рассматривая задачу передачи данных по сети, можно временно абстрагироваться от большинства проблем организации связи и заниматься только одним аспектом — организацией надежной передачи данных в нужной последовательности. При этом можно предполагать, что мы как-то умеем передавать данные между двумя компьютерами в сети, хотя, быть может, и с потерями и с нарушением порядка их прибытия по сравнению с порядком отправки. Установленные ограничения выделяют достаточно узкий набор задач. Любое их решение представляет собой некоторый *протокол передачи данных транспортного уровня*, т.е. нацеленный именно на надежную упорядоченную доставку данных. Выбирая такой протокол из уже существующих, например, TCP, или разрабатывая новый, мы производим уточнение исходной общей задачи передачи данных.

Другой способ уточнения — перейти к рассмотрению протоколов, обеспечивающих исходные условия для нашей первой абстракции, т.е. возможность вообще что-то передавать по сети. При этом возникают протоколы нижележащих уровней — *сетевого* (отвечают за организацию связи между не соединенными непосредственно компьютерами при наличии между ними цепи машин, соединенных напрямую), *канального* (такие протоколы отвечают за определение формата передаваемых данных и надежность передачи отдельных элементов информации между двумя физически соединенными компьютерами) и *физического* (отвечают за определение физического носителя передаваемого сигнала и правильность интерпретации таких сигналов обеими машинами, в частности, за конкретный способ передачи битов с помощью электрических сигналов или радиоволн).

- **Модульность (modularity).**
Модульность — принцип организации больших систем в виде наборов подсистем, модулей или компонентов. Этот принцип предписывает организовывать сложную систему в виде набора более простых систем — модулей, взаимодействующих друг с другом через четко определенные интерфейсы. При этом каждая задача, решаемая всей системой, разбивается на более простые, решаемые отдельными модулями подзадачи, решение которых, будучи скомбинировано определенным образом, дает в итоге решение исходной задачи. После этого можно отдельно рассматривать каждую подзадачу и модуль, ответственный за ее решение, и отдельно — вопросы интеграции полученного набора модулей в целостную систему, способную решать исходные задачи.

  Выделение четких интерфейсов для взаимодействия упрощает интеграцию, позволяя проводить ее на основе явно очерченных возможностей этих интерфейсов, без обращения к многочисленным внутренним элементам модулей, что привело бы к росту сложности.

  Пример.
  Примером разбиения на модули может служить структура пакетов и классов библиотеки JDK. Классы, связанные с основными сущностями языка Java и виртуальной машины, собраны в пакете `java.lang`. Вспомогательные широко применяемые в различных приложениях классы, такие, как коллекции, представления даты и пр., собраны в `java.util`. Классы, используемые для реализации потокового ввода-вывода — в пакете `java.io`, и т.д.
  Интерфейсом класса служат его общедоступные методы, а интерфейсом пакета — его общедоступные классы.

  Другой пример.
  Другой пример модульности — принятый способ организации протоколов передачи данных. Мы уже видели, что удобно выделять несколько уровней протоколов, чтобы на каждом решать свои задачи. При этом надо определить, как информация передается от машины к машине при помощи всего этого многоуровневого механизма. Обычное решение таково: для каждого уровня определяется способ передачи информации с или на верхний уровень — предоставляемые данным уровнем службы. Точно так же определяется, в каких службах нижнего уровня нуждается верхний, т.е. как передать данные на нижний уровень и получить их оттуда. После этого каждый протокол на данном уровне может быть сформулирован в терминах обращений к нижнему уровню и должен реализовать операции-

службы, необходимые верхнему. Это позволяет заменять протокол-модуль на одном уровне без внесения изменений в другие.
Хорошее разбиение системы на модули — непростая задача. При ее выполнении привлекаются следующие дополнительные принципы.

- **Выделение интерфейсов и сокрытие информации.**
  Модули должны взаимодействовать друг с другом через четко определенные *интерфейсы* и скрывать друг от друга внутреннюю информацию — внутренние данные, детали реализации интерфейсных операций.
  При этом интерфейс модуля обычно значительно меньше, чем набор всех операций и данных в нем.
  Например, класс `java.util.Queue<type E>`, реализующий функциональность очереди элементов типа `E`, имеет следующий интерфейс.

`E element()`	Возвращает элемент, стоящий в голове очереди, не изменяя ее. Создает исключение `NoSuchElementException`, если очередь пуста.
`boolean` `offer(E o)`	Вставляет, если возможно, данный элемент в конец очереди. Возвращает `true`, если вставка прошла успешно, `false` — иначе.
`E peek()`	Возвращает элемент, стоящий в голове очереди, не изменяя ее. Возвращает `null`, если очередь пуста.
`E poll()`	Возвращает элемент, стоящий в голове очереди, и удаляет его из очереди. Возвращает `null`, если очередь пуста.
`E remove()`	Возвращает элемент, стоящий в голове очереди, и удаляет его из очереди. Создает исключение `NoSuchElementException`, если очередь пуста.

  Внутренние же данные и операции одного из классов, реализующих данный интерфейс, — `PriorityBlockingQueue<E>` — достаточно сложны. Этот класс реализует очередь с эффективной синхронизацией операций, позволяющей работать с таким объектом нескольким параллельным потокам без лишних ограничений на их синхронизацию. Например, один поток может добавлять элемент в конец непустой очереди, а другой в то же время извлекать ее первый элемент.

```
package java.util.concurrent;
import java.util.concurrent.locks.*;
import java.util.*;
public class PriorityBlockingQueue<E> extends AbstractQueue<E>
    implements BlockingQueue<E>, java.io.Serializable {
    private static final long serialVersionUID = 5595510919245408276L;
    private final PriorityQueue<E> q;
    private final ReentrantLock lock = new ReentrantLock(true);
    private final ReentrantLock.ConditionObject notEmpty =
lock.newCondition();
    public PriorityBlockingQueue() { ... }
    public PriorityBlockingQueue(int initialCapacity) { … }
    public PriorityBlockingQueue(int initialCapacity,
                                 Comparator<? super E> comparator) { … }
    public PriorityBlockingQueue(Collection<? extends E> c) { ... }
    public boolean add(E o) { ... }
    public Comparator comparator() { … }
    public boolean offer(E o) { … }
    public void put(E o) { … }
    public boolean offer(E o, long timeout, TimeUnit unit) { … }
    public E take() throws InterruptedException { … }
```

```
    public E poll() { … }
    public E poll(long timeout, TimeUnit unit) throws InterruptedException {
… }
    public E peek() { … }
    public int size() { … }
    public int remainingCapacity() { … }
    public boolean remove(Object o) { … }
    public boolean contains(Object o) { … }
    public Object[] toArray() { … }
    public String toString() { … }
    public int drainTo(Collection<? super E> c) { … }
    public int drainTo(Collection<? super E> c, int maxElements) { … }
    public void clear() { … }
    public <T> T[] toArray(T[] a) { … }
    public Iterator<E> iterator() { … }
    private class Itr<E> implements Iterator<E> {
        private final Iterator<E> iter;
        Itr(Iterator<E> i) { … }
        public boolean hasNext() { … }
        public E next() { … }
        public void remove() { … }
    }
    private void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException { … }
}
```

- **Адекватность, полнота, минимальность и простота интерфейсов.**
  Этот принцип объединяет ряд свойств, которыми должны обладать хорошо спроектированные интерфейсы.
  - ***Адекватность интерфейса*** означает, что интерфейс модуля дает возможность решать именно те задачи, которые нужны пользователям этого модуля.
    Например, добавление в интерфейс очереди метода, позволяющего получить любой ее элемент по его номеру в очереди, сделало бы этот интерфейс не вполне адекватным — он превратился бы почти в интерфейс списка, который используется для решения других задач. Очереди же используются там, где полная функциональность списка не нужна, а реализация очереди может быть сделана более эффективной.
  - ***Полнота интерфейса*** означает, что интерфейс позволяет решать все значимые задачи в рамках функциональности модуля.
    Например, отсутствие в интерфейсе очереди метода `offer()` сделало бы его бесполезным — никому не нужна очередь, из которой можно брать элементы, а класть в нее ничего нельзя.
    Более тонкий пример — методы `element()` и `peek()`. Нужда в них возникает, если программа не должна изменять очередь, и в то же время ей нужно узнать, какой элемент лежит в ее начале. Отсутствие такой возможности потребовало бы создавать собственное дополнительное хранилище элементов в каждой такой программе.
  - ***Минимальность интерфейса*** означает, что предоставляемые интерфейсом операции решают различные по смыслу задачи и ни одну из них нельзя реализовать с помощью всех остальных (или же такая реализация довольно сложна и неэффективна).
    Представленный в примере интерфейс очереди не минимален — методы `element()` и `peek()`, а также `poll()` и `remove()` можно выразить друг через друга.

Минимальный интерфейс очереди получился бы, например, если выбросить пару методов `element()` и `remove()`.
Большое значение минимальности интерфейса уделяют, если размер модулей оказывает сильное влияние на производительность программы. Например, при проектировании модулей операционной системы — чем меньше она занимает места в памяти, тем больше его останется для приложений, непосредственно необходимых пользователям.
При проектировании библиотек более высокого уровня имеет смысл не делать интерфейс минимальным, давая пользователям этих библиотек возможности для повышения производительности и понятности их программ. Например, часто бывает полезно реализовать действия «проверить, что элемент не принадлежит множеству, и, если нет, добавить его» в одном методе, не заставляя пользователей каждый раз сначала проверять принадлежность элемента множеству, а затем уже добавлять его.

- ***Простота интерфейса*** означает, что интерфейсные операции достаточно элементарны и не представимы в виде композиций некоторых более простых операций на том же уровне абстракции, при том же понимании функциональности модуля.
  Скажем, весь интерфейс очереди можно было бы свести к одной операции `Object queue(Object o, boolean remove)`, которая добавляет в очередь объект, указанный в качестве первого параметра, если это не **`null`**, а также возвращает объект в голове очереди (или **`null`**, если очередь пуста) и удаляет его, если в качестве второго параметра указать **`true`**. Однако, такой интерфейс явно сложнее для понимания, чем представленный выше.

- **Разделение ответственности.**
  Основной принцип выделения модулей — создание отдельных модулей под каждую задачу, решаемую системой или необходимую в качестве составляющей для решения ее основных задач.
  Пример.
  Класс `java.util.Date` представляет временную метку, состоящую из даты и времени. Это представление должно быть независимо от используемого календаря, формы записи дат и времени в данной стране и часового пояса.
  Для построения конкретных экземпляров этого класса на основе строкового представления даты и времени (например, «22:32:00, June 15, 2005») в том виде, как их используют в Европе, используется класс `java.util.GregorianCalendar`, поскольку интерпретация записи даты и времени зависит от используемой календарной системы. Разные календари представляются различными объектами интерфейса `java.util.Calendar`, которые отвечают за преобразование всех дат в некоторое независимое представление.
  Для создания строкового представления времени и даты используется класс `java.text.SimpleDateFormat`, поскольку нужное представление, помимо календарной системы, может иметь различный порядок перечисления года, месяца и дня месяца и различное количество символов, выделяемое под представление разных элементов даты (например, «22:32:00, June 15, 2005» и «05.06.15, 22:32»).
  Принцип разделения ответственности имеет несколько важных частных случаев.

  - **Разделение политик и алгоритмов.**
    Этот принцип используется для отделения постоянных, неизменяемых алгоритмов обработки данных от изменяющихся их частей и для выделения этих частей, называемых *политиками,* в параметры общего алгоритма.
    Так, политика, определяющая формат строкового представления даты и времени, задается в виде форматной строки при создании объекта класса `java.text.SimpleDateFormat`. Сам же алгоритм построения этого представления основывается на этой форматной строке и на самих времени и дате.
    Другой пример. Стоимость товара для клиента может зависеть от

привилегированности клиента, размера партии, которую он покупает и сезонных скидок. Все перечисленные элементы можно выделить в виде политик, являющихся, вместе с базовой ценой товара, входными данными для алгоритма вычисления итоговой стоимости.

- **Разделение интерфейса и реализации.**
  Этот принцип используется при отделении внешне видимой структуры модуля, описания задач, которые он решает, от способов решения этих задач.
  Пример такого разделения — отделение интерфейса абстрактного списка `java.util.List<E>` от многих возможных реализаций этого интерфейса, например, `java.util.ArrayList<E>`, `java.util.LinkedList<E>`. Первый из этих классов реализует список на основе массива, а второй — на основе ссылочной структуры данных.

- **Слабая связность (coupling) модулей и сильное сродство (cohesion) функций в одном модуле.**
  Оба эти принципа используются для выделения модулей в большой системе и тесно связаны с разделением ответственности между модулями. Первый требует, чтобы зависимостей между модулями было как можно меньше. Модуль, зависящий от большинства остальных модулей в системе, скорее всего, надо перепроектировать — это означает, что он решает слишком много задач.
  И наоборот, «сродство» функций, выполняемых одним модулем, должно быть как можно выше. Хотя на уровне кода причины этого «сродства» могут быть разными — работа с одними и теми же данными, зависимость от работы друг друга, необходимость синхронизации при параллельном выполнении и пр. — цена их разделения должна быть достаточно высокой. Наиболее существенно то, что эти функции решают тесно связанные друг с другом задачи.
  Так, можно добавить в интерфейс очереди метод `void println(String)`, отправляющий строку на стандартный вывод. Но он совсем не связан с остальными и с задачами, решаемыми очередью. Следовательно, трудоемкость анализа и внесения изменений в полученную систему будет значительно выше — ведь изменения в контексте разных задач возникают обычно независимо. Поэтому гораздо лучше поместить такой метод в другой модуль.

- **Переиспользование.**
  Этот принцип требует избегать повторений описаний одних и тех же знаний — в виде структур данных, действий, алгоритмов, одного и того же кода — в разных частях системы. Вместо этого в хорошо спроектированной системе выделяется один источник, одно место фиксации для каждого элемента знаний и организуется его переиспользование во всех местах, где нужно использовать этот элемент знаний. Такая организация позволяет при необходимости (например, при исправлении ошибки или расширении имеющихся возможностей) удобным образом модифицировать код и документы системы в соответствии с новым содержанием элементов знаний, поскольку каждый из них зафиксирован ровно в одном месте.
  Примером может служить организация библиотечных классов `java.util.TreeSet` и `java.util.TreeMap`. Первый класс реализует хранение множества элементов, на которых определен порядок, в виде сбалансированного дерева. Второй класс реализует то же самое для ассоциативного массива или словаря (map), если определен порядок его ключей. Все алгоритмы работы со сбалансированным деревом в обоих случаях одинаковы, поэтому имеет смысл реализовать их только один раз. Если посмотреть на код этих классов в библиотеке JDK от компании Sun, можно увидеть, что ее разработчики так и поступили — класс `TreeSet` реализован как соответствующий ассоциативный массив `TreeMap`, в котором ключи представляют собой множество хранимых значений, а значение в любой паре (ключ, значение) равно **`null`**.

Литература к Лекции 1

[1] Документация по технологиям Java http://java.sun.com/docs/index.html
[2] Т. Кормен, Ч. Лейзерсон, Р. Ривест. Алгоритмы: построение и анализ. М.: МЦНМО, 1999.
[3] И. Соммервилл. Инженерия программного обеспечения. М.: Вильямс, 2002.
[4] Э. Хант, Д. Томас. Программист-прагматик. М.: Лори, 2004.
[5] Е. А. Жоголев. Лекции по технологии программирования: Учебное пособие. М.: Издательский отдел факультета ВМиК МГУ, 2001.
[6] Г. Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Второе издание. М.: Бином, СПб.: Невский диалект, 2000.
[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture. Wiley, 2002.

Лекция 2. Жизненный цикл и процессы разработки ПО

Аннотация

Вводятся понятия жизненного цикла ПО и технологических процессов его разработки. Рассматриваются различные способы организации жизненного цикла ПО, каскадные и итеративные модели жизненного цикла, а также набор стандартов, регулирующих процессы разработки ПО в целом.

Ключевые слова

Жизненный цикл ПО, виды деятельности, роли заинтересованных лиц, процессы жизненного цикла, процесс разработки ПО, стандарты жизненного цикла ПО, модель зрелости возможностей, модели жизненного цикла ПО, каскадная модель жизненного цикла, итеративная модель жизненного цикла, спиральная модель жизненного цикла.

Текст лекции

Понятие жизненного цикла ПО

В первой лекции говорилось о том, что сложную программную систему построить «простыми» методами невозможно. Ее разработка с неизбежностью будет тоже сложной деятельностью.

Привести такое предприятие к успеху возможно на основе общих принципов работы со сложными системами: организовав его в виде набора модулей, используя разные уровни абстракции, переиспользуя отдельные элементы в разных местах так, чтобы изменения в таком элементе автоматически отражались всюду, где он используется.

Разработка ПО — разновидность человеческой деятельности. Выделить ее компоненты можно, определив набор задач, которые нужно решить для достижения конечной цели — построения достаточно качественной системы в рамках заданных сроков и ресурсов. Для решения каждой такой задачи организуется вспомогательная деятельность, к которой можно также применить декомпозицию на отдельные, более мелкие деятельности, и т.д. В итоге должно стать понятно, как решать каждую отдельную подзадачу и всю задачу целиком на основе имеющихся решений для подзадач.

В качестве примеров деятельностей, которые нужно проводить для построения программной системы, можно привести ***проектирование*** — выделение отдельных модулей и определение связей между ними с целью минимизации зависимостей между частями проекта и достижения лучшей его обозримости в целом, ***кодирование*** — разработку кода отдельных модулей, разработку пользовательской документации, которая необходима для достаточно сложной системы.

Однако для корректного с точки зрения инженерии и экономики рассмотрения вопросов создания сложных систем необходимо, чтобы были затронуты и вопросы эксплуатации системы, внесения в нее изменений, а также самые первые действия в ходе ее создания — анализ потребностей пользователей и выработка решений, «изобретение» функций, удовлетворяющих эти потребности. Без этого невозможно, с одной стороны, учесть реальную эффективность системы в виде отношения полученных результатов ко всем сделанным затратам и, с другой стороны, правильно оценивать в ходе разработки степень соответствия системы реальным нуждам пользователей и заказчиков.

Все эти факторы приводят к необходимости рассмотрения всей совокупности деятельностей, связанных с созданием и использованием ПО, начиная с возникновения идеи о новом продукте и заканчивая удалением его последней копии. Весь период существования ПО, связанный с подготовкой к его разработке, разработкой, использованием и модификациями, начиная с того момента, когда принимается решение разработать/приобрести/собрать из имеющихся компонентов новую систему или приходит сама идея о необходимости программы определенного рода, до того момента, когда полностью прекращается всякое ее использование, называют ***жизненным циклом ПО***.

В ходе жизненного цикла ПО проходит через анализ предметной области, сбор требований, проектирование, кодирование, тестирование, сопровождение и др. ***виды деятельности***. Каждый вид представляет собой достаточно однородный набор действий, выполняемых для решения одной задачи или группы тесно связанных задач в рамках разработки и поддержки эксплуатации ПО.

При этом создаются и перерабатываются различного рода ***артефакты*** — создаваемые человеком информационные сущности, документы в достаточно общем смысле, участвующие в качестве входных данных и получающиеся в качестве результатов различных деятельностей. Примерами артефактов являются: модель предметной области, описание требований, техническое задание, архитектура системы, проектная документация на систему в целом и на ее компоненты, прототипы системы и компонентов, собственно, исходный код, пользовательская документация, документация администратора системы, руководство по развертыванию, база пользовательских запросов, план проекта, и пр.

На различных этапах в создание и эксплуатацию ПО вовлекаются люди, выполняющие различные ***роли***. Каждая роль может быть охарактеризована как абстрактная группа заинтересованных лиц, участвующих в деятельности по созданию и эксплуатации системы и решающих одни и те же задачи или имеющих одни и те же интересы по отношению к ней. Примерами ролей являются: бизнес-аналитик, инженер по требованиям, архитектор, проектировщик пользовательского интерфейса, программист-кодировщик, технический писатель, тестировщик, руководитель проекта по разработке, работник отдела продаж, конечный пользователь, администратор системы, инженер по поддержке и т.п.

Похоже, что общую структуру жизненного цикла любого ПО задать невозможно, поскольку она существенно зависит от целей, для которых это ПО разрабатывается или приобретается, и от решаемых им задач. Структура жизненного цикла будет существенно разной у программы для форматирования кода, которая сначала делалась программистом для себя, а впоследствии была признана перспективной в качестве продукта и переработана, и у комплексной системы автоматизации предприятия, которая с самого начала задумывалась как таковая. Тем не менее, часто определяют основные элементы структуры жизненного цикла в виде ***модели жизненного цикла ПО***. Модель жизненного цикла ПО выделяет конкретные наборы видов деятельности (обычно разбиваемых на еще более мелкие активности), артефактов, ролей и их взаимосвязи, а также дает рекомендации по организации процесса в целом. Эти рекомендации включают ответы на вопросы о том, какие артефакты являются входными данными у каких видов деятельности, а какие появляются в качестве результатов, какие роли вовлечены в различные деятельности, как различные деятельности связаны друг с другом, каковы критерии качества полученных результатов, как оценить степень соответствия различных артефактов общим задачам проекта и когда можно переходить от одной деятельности к другой.

Жизненный цикл ПО является составной частью жизненного цикла программно-аппаратной системы, в которую это ПО входит. Поэтому часто различные его аспекты рассматриваются в связи с элементами жизненного цикла системы в целом.

Существует набор стандартов, определяющих различные элементы в структуре жизненных циклов ПО и программно-аппаратных систем. В качестве основных таких элементов выделяют ***технологические процессы*** — структурированные наборы деятельностей, решающих некоторую общую задачу или связанную совокупность задач, такие, как процесс сопровождения ПО, процесс обеспечения качества, процесс разработки документации и пр. Процессы могут определять разные этапы жизненного цикла и увязывать их с различными видами деятельностей, артефактами и ролями заинтересованных лиц.

Стоит отметить, что процессом (или технологическим процессом) называют и набор процессов, увязанных для совместного решения более крупной задачи, например, всю совокупность деятельностей, входящих в жизненный цикл ПО. Таким образом, процессы могут разбиваться на подпроцессы, решающие частные подзадачи той задачи, с которой работает общий процесс.

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. Список и общее содержание этих стандартов представлены ниже.

Группа стандартов ISO

- **ISO/IEC 12207 Standard for Information Technology — Software Life Cycle Processes** [1] (процессы жизненного цикла ПО, есть его российский аналог **ГОСТ Р-1999** [2]). 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

Таблица 1. Процессы жизненного цикла ПО по ISO 12207.

  Процессы строятся из отдельных *видов деятельности (activities)*.
  Стандартом определены 74 вида деятельности, связанной с разработкой и поддержкой ПО. Здесь мы упомянем только некоторые из них.

  - Приобретение ПО включает такие деятельности, как инициация приобретения, подготовка запроса предложений, подготовка контракта, анализ поставщиков, получение ПО и завершение приобретения.

- Разработка ПО включает развертывание процесса разработки, анализ системных требований, проектирование (программно-аппаратной) системы в целом, анализ требований к ПО, проектирование архитектуры ПО, детальное проектирование, кодирование и отладочное тестирование, интеграцию ПО, квалификационное тестирование ПО, системную интеграцию, квалификационное тестирование системы, развертывание (установку или инсталляцию) ПО, поддержку процесса получения ПО.
- Поддержка ПО включает развертывание процесса поддержки, анализ возникающих проблем и необходимых изменений, внесение изменений, экспертизу и передачу измененного ПО, перенос ПО с одной платформы на другую, изъятие ПО из эксплуатации.
- Управление проектом включает запуск проекта и определение его рамок, планирование, выполнение проекта и надзор за его выполнением, экспертизу и оценку проекта, свертывание проекта.

Каждый вид деятельности нацелен на решение одной или нескольких *задач (tasks)*. Всего определено 224 различные задачи. Например:

- Развертывание процесса разработки состоит из определения модели жизненного цикла, документирования и контроля результатов отдельных работ, выбора используемых стандартов, языков и инструментов и пр.
- Перенос ПО между платформами состоит из разработки плана переноса, оповещения пользователей, выполнения анализа произведенных действий и пр.

- **ISO/IEC 15288 Standard for Systems Engineering — System Life Cycle Processes** [3] (процессы жизненного цикла систем).
  Отличается от предыдущего нацеленностью на рассмотрение программно-аппаратных систем в целом.
  В данный момент продолжается работа по приведению этого стандарта в соответствие с предыдущим.
  ISO/IEC 15288 предлагает похожую схему рассмотрения жизненного цикла системы в виде набора процессов. Каждый процесс описывается набором его *результатов (outcomes)*, которые достигаются при помощи различных видов деятельности.

  Всего выделено 26 процессов, объединяемых в 5 групп.

Процессы выработки соглашений	**Процессы уровня организации**	**Процессы уровня проекта**	**Технические процессы**	**Специальные процессы**
Приобретение системы; Поставка системы	Управление окружением; Управление инвестициями; Управление процессами; Управление ресурсами; Управление качеством	Планирование; Оценивание; Мониторинг; Управление рисками; Управление конфигурацией; Управление информацией; Выработка решений	Определение требований; Анализ требований; Проектирование архитектуры; Реализация; Интеграция; Верификация; Валидация; Передача в использование; Эксплуатация; Поддержка; Изъятие из эксплуатации	Адаптация описываемых стандартом процессов под нужды конкретного проекта

Таблица 2. Процессы жизненного цикла систем по ISO 15288.

Помимо процессов, определено 123 различных результата и 208 видов деятельности, нацеленных на их достижение. Например, определение требований имеет следующие результаты.

- Должны быть поставлены технические задачи, которые предстоит решить.
- Должны быть сформулированы системные требования.

Деятельности в рамках этого процесса следующие.

- Определение границ функциональности системы.
- Определение функций, которые необходимо поддерживать.
- Определение критериев оценки качества при использовании системы.
- Анализ и выделение требований по безопасности.
- Анализ требований защищенности.
- Выделение критических для данной системы аспектов качества и требований к ним.
- Анализ целостности системных требований.
- Демонстрация прослеживаемости требований.
- Фиксация и поддержка набора системных требований.

- **ISO/IEC 15504 (SPICE) Standard for Information Technology — Software Process Assessment** [4] (оценка процессов разработки и поддержки ПО).
Определяет правила оценки процессов жизненного цикла ПО и их возможностей, опирается на модель CMMI (см. ниже) и больше ориентирован на оценку процессов и возможностей их улучшения.
В качестве основы для оценки процессов определяет некоторую базовую модель, аналогичную двум описанным выше. В ней выделены категории процессов, процессы и виды деятельности.

  Определяются 5 категорий, включающих 35 процессов и 201 вид деятельности.

Отношения заказчик-поставщик	**Процессы уровня организации**	**Процессы уровня проекта**	**Инженерные процессы**	**Процессы поддержки**
Приобретение ПО; Составление контракта; Определение нужд заказчика; Проведение совместных экспертиз и аудитов; Подготовка к передаче; Поставка и развертывание; Поддержка эксплуатации; Предоставление услуг; Оценка удовлетворенности заказчиков	Развитие бизнеса; Определение процессов; Усовершенствование процессов; Обучение; Обеспечение переиспользования; Обеспечение инструментами; Обеспечение среды для работы	Планирование жизненного цикла; Планирование проекта; Построение команды; Управление требованиями; Управление качеством; Управление рисками; Управление ресурсами и графиком работ; Управление подрядчиками	Выделение системных требований и проектирование системы в целом; Выделение требований к ПО; Проектирование ПО; Реализация, интеграция и тестирование ПО; Интеграция и тестирование системы; Сопровождение системы и ПО	Разработка документации; Управление конфигурацией; Обеспечение качества; Разрешение проблем; Проведение экспертиз

Таблица 3. Процессы жизненного цикла ПО и систем по ISO 15504.

Например, приобретение ПО включает такие виды деятельности, как определение потребности в ПО, определение требований, подготовку стратегии покупки, подготовку запроса предложений, выбор поставщика.

Группа стандартов IEEE

- **IEEE 1074-1997 — IEEE Standard for Developing Software Life Cycle Processes** [5] (стандарт на создание процессов жизненного цикла ПО).
  Нацелен на описание того, как создать специализированный процесс разработки в рамках конкретного проекта. Описывает ограничения, которым должен удовлетворять любой такой процесс, и, в частности, общую структуру процесса разработки. В рамках этой структуры определяет основные виды деятельностей, выполняемых в этих процессах и документы, требующиеся на входе и возникающие на выходе этих деятельностей. Всего рассматриваются 5 подпроцессов, 17 групп деятельностей и 65 видов деятельности. Например, подпроцесс разработки состоит из групп деятельностей по выделению требований, по проектированию и по реализации. Группа деятельностей по проектированию включает архитектурное проектирование, проектирование баз данных, проектирование интерфейсов, детальное проектирование компонентов.
- **IEEE/EIA 12207-1997 — IEEE/EIA Standard: Industry Implementation of International Standard ISO/IEC 12207:1995 Software Life Cycle Processes** [6-8] (промышленное использование стандарта ISO/IEC 12207 на процессы жизненного цикла ПО).
  Аналог ISO/IEC 12207, сменил ранее использовавшиеся стандарты J-Std-016-1995 EIA/IEEE Interim Standard for Information Technology — Software Life Cycle Processes — Software Development Acquirer-Supplier Agreement (промежуточный стандарт на процессы жизненного цикла ПО и соглашения между поставщиком и заказчикам ПО) и стандарт министерства обороны США MIL-STD-498.

Группа стандартов CMM, разработанных SEI

- **Модель зрелости возможностей CMM (Capability Maturity Model)** [9,10] предлагает унифицированный подход к оценке возможностей организации выполнять задачи различного уровня. Для этого определяются 3 уровня элементов: *уровни зрелости организации (maturity levels)*, *ключевые области процесса (key process areas)* и *ключевые практики (key practices)*. Чаще всего под моделью CMM имеют в виду модель уровней зрелости. В настоящий момент CMM считается устаревающей и сменяется моделью CMMI (см. ниже).
  - Уровни зрелости.
    CMM описывает различные степени зрелости процессов в организациях, определяя 5 уровней организаций.
    - Уровень 1, начальный (initial).
      Организации, разрабатывающие ПО, но не имеющие осознанного процесса разработки, не производящие планирования и оценок своих возможностей.
    - Уровень 2, повторяемый (repeatable).
      В таких организациях ведется учет затрат ресурсов и отслеживается ход проектов, установлены правила управления проектами, основанные на имеющемся опыте.
    - Уровень 3, определенный (defined).
      В таких организациях имеется принятый, полностью документированный, соответствующий реальному положению дел и доступный персоналу процесс разработки и сопровождения ПО. Он должен включать как управленческие, так и технические подпроцессы, а также обучение сотрудников работе с ним.
    - Уровень 4, управляемый (manageable).
      В этих организациях, помимо установленного и описанного процесса, используются измеримые показатели качества продуктов и результативности процессов, которые позволяют достаточно точно предсказывать объем ресурсов (времени, денег, персонала), необходимый для разработки продукта с определенным качеством.
    - Уровень 5, совершенствующийся (optimizing).
      В таких организациях, помимо процессов и методов их оценки, имеются методы определения слабых мест, определены процедуры поиска и оценки новых методов и

техник разработки, обучения персонала работе с ними и их включения в общий процесс организации в случае повышения эффективности производства.

- Ключевые области процесса.
  Согласно CMM, уровни зрелости организации можно определять по использованию четко определенных техник и процедур, относящихся к различным ключевым областям процесса. Каждая такая область представляет собой набор связанных видов деятельности, которые нацелены на достижение целей, существенных для общей оценки результативности технологического процесса. Всего выделяется 18 областей. Множество областей, которые должны поддерживаться организацией, расширяется при переходе к более высоким уровням зрелости.
  - К первому уровню не предъявляется никаких требований.
  - Организации второго уровня зрелости должны поддерживать управление требованиями, планирование проектов, надзор за ходом проекта, управление подрядчиками, обеспечение качества ПО, управление конфигурацией.
  - Организации третьего уровня должны, помимо деятельностей второго уровня, поддерживать проведение экспертиз, координацию деятельности отдельных групп, разработку программного продукта, интегрированное управление разработкой и сопровождением, обучение персонала, выработку и поддержку технологического процесса организации, контроль соблюдения технологического процесса организации.
  - К деятельностям организаций четвертого уровня добавляются: управление качеством ПО и управление процессом, основанное на измеримых показателях.
  - Организации пятого уровня зрелости должны дополнительно поддерживать управление изменениями процесса, управление изменениями используемых технологий и предотвращение дефектов.
- Ключевые практики.
  Ключевые области процесса описываются с помощью наборов ключевых практик. Ключевые практики классифицированы на несколько видов: обязательства (commitments to perform), возможности (abilities to perform), деятельности (activities performed), измерения (measurements and analysis) и проверки (verifying implementations).
  Например, управление требованиями связано со следующими практиками.
  - Обязательство.
    Проекты должны следовать определенной политике организации по управлению требованиями.
  - Возможности.
    В каждом проекте должен определяться ответственный за анализ системных требований и привязку их к аппаратному, программному обеспечению и другим компонентам системы.
    Требования должны быть документированы.
    Для управления требованиями должны быть выделены адекватные ресурсы и бюджет.
    Персонал должен проходить обучение в области управления требованиями.
  - Деятельности.
    Прежде чем быть включенными в проект, требования подвергаются анализу на полноту, адекватность, непротиворечивость и пр.
    Выделенные требования используются в качестве основы для планирования и выполнения других работ.
    Изменения в требованиях анализируются и включаются в проект.

- Измерение.
  Производится периодическое определение статуса требований и статуса деятельности по управлению ими.
- Проверки.
  Деятельность по управлению требованиями периодически анализируется старшими менеджерами.
  Деятельность по управлению требованиями периодически и на основании значимых событий анализируется менеджером проекта.
  Группа обеспечения качества проводит анализ и аудит деятельности по управлению требованиями и отчитывается по результатам этого анализа.

Таблица 4 суммирует информацию о количестве практик различных видов, приписанных к разным ключевым областям процесса.

Уровни	Область процесса	Обязательства	Возможности	Деятельности	Измерения	Проверки
2	Управление требованиями	1	4	3	1	3
	Планирование проектов	2	4	15	1	3
	Надзор за ходом проекта	2	5	13	1	3
	Управление подрядчиками	2	3	13	1	3
	Обеспечение качества ПО	1	4	8	1	3
	Управление конфигурацией	1	5	10	1	4
3	Контроль соблюдения технологического процесса	3	4	7	1	1
	Выработка и поддержка технологического процесса	1	2	6	1	1
	Обучение персонала	1	4	6	2	3
	Интегрированное управление	1	3	11	1	3
	Разработка программного продукта	1	4	10	2	3
	Координация деятельности групп	1	5	7	1	3
	Проведение экспертиз	1	3	3	1	1
4	Управление процессом на основе метрик	2	5	7	1	3
	Управление качеством ПО	1	3	5	1	3
5	Предотвращение дефектов	2	4	8	1	3
	Управление изменениями технологий	3	5	8	1	2
	Управление изменениями процесса	2	4	10	1	2

Таблица 4. Количество ключевых практик в разных областях процесса по CMM версии 1.1.

- **Интегрированная модель зрелости возможностей CMMI (Capability Maturity Model Integration)** [11,12].
  Эта модель представляет собой результат интеграции моделей CMM для продуктов и процессов, а также для разработки ПО и разработки программно-аппаратных систем. Основные изменения по сравнению с CMM следующие.
  - Созданы два несколько отличающихся изложения модели — непрерывное и поэтапное. Первое предназначено для облегчения миграции от поддержки американского отраслевого стандарта EIA/AIS 713 и постепенного усовершенствования процессов за

счет внедрения различных практик. Второе предназначено для облегчения миграции от поддержки CMM и поуровневого рассмотрения вводимых практик.

- Элементы модели получили четкие пометки о том, являются ли они обязательными (required), рекомендуемыми (expected) или информативными (informative).
- Используемые практики разделяются на общие (generic) и специфические (specific). Они дополняются набором общих и специфических целей, которые необходимы для достижения определенного уровня зрелости в определенных областях процесса.
- Некоторые уровни зрелости получили другие названия. Второй уровень назван управляемым (managed), а четвертый — управляемым на основе метрик (quantitatively managed).
- Набор выделяемых областей процесса и практик значительно изменился.
  Все области процесса делятся на 4 категории. В приводимом ниже списке области процесса помечены номером уровня, начиная с которого они должны поддерживаться согласно CMMI.
  - Управление процессом.
    Включает выработку и поддержку процесса (3), контроль соблюдения процесса (3), обучение (3), измерение показателей процесса (4), внедрение инноваций (5).
  - Управление проектом.
    Включает планирование проектов (2), контроль хода проекта (2), управление соглашениями с поставщиками (2), интегрированное управление проектами (3), управление рисками (3), построение команд (3), управление поставщиками (3) и измерение показателей результативности и хода проекта (4).
  - Технические.
    Включают выработку требований (3), управление требованиями (2), выработку технических решений (3), интеграцию продуктов (3), верификацию (3) и валидацию (3).
  - Поддерживающие.
    Включают управление конфигурацией (2), обеспечение качества продуктов и процессов (2), проведение измерений и анализ их результатов (2), управление окружением (3), анализ и принятие решений (3), анализ, разрешение и предотвращение проблем (5).

В целом перечисленные стандарты связаны так, как показано на Рис. 2 (сплошные стрелки указывают направления исторического развития, жирная стрелка обозначает идентичность, пунктирные стрелки показывают влияние одних стандартов на другие).

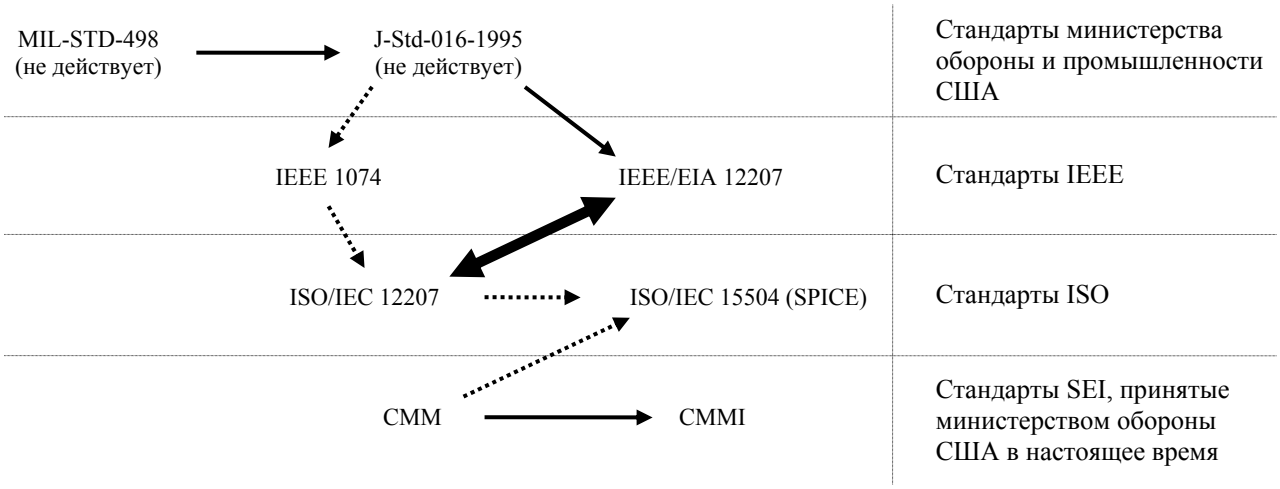


Рисунок 2. Стандарты, описывающие структуру жизненного цикла ПО.

Стандарты являются суммой опыта, который был накоплен экспертами в инженерии ПО на основе огромного количества проектов, проводившихся в рамках коммерческих структур США и

Европы и в рамках военных контрактов. Б***о***льшая часть стандартов создавалась как набор критериев отбора поставщиков программного обеспечения для министерства обороны США, и эту задачу они решают достаточно успешно.

Все рассмотренные стандарты определяют некоторый набор видов деятельности, из которых должен состоять процесс разработки, и задают ту или иную структуру на этих видах деятельности, выделяя их элементы. Вместе с тем, как можно заметить, они не могут быть сведены без существенных изменений в единую модель жизненного цикла ПО. В целом имеющиеся стандарты слабо согласованы между собой. Так что на сегодняшний день (2005 год) нет согласованного комплекта стандартов, покрывающего всю данную область, и в ближайшие несколько лет он вряд ли появится, хотя работа по согласованию различных стандартов ведется.

Кроме того, данные стандарты не предписывают четких и однозначных схем построения жизненного цикла ПО, в частности, связей между отдельными деятельностями. Это сделано намеренно, поскольку ранее действовавшие стандарты типа DoD-Std-2167, были достаточно жестко привязаны к каскадной модели жизненного цикла (см. ниже) и тем самым препятствовали использованию более прогрессивных технологий разработки. Современные стандарты стараются максимально общим образом определить набор видов деятельности, которые должны быть представлены в рамках жизненного цикла (с учетом целей отдельных проектов — т.е. проект, не стремящийся достичь каких-то целей, может не включать деятельностей, связанных с их достижением), и описать их при помощи наборов входных документов и результатов.

Стоит заметить, что стандарты могут достаточно сильно разойтись с реальной разработкой, если в ней используются новейшие методы автоматизации разработки и сопровождения ПО. Стандарты организаций ISO и IEEE построены на основе имеющегося эмпирического опыта разработки, полученного в рамках распространенных некоторое время назад парадигм и инструментальных средств. Это не значит, что они устарели, поскольку их авторы имеют достаточно хорошее представление о новых методах и технологиях разработки и пытались смотреть вперед. Но при использовании новаторских технологий в создании ПО часть требований стандарта может обеспечиваться совершенно иными способами, чем это предусмотрено в нем, а часть артефактов может отсутствовать в рамках данной технологии, исчезнув внутри автоматизированных процессов.

Модели жизненного цикла

Все обсуждаемые стандарты так или иначе пытаются описать, как должен выглядеть *любой* процесс разработки ПО. При этом они вынуждены вводить слишком общие модели жизненного цикла ПО, которые тяжело использовать при организации конкретного проекта.

В рамках специфических моделей жизненного цикла, которые предписывают правила организации разработки ПО в рамках данной отрасли или организации, определяются более конкретные процессы разработки. Отличаются они от стандартов, прежде всего, большей детальностью и четким описанием связей между отдельными видами деятельности, определением потоков данных (документов и артефактов) в ходе жизненного цикла. Таких моделей довольно много, ведь фактически каждый раз, когда некоторая организация определяет собственный процесс разработки, в качестве основы этого процесса разрабатывается некоторая модель жизненного цикла ПО. В рамках данной лекции мы рассмотрим лишь несколько моделей. К сожалению, очень тяжело выбрать критерии, по которым можно было бы дать хоть сколько-нибудь полезную классификацию известных моделей жизненного цикла.

Наиболее широко известной и применяемой долгое время оставалась так называемая ***каскадная*** или ***водопадная (waterfall)*** модель жизненного цикла, которая, как считается, была впервые четко сформулирована в работе [13] и впоследствии запечатлена в стандартах министерства обороны США в семидесятых-восьмидесятых годах XX века. Эта модель предполагает последовательное выполнение различных видов деятельности, начиная с выработки требований и заканчивая сопровождением, с четким определением границ между этапами, на которых набор документов, созданный на предыдущей стадии, передается в качестве входных данных для следующей. Таким образом, каждый вид деятельности выполняется на какой-то одной фазе жизненного цикла. Предлагаемая в статье [13] последовательность шагов разработки

показана на Рис. 3. «Классическая» каскадная модель предполагает только движение вперед по этой схеме: все необходимое для проведения очередной деятельности должно быть подготовлено в ходе предшествующих работ.

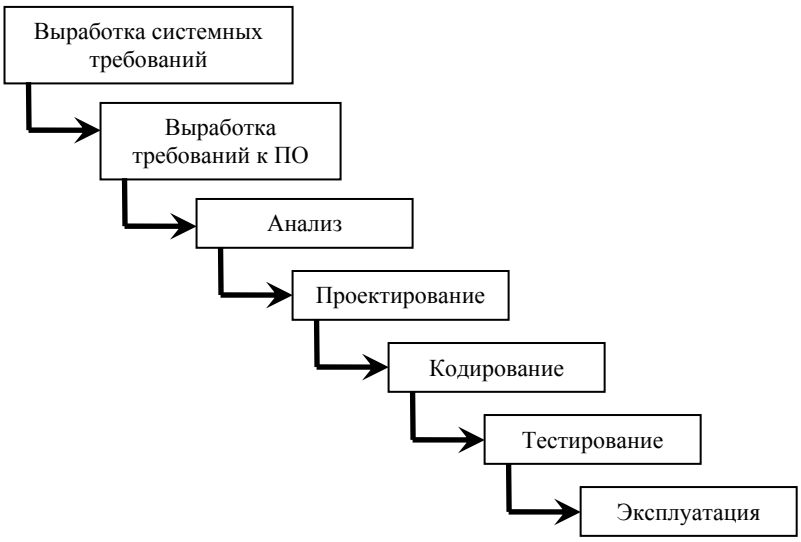


Рисунок 3. Последовательность разработки согласно «классической» каскадной модели.

Однако, если внимательно прочитать статью [13], оказывается, что она не предписывает следование именно этому порядку работ, а, скорее, представляет модель *итеративного процесса* (см. Рис. 4) — в ее последовательном виде эта модель закрепилась, по-видимому, в представлении чиновников из министерств и управленцев компаний, работающих с этими министерствами по контрактам. При реальной работе в соответствии с моделью, допускающей движение только в одну сторону, обычно возникают проблемы при обнаружении недоработок и ошибок, сделанных на ранних этапах. Но еще более тяжело иметь дело с изменениями окружения, в котором разрабатывается ПО (это могут быть изменения требований, смена подрядчиков, изменения политик разрабатывающей или эксплуатирующей организации, изменения отраслевых стандартов, появление конкурирующих продуктов и пр.).

Работать в соответствии с этой моделью можно, только если удается предвидеть заранее возможные перипетии хода проекта и тщательно собирать и интегрировать информацию на первых этапах, с тем, чтобы впоследствии можно было пользоваться их результатами без оглядки на возможные изменения.

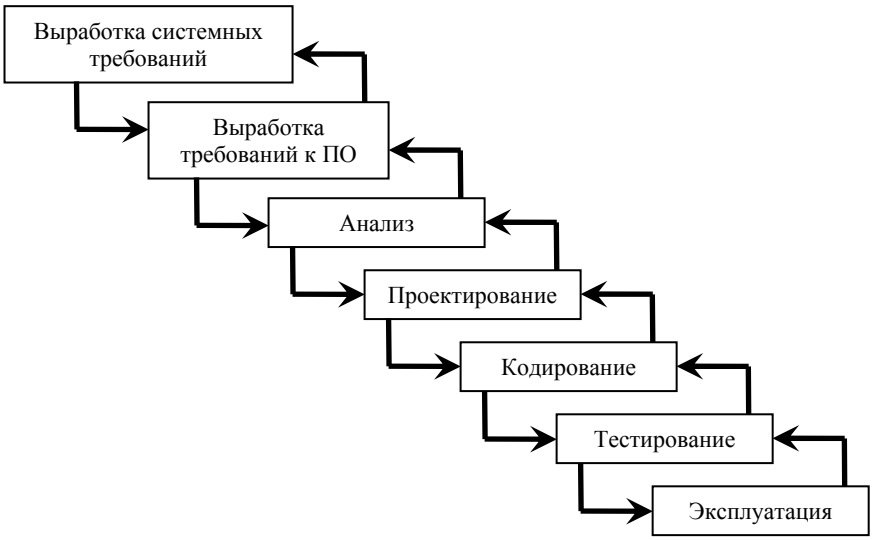


Рисунок 4. Ход разработки, предлагаемый в статье [13].

Среди разработчиков и исследователей, имевших дело с разработкой сложного ПО, практически с самого зарождения индустрии производства программ (см., например, [14])

б*о*льшую популярность имели модели *эволюционных* или *итеративных* процессов, поскольку они обладают большей гибкостью и способностью работать в меняющемся окружении.

Итеративные или ***инкрементальные модели*** (известно несколько таких моделей) предполагают разбиение создаваемой системы на набор кусков, которые разрабатываются с помощью нескольких последовательных проходов всех работ или их части.

На первой итерации разрабатывается кусок системы, не зависящий от других. При этом большая часть или даже полный цикл работ проходится на нем, затем оцениваются результаты и на следующей итерации либо первый кусок переделывается, либо разрабатывается следующий, который может зависеть от первого, либо как-то совмещается доработка первого куска с добавлением новых функций. В результате на каждой итерации можно анализировать промежуточные результаты работ и реакцию на них всех заинтересованных лиц, включая пользователей, и вносить корректирующие изменения на следующих итерациях. Каждая итерация может содержать полный набор видов деятельности от анализа требований, до ввода в эксплуатацию очередной части ПО.

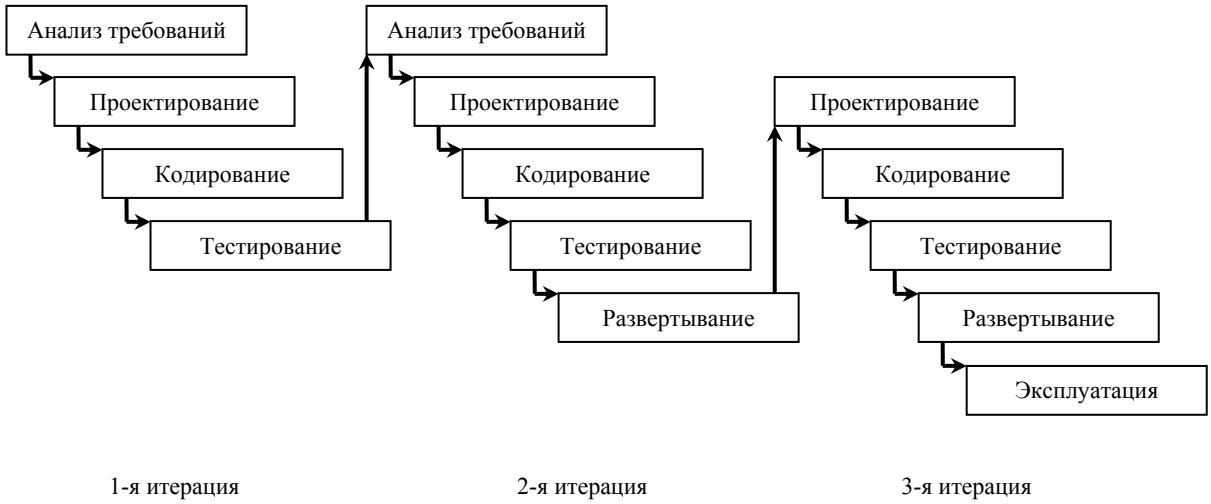


Рисунок 5. Возможный ход работ по итеративной модели.

Каскадная модель с возможностью возвращения на предшествующий шаг при необходимости пересмотреть его результаты, становится итеративной.

Итеративный процесс предполагает, что разные виды деятельности не привязаны намертво к определенным этапам разработки, а выполняются по мере необходимости, иногда повторяются, до тех пор, пока не будет получен нужный результат.

Вместе с гибкостью и возможностью быстро реагировать на изменения, итеративные модели привносят дополнительные сложности в управление проектом и отслеживание его хода. При использовании итеративного подхода значительно сложнее становится адекватно оценить текущее состояние проекта и спланировать долгосрочное развитие событий, а также предсказать сроки и ресурсы, необходимые для обеспечения определенного качества результата.

Развитием идеи итераций является ***спиральная*** модель жизненного цикла ПО, предложенная Боэмом (Boehm) в [15]. Она предлагает каждую итерацию начинать с выделения целей и планирования очередной итерации, определения основных альтернатив и ограничений при ее выполнении, их оценки, а также оценки возникающих рисков и определения способов избавления от них, а заканчивать итерацию оценкой результатов проведенных в ее рамках работ.

Основным ее новым элементом является общая структура действий на каждой итерации — планирование, определение задач, ограничений и вариантов решений, оценка предложенных решений и рисков, выполнение основных работ итерации и оценка их результатов.

Название спиральной эта модель получила из-за изображения хода работ в «полярных координатах», в которых угол соответствует выполняемому этапу в рамках общей структуры итераций, а удаление от начала координат — затраченным ресурсам.

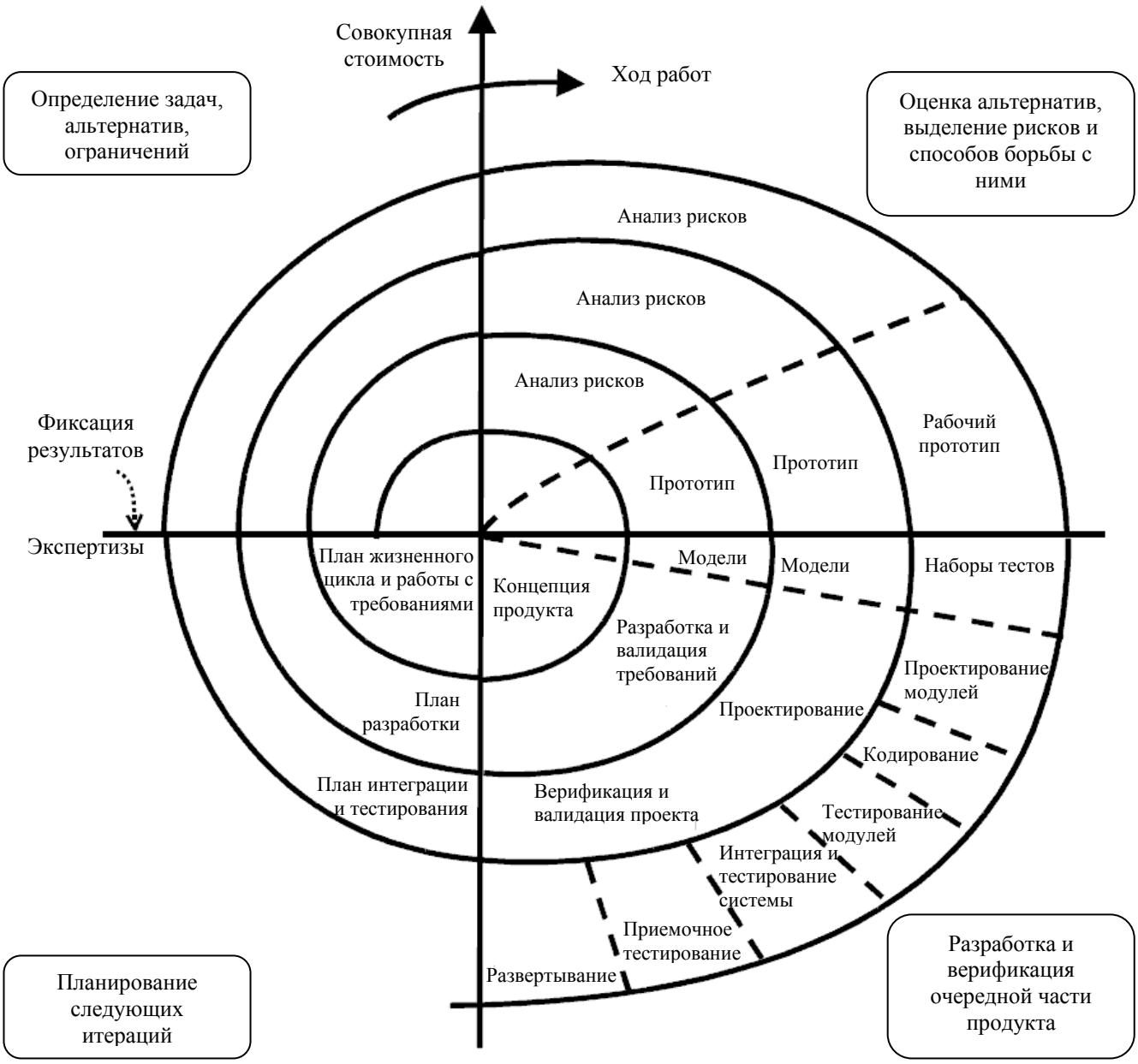


Рисунок 6. Изображение хода работ по спиральной модели согласно [15].

Рис. 6 показывает возможное развитие проекта по спиральной модели. Количество витков, а также расположение и набор видов деятельности в правом нижнем квадранте могут изменяться в зависимости от результатов планирования и анализа рисков, проводимых на предыдущих этапах.

На следующей лекции мы рассмотрим в деталях два современных итеративных процесса разработки — унифицированный процесс разработки Rational и экстремальное программирование.

Литература к Лекции 2

[1] ISO/IEC 12207:1995, Information Technology — Software life cycle processes, 1995. Amendments 2002, 2004.

[2] ГОСТ Р-1999. ИТ. Процессы жизненного цикла программных средств.

[3] ISO/IEC 15288:2002, Systems engineering — System life cycle processes, 2002.

[4] ISO/IEC 15504-1-9, Information technology — Process assessment, Parts 1-9. 15504-1,3,4:2004, 15504-2:2003/Cor 1:2004, TR 15504-5:2004.

[5] IEEE 1074-1997 IEEE Standard for Developing Software Life Cycle Processes, 1997.

[6] IEEE/EIA 12207.0-1996 Industry Implementation of International Standard ISO/IEC 12207:1995, New York, Mar. 1998.

[7] IEEE/EIA 12207.1-1997 Industry Implementation of International Standard ISO/IEC 12207:1995 Software Life Cycle Processes — Life Cycle Data, New York, Apr. 1998.
[8] IEEE/EIA 12207.2-1997 Industry Implementation of Int'l Standard ISO/IEC 12207:1995 Software Life Cycle Processes — Implementation Considerations, New York, Apr. 1998.
[9] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber. Capability Maturity Model for Software, Version 1.1, SEI Technical Report CMU/SEI-93-TR-024, Software Engineering Institute, Pittsburgh, Feb. 1993.
http://www.sei.cmu.edu/pub/documents/93.reports/pdf/tr24.93.pdf
[10] M. C. Paulk, C. V. Weber, S. M. Garcia, M. B. Chrissis, and M. Bush. Key Practices of the Capability Maturity Model, Version 1.1, SEI Technical Report CMU/SEI-93-TR-025, Software Engineering Institute, Pittsburgh, Feb. 1993.
http://www.sei.cmu.edu/pub/documents/93.reports/pdf/tr25.93.pdf
[11] Capability Maturity Model Integration (CMMI), Version 1.1. CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1.1). Continuous Representation. SEI Technical Report CMU/SEI-2002-TR-011, Software Engineering Institute, Pittsburgh, March 2002.
http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr011.pdf
[12] Capability Maturity Model Integration (CMMI), Version 1.1. CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1.1). Staged Representation. SEI Technical Report CMU/SEI-2002-TR-012, Software Engineering Institute, Pittsburgh, March 2002.
http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr012.pdf
[13] W. W. Royce. Managing the Development of Large Software Systems. Proceedings of IEEE WESCON, pp. 1–9, August 1970.
Переиздана: Proceedings of the 9th International Software Engineering Conference, Computer Society Press, pp. 328–338, 1987.
[14] B. Randell, F. W. Zurcher. Iterative Multi-Level Modeling: A Methodology for Computer System Design. Proc. IFIP, IEEE CS Press, 1968.
[15] B. Boehm. A Spiral Model of Software Development and Enhancement. Computer, May 1988, pp. 61-72.
[16] И. Соммервилл. Инженерия программного обеспечения. М.: Вильямс, 2002.
[17] У. Ройс. Управление проектами по созданию программного обеспечения. М.: Лори, 2002.
[18] Э. Дж. Брауде. Технология разработки программного обеспечения. СПб.: Питер, 2004.

Лекция 3. Унифицированный процесс разработки и экстремальное программирование

Аннотация

Рассматриваются в деталях модели разработки ПО, предлагаемые в рамках унифицированного процесса разработки Rational (RUP) и экстремального программирования (XP).

Ключевые слова

«Тяжелые» процессы разработки, «живые» методы разработки, унифицированный процесс Rational, экстремальное программирование, модели ПО.

Текст лекции

«Тяжелые» и «легкие» процессы разработки

В этой лекции мы рассмотрим детально два процесса разработки — *унифицированный процесс Rational (Rational Unified Process, RUP)* и *экстремальное программирование (Extreme Programming, XP)*. Оба они являются примерами итеративных процессов, но построены на основе различных предположений о природе разработки программного обеспечения и, соответственно, достаточно сильно отличаются.

RUP является примером так называемого *«тяжелого»* процесса, детально описанного и предполагающего поддержку собственно разработки исходного кода ПО большим количеством вспомогательных действий. Примерами подобных действий являются разработка планов, технических заданий, многочисленных проектных моделей, проектной документации, и пр. Основная цель такого процесса — отделить успешные практики разработки и сопровождения ПО от конкретных людей, умеющих их применять. Многочисленные вспомогательные действия дают надежду сделать возможным успешное решение задач по конструированию и поддержке сложных систем с помощью имеющихся работников, не обязательно являющихся суперпрофессионалами.

Для достижения этого выполняется иерархическое пошаговое детальное описание предпринимаемых в той или иной ситуации действий, чтобы можно было научить обычного работника действовать аналогичным образом. В ходе проекта создается много промежуточных документов, позволяющих разработчикам последовательно разбивать стоящие перед ними задачи на более простые. Эти же документы служат для проверки правильности решений, принимаемых на каждом шаге, а также отслеживания общего хода работ и уточнения оценок ресурсов, необходимых для получения желаемых результатов.

Экстремальное программирование, наоборот, представляет так называемые *«живые» (agile)* методы разработки, называемые также *«легкими»* процессами. Они делают упор на использовании хороших разработчиков, а не хорошо отлаженных процессов разработки. Живые методы избегают фиксации четких схем действий, чтобы обеспечить б*о*льшую гибкость в каждом конкретном проекте, а также выступают против разработки дополнительных документов, не вносящих непосредственного вклада в получение готовой работающей программы.

Унифицированный процесс Rational

RUP [1,2] является довольно сложной, детально проработанной итеративной моделью жизненного цикла ПО.

Исторически RUP является развитием модели процесса разработки, принятой в компании Ericsson в 70-х–80-х годах XX века. Эта модель была создана Джекобсоном (Ivar Jacobson), впоследствии, в 1987, основавшим собственную компанию Objectory AB именно для развития технологического процесса разработки ПО как отдельного продукта, который можно было бы переносить в другие организации. После вливания Objectory в Rational в 1995 разработки Джекобсона были интегрированы с работами Ройса (Walker Royce, сын автора «классической» каскадной модели), Крухтена (Philippe Kruchten) и Буча (Grady Booch), а также с развивавшимся параллельно *универсальным языком моделирования (Unified Modeling Language, UML)*.

RUP основан на трех ключевых идеях.

- Весь ход работ направляется итоговыми целями проекта, выраженными в виде ***вариантов использования (use cases)*** — сценариев взаимодействия результирующей программной системы с пользователями или другими системами, при выполнении которых пользователи получают значимые для них результаты и услуги. Разработка начинается с выделения вариантов использования и на каждом шаге контролируется степенью приближения к их реализации.
- Основным решением, принимаемым в ходе проекта, является ***архитектура*** результирующей программной системы. Архитектура устанавливает набор компонентов, из которых будет построено ПО, ответственность каждого из компонентов (т.е. решаемые им подзадачи в рамках общих задач системы), четко определяет интерфейсы, через которые они могут взаимодействовать, а также способы взаимодействия компонентов друг с другом.
  Архитектура является одновременно основой для получения качественного ПО и базой для планирования работ и оценок проекта в терминах времени и ресурсов, необходимых для достижения определенных результатов. Она оформляется в виде набора графических моделей на языке UML.
- Основой процесса разработки являются *планируемые и управляемые итерации*, объем которых (реализуемая в рамках итерации функциональность и набор компонентов) определяется на основе архитектуры.

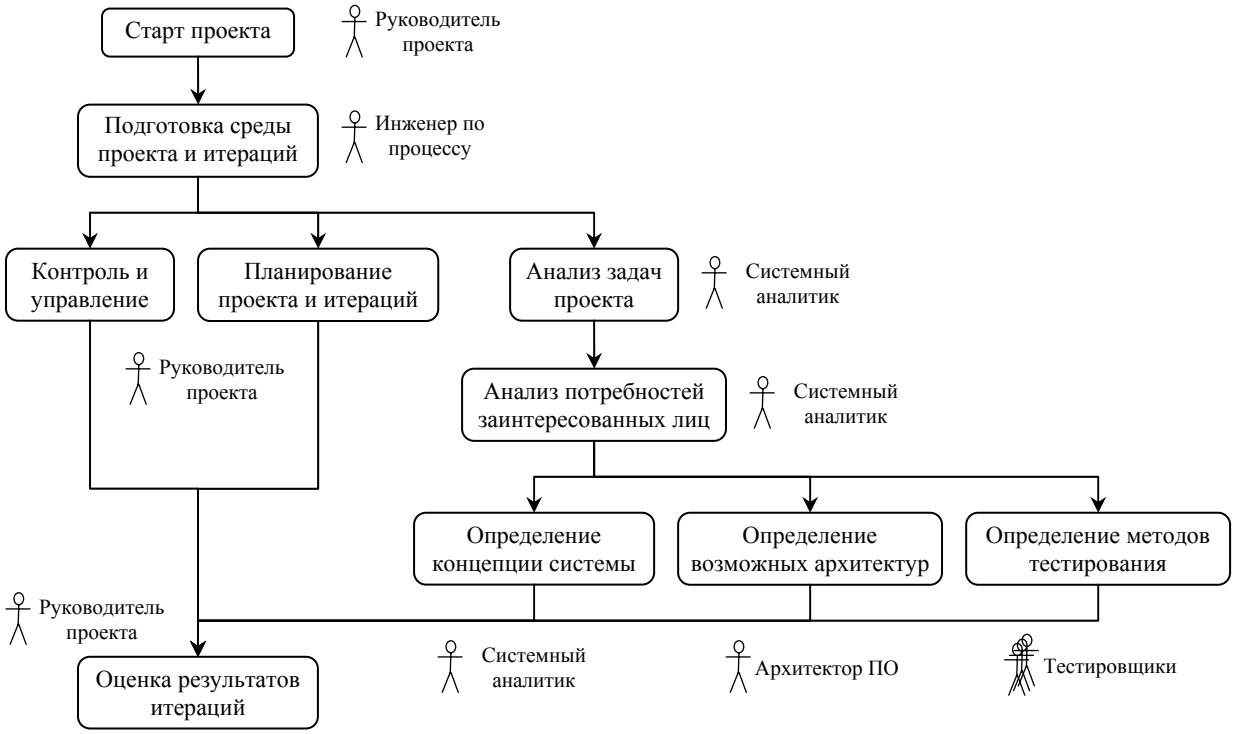


Рисунок 7. Пример хода работ на фазе начала проекта.

RUP выделяет в жизненном цикле на 4 основные фазы, в рамках каждой из которых возможно проведение нескольких итераций. Кроме того, разработка системы может пройти через несколько циклов, включающих все 4 фазы.

1. **Фаза начала проекта (Inception)**
   Основная цель этой фазы — достичь компромисса между всеми заинтересованными лицами относительно задач проекта и выделяемых на него ресурсов.
   На этой стадии определяются основные цели проекта, руководитель и бюджет, основные средства выполнения — технологии, инструменты, ключевые исполнители. Также, возможно, происходит апробация выбранных технологий, чтобы убедиться в возможности достичь целей с их помощью, и составляются предварительные планы проекта.

На эту фазу может уходить около 10% времени и 5% трудоемкости одного цикла.
Пример хода работ показан на Рис. 7.

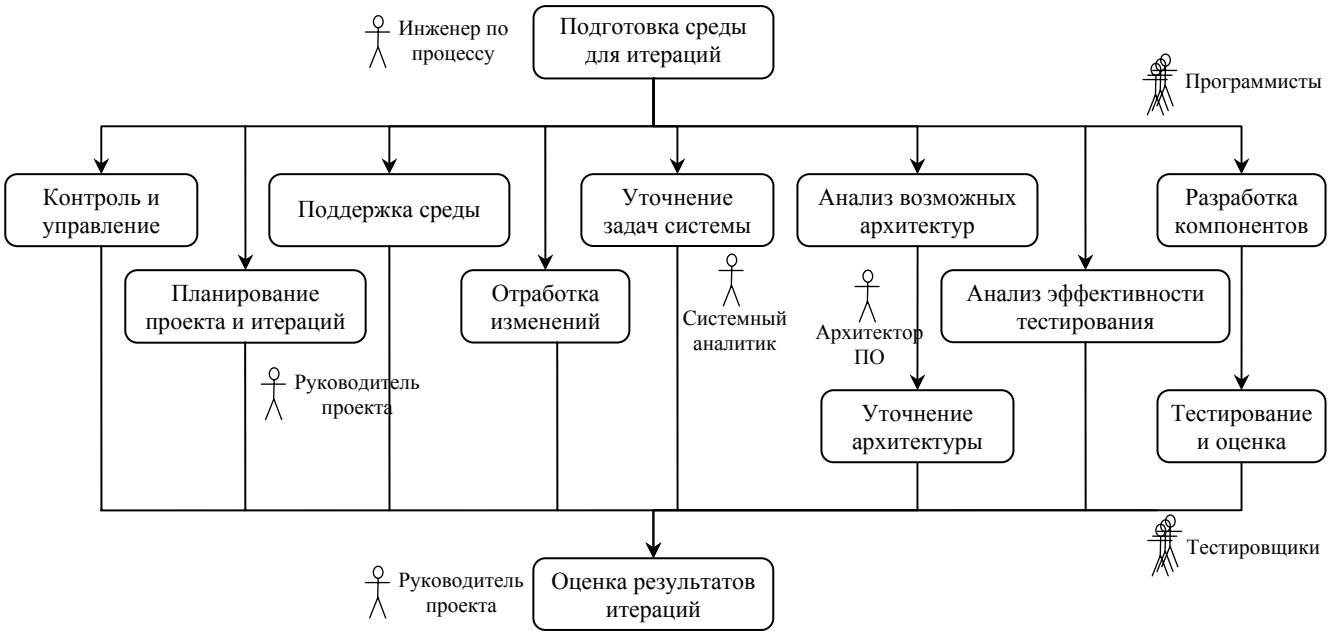


Рисунок 8. Пример хода работ на фазе проектирования.

2. **Фаза проектирования (Elaboration)**
   Основная цель этой фазы — на базе основных, наиболее существенных требований разработать стабильную базовую архитектуру продукта, которая позволяет решать поставленные перед системой задачи и в дальнейшем используется как основа разработки системы.
   На эту фазу может уходить около 30% времени и 20% трудоемкости одного цикла.
   Пример хода работ представлен на Рис. 8.

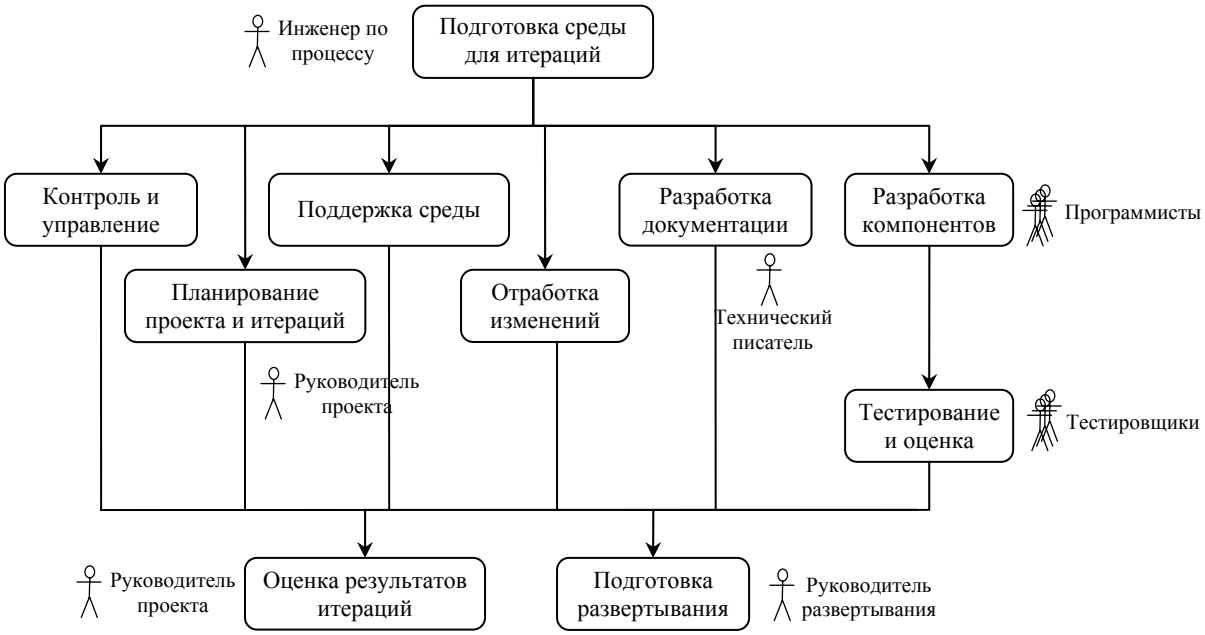


Рисунок 9. Пример хода работ на фазе построения.

3. **Фаза построения (Construction)**
   Основная цель этой фазы — детальное прояснение требований и разработка системы, удовлетворяющей им, на основе спроектированной ранее архитектуры. В результате должна получиться система, реализующая все выделенные варианты использования.

На эту фазу уходит около 50% времени и 65% трудоемкости одного цикла.
Пример хода работ на этой фазе представлен на Рис. 9.

4. **Фаза внедрения (Transition)**
   Цель этой фазы — сделать систему полностью доступной конечным пользователям. На этой стадии происходит развертывание системы в ее рабочей среде, бета-тестирование, подгонка мелких деталей под нужды пользователей.
   На эту фазу может уходить около 10% времени и 10% трудоемкости одного цикла.
   Пример хода работ представлен на Рис. 10.

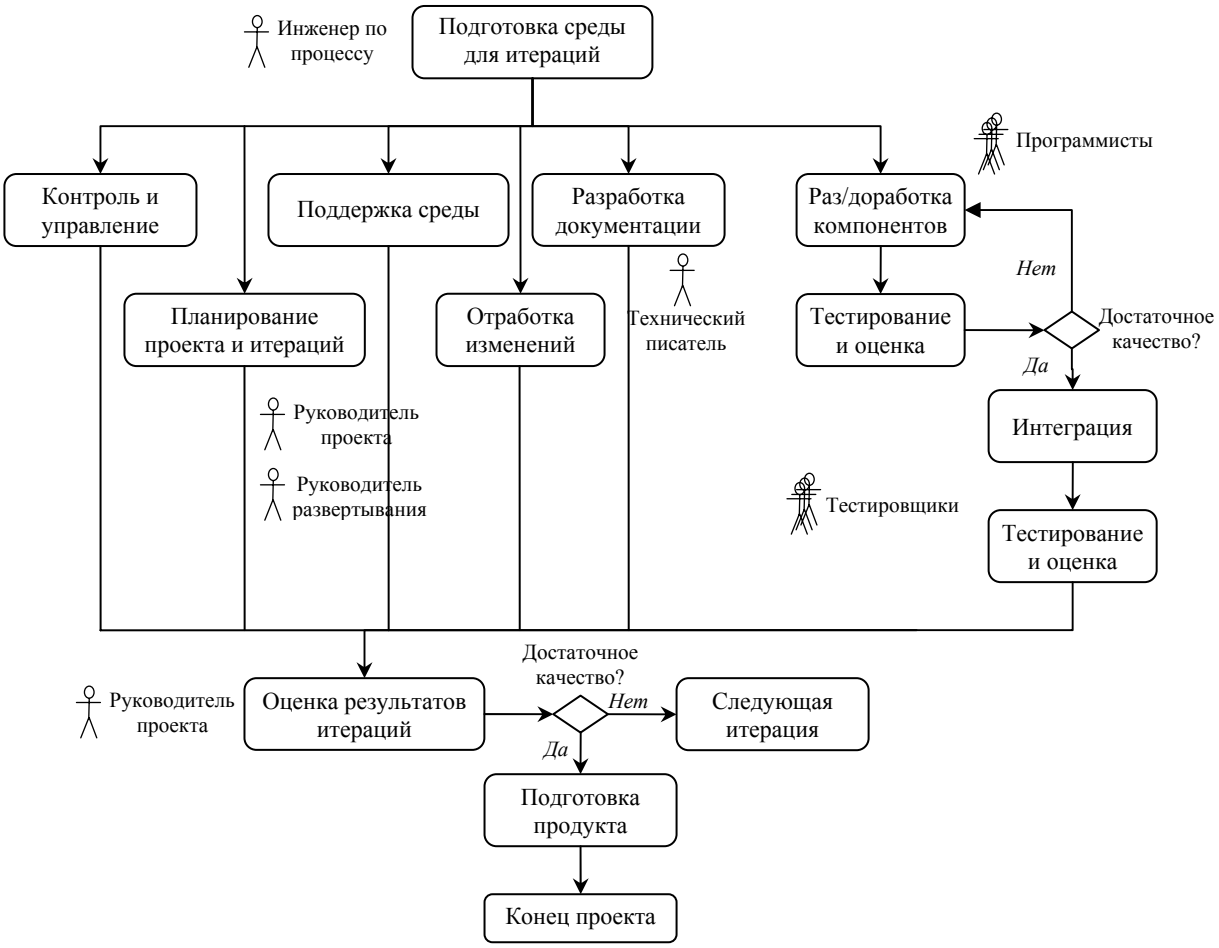


Рисунок 10. Пример хода работ на фазе внедрения.

Артефакты, вырабатываемые в ходе проекта, могут быть представлены в виде баз данных и таблиц с информацией различного типа, разных видов документов, исходного кода и объектных модулей, а также моделей, состоящих из отдельных элементов. Основные артефакты и потоки данных между ними согласно RUP изображены на Рис. 11.

Наиболее важные с точки зрения RUP артефакты проекта — это модели, описывающие различные аспекты будущей системы. Большинство моделей представляют собой наборы диаграмм UML. Основные используемые виды моделей следующие.

- **Модель вариантов использования (Use-Case Model).**
  Эта модель определяет *требования к ПО* — то, что система должна делать — в виде набора вариантов использования. Каждый вариант использования задает сценарий взаимодействия системы с ***действующими лицами (actors)*** или ролями, дающий в итоге значимый для них результат. Действующими лицами могут быть не только люди, но и другие системы, взаимодействующие с рассматриваемой. Вариант использования определяет основной ход событий, развивающийся в нормальной ситуации, а также может включать несколько альтернативных сценариев, которые начинают работать только при специфических условиях.

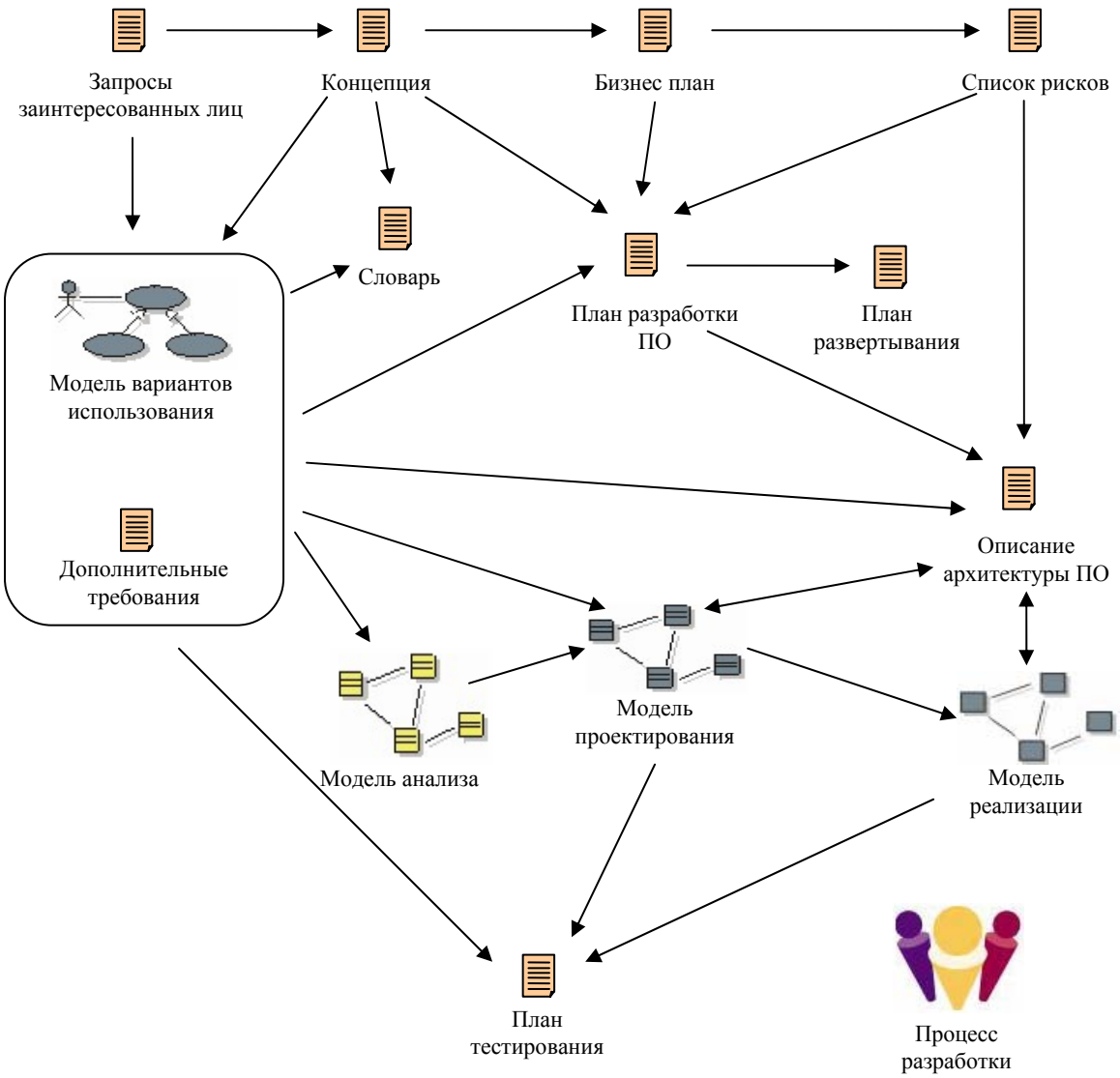


Рисунок 11. Основные артефакты проекта по RUP и потоки данных между ними.

Модель вариантов использования служит основой для проектирования и оценки готовности системы к внедрению.

Примером варианта использования может служить сценарий действий клиента Интернет-магазина по отношению к сайту этого магазина, в результате которых клиент заказывает товар, например, книги. Такой вариант использования можно назвать «Заказ товара». Если нас интересует сайт магазина только как программная система, результатом можно считать то, что запись о сделанном заказе занесена в базу данных, а оператору заказов отправлено электронное письмо, содержащее всю информацию, которая необходима для того, чтобы можно было сформировать заказ. В нее входит контактная информация покупателя, идентификатор заказа и, например, список заказанных книг с их ISBN, их количество для каждого наименования и номера партий для удобства их поиска на складе. При этом выполнение остальной части варианта использования — это дело других составляющих системы под названием «Интернет-магазин». Эта работа может включать звонок или письмо клиенту и подтверждение, что именно он сделал заказ, вопрос об удобных для него форме, времени и адресе доставки и форме оплаты, формирование заказа, передача его для доставки курьеру, доставка и подтверждение получения заказа и оплаты. В нашем примере действующими лицами являются клиент, делающий заказ, и оператор заказов.

Альтернативные сценарии в рамках данного варианта могут выполняться, если, например, заказанного пользователем товара нет на складе или сам пользователь находится на плохом счету в магазине из-за неоплаченных прежних заказов, или, наоборот, он является привилегированным клиентом или представителем крупной организации.

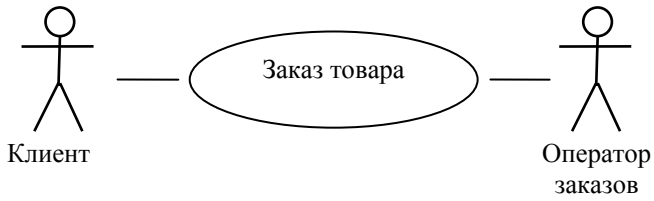


Рисунок 12. Пример варианта использования и действующих лиц.

- **Модель анализа (Analysis Model)**.
  Она включает основные классы, необходимые для реализации выделенных вариантов использования, а также возможные связи между классами. Выделяемые классы разбиваются на три разновидности — *интерфейсные*, *управляющие* и *классы данных*. Эти классы представляют собой набор сущностей, в терминах которых работа системы должна представляться пользователям. Они являются понятиями, с помощью которых достаточно удобно объяснять себе и другим происходящее внутри системы, не слишком вдаваясь в детали.

  Интерфейсные классы (boundary classes) соответствуют устройствам или способам обмена данными между системой и ее окружением, в том числе пользователями. ***Классы данных (entity classes)*** соответствуют наборам данных, описывающих некоторые однотипные сущности внутри системы. Эти сущности являются абстракциями представлений пользователей о данных, с которыми работает система. ***Управляющие классы (control classes)*** соответствуют алгоритмам, реализующим какие-то значимые преобразования данных в системе и управляющим обменом данными с ее окружением в рамках вариантов использования.

  В нашем примере с Интернет-магазином можно было бы выделить следующие классы в модели анализа: интерфейсный класс, предоставляющий информацию о товаре и возможность сделать заказ; интерфейсный класс, представляющий сообщение оператору; управляющий класс, обрабатывающий введенную пользователем информацию и преобразующий ее в данные о заказе и сообщение оператору; класс данных о заказе. Соответствующая модель приведена на Рис. 13.

  Каждый класс может играть несколько ролей в реализации одного или нескольких вариантов использования. Каждая роль определяет его обязанности и свойства, тоже являющиеся частью модели анализа.
  В рамках других подходов модель анализа часто называется ***концептуальной моделью*** системы. Она состоит из набора классов, совместно реализующих все варианты использования и служащих основой для понимания работы системы и объяснения ее правил всем заинтересованным лицам.

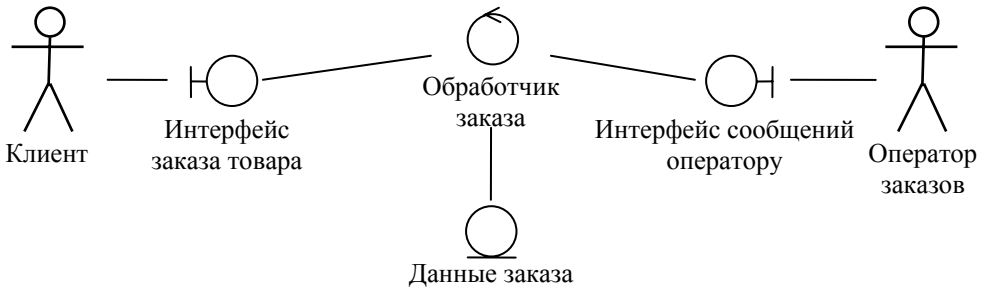


Рисунок 13. Пример модели анализа для одного варианта использования.

- **Модель проектирования (Design Model)**.
  Модель проектирования является детализацией и специализацией модели анализа. Она также состоит из классов, но более четко определенных, с более точным и детальным распределением обязанностей, чем классы модели анализа. Классы модели проектирования должны быть специализированы для конкретной используемой платформы. Каждая такая

платформа может включать операционные системы всех вовлеченных машин; используемые языки программирования; интерфейсы и классы конкретных компонентных сред, таких как J2EE, .NET, COM или CORBA; интерфейсы выбранных для использования систем управления базами данных, СУБД, например, Oracle или MS SQL Server; используемые библиотеки разработки пользовательского интерфейса, такие как swing или swt в Java, MFC или gtk; интерфейсы взаимодействующих систем и пр.

В нашем примере, прежде всего, необходимо детализировать классы, уточнить их функциональность. Скажем, для того, чтобы клиенту было удобнее делать заказ, нужно предоставить ему список имеющихся товаров, какие-то способы навигации и поиска в этом списке, а также детальную информацию о товаре. Это значит, что интерфейс заказа товара реализуется в виде набора классов, представляющих, например, различные страницы сайта магазина. Точно так же данные заказа должны быть детализированы в виде нескольких таблиц в СУБД, включающих, как правило, данные самого заказа (дату, ссылку на данные клиента, строки с количеством отдельных товаров и ссылками на товары), данные товаров, клиента и пр. Кроме того, для реляционной СУБД понадобятся классы-посредники между ее таблицами и объектной структурой остальной программы. Обработчик заказа может быть реализован в виде набора объектов нескольких классов, например, с выделенным отдельно набором часто изменяемых политик (скидки на определенные категории товаров и определенным категориям клиентов, сезонные скидки, рекламные комплекты и пр.) и более постоянным общим алгоритм обработки.
Далее, приняв, например, решение реализовывать систему с помощью технологий J2EE или .NET, мы тем самым определяем дополнительные ограничения на структуру классов, да и на само их количество. О правилах построения ПО на основе этих технологий рассказывается в следующих лекциях.

- **Модель реализации (Implementation Model)**.
  Под *моделью реализации* в рамках RUP и UML понимают набор *компонентов* результирующей системы и связей между ними. Под компонентом здесь имеется в виду ***компонент сборки*** — минимальный по размерам кусок кода системы, который может участвовать или не участвовать в определенной ее конфигурации, единица сборки и конфигурационного управления. Связи между компонентами представляют собой зависимости между ними. Если компонент зависит от другого компонента, он не может быть поставлен отдельно от него.
  Часто компоненты представляют собой отдельные файлы с исходным кодом. Далее мы познакомимся с компонентами J2EE, состоящими из нескольких файлов.
- **Модель развертывания (Deployment Model)**.
  Модель развертывания представляет собой набор ***узлов*** системы, являющихся физически отдельными устройствами, которые способны обрабатывать информацию — серверами, рабочими станциями, принтерами, контроллерами датчиков и пр., со *связями* между ними, образованными различного рода сетевыми соединениями. Каждый узел может быть нагружен некоторым множеством компонентов, определенных в модели реализации.
  Цель построения модели развертывания — определить физическое положение компонентов распределенной системы, обеспечивающее выполнение ею нужных функций в тех местах, где эти функции будут доступны и удобны для пользователей.
  В нашем примере Web-сайта магазина узлами системы являются один или несколько компьютеров, на которых развернуты Web-сервер, пересылающий по запросу пользователя текст нужной странички, набор программных компонентов, отвечающих за генерацию страничек, обработку действий пользователя и взаимодействие с базой данных, и СУБД, в рамках которой работает база данных системы. Кроме того, строго говоря, в систему входят все компьютеры клиентов, на которых работает Web-браузер, делающий возможным просмотр страничек сайта и пересылку кодированных действий пользователя для их обработки.
- **Модель тестирования (Test Model** или **Test Suite)**.
  В рамках этой модели определяются ***тестовые варианты*** или ***тестовые примеры (test***

cases) и *тестовые процедуры (test scripts)*. Первые являются определенными сценариями работы одного или нескольких действующих лиц с системой, разворачивающимися в рамках одного из вариантов использования. Тестовый вариант включает, помимо входных данных на каждом шаге, где они могут быть введены, условия выполнения отдельных шагов и корректные ответы системы для всякого шага, на котором ответ системы можно наблюдать. В отличие от вариантов использования, в тестовых вариантах четко определены входные данные, и, соответственно, тестовый вариант либо вообще не имеет альтернативных сценариев, либо предусматривает альтернативный порядок действий в том случае, если система может вести себя недетерминировано и выдавать разные результаты в ответ на одни и те же действия. Все другие альтернативы обычно заканчиваются вынесением вердикта о некорректной работе системы.
Тестовая процедура представляет собой способ выполнения одного или нескольких тестовых вариантов и их составных элементов (отдельных шагов и проверок). Это может быть инструкция по ручному выполнению входящих в тестовый вариант действий или программный компонент, автоматизирующий запуск тестов.
Для выделенного варианта использования «Заказ товара» можно определить следующие тестовые варианты:

- заказать один из имеющихся на складе товаров и проверить, что сообщение об этом заказе поступило оператору;
- заказать большое количество товаров и проверить, что все работает так же;
- заказать отсутствующий на складе товар и проверить, что в ответ приходит сообщение о его отсутствии;
- сделать заказ от имени пользователя, помещенного в «черный список», и проверить, что в ответ приходит сообщение о неоплаченных прежних заказах.

RUP также определяет *дисциплины,* включающие различные наборы деятельностей, которые в разных комбинациях и с разной интенсивностью выполняются на разных фазах. В документации по процессу каждая дисциплина сопровождается довольно большой диаграммой, поясняющей действия, которые нужно выполнить в ходе работ в рамках данной дисциплины, артефакты, с которыми надо иметь дело, и роли вовлеченных в эти действия лиц.

- **Моделирование предметной области (бизнес-моделирование, Business Modeling)**
  Задачи этой деятельности — понять предметную область или бизнес-контекст, в которых должна будет работать система, и убедиться, что все заинтересованные лица понимают его одинаково, осознать имеющиеся проблемы, оценить их возможные решения и их последствия для бизнеса организации, в которой будет работать система.
  В результате моделирования предметной области должна появиться ее модель в виде набора диаграмм классов (объектов предметной области) и деятельностей (представляющих бизнес-операции и бизнес-процессы). Эта модель служит основой модели анализа.
- **Определение требований (Requirements)**
  Задачи — понять, что должна делать система, и убедиться во взаимопонимании по этому поводу между заинтересованными лицами, определить границы системы и основу для планирования проекта и оценок затрат ресурсов в нем.
  Требования принято фиксировать в виде модели вариантов использования.
- **Анализ и проектирование (Analysis and Design)**
  Задачи — выработать архитектуру системы на основе требований, убедиться, что данная архитектура может быть основой работающей системы в контексте ее будущего использования.
  В результате проектирования должна появиться модель проектирования, включающая диаграммы классов системы, диаграммы ее компонентов, диаграммы взаимодействий между объектами в ходе реализации вариантов использования, диаграммы состояний для отдельных объектов и диаграммы развертывания.

- **Реализация (Implementation)**
  Задачи — определить структуру исходного кода системы, разработать код ее компонентов и протестировать их, интегрировать систему в работающее целое.
- **Тестирование (Test)**
  Задачи — найти и описать дефекты системы (проявления недостатков ее качества), оценить ее качество в целом, оценить выполнены или нет гипотезы, лежащие в основе проектирования, оценить степень соответствия системы требованиям.
- **Развертывание (Deployment)**
  Задачи — установить систему в ее рабочем окружении и оценить ее работоспособность на том месте, где она должна будет работать.
- **Управление конфигурациями и изменениями (Configuration and Change Management)**
  Задачи — определение элементов, подлежащих хранению в репозитории проекта и правил построения из них согласованных конфигураций, поддержание целостности текущего состояния системы, проверка согласованности вносимых изменений.
- **Управление проектом (Project Management)**
  Задачи — планирование, управление персоналом, обеспечение взаимодействия на благо проекта между всеми заинтересованными лицами, управление рисками, отслеживание текущего состояния проекта.
- **Управление средой проекта (Environment)**
  Задачи — подстройка процесса под конкретный проект, выбор и замена технологий и инструментов, используемых в проекте.

Первые пять дисциплин считаются рабочими, остальные — поддерживающими. Распределение объемов работ по дисциплинам в ходе проекта выглядит, согласно руководству по RUP, примерно так, как показано на Рис. 14.

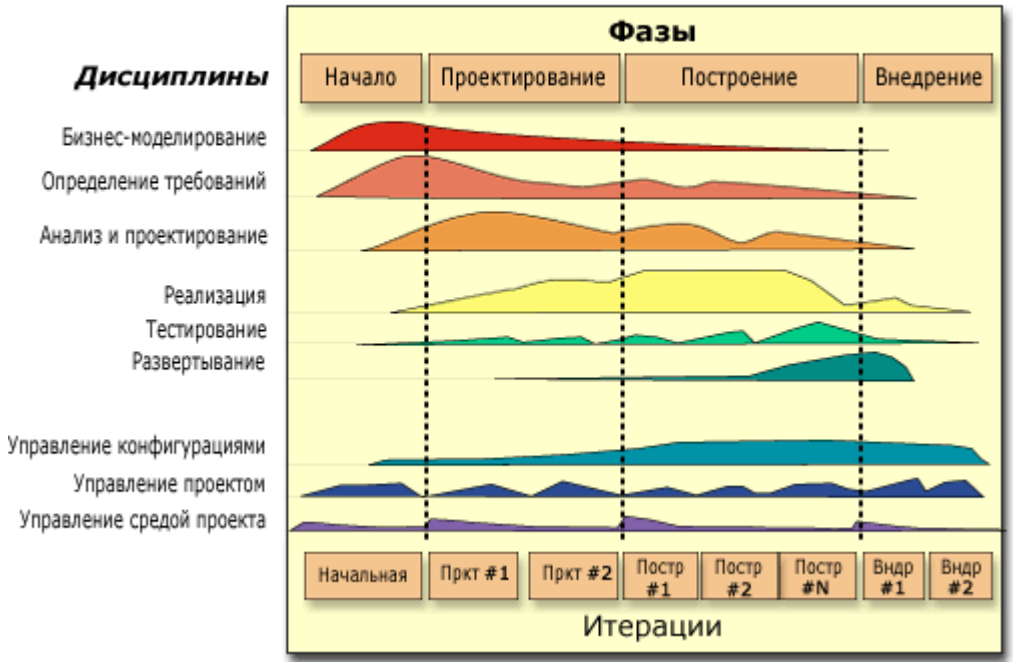


Рисунок 14. Распределение работ между различными дисциплинами в проекте по RUP.

Напоследок перечислим техники, используемые в RUP согласно [3].

- Выработка концепции проекта (project vision) в его начале для четкой постановки задач.
- Управление по плану.
- Снижение рисков и отслеживание их последствий, как можно более раннее начало работ по преодолению рисков.

- Тщательное экономическое обоснование всех действий — делается только то, что нужно заказчику и не приводит к невыгодности проекта.
- Как можно более раннее формирование базовой архитектуры.
- Использование компонентной архитектуры.
- Прототипирование, инкрементная разработка и тестирование.
- Регулярные оценки текущего состояния.
- Управление изменениями, постоянная отработка изменений извне проекта.
- Нацеленность на создание продукта, работоспособного в реальном окружении.
- Нацеленность на качество.
- Адаптация процесса под нужды проекта.

Экстремальное программирование

Экстремальное программирование (Extreme Programming, XP) [4] возникло как эволюционный метод разработки ПО «снизу-вверх». Этот подход является примером так называемого метода *«живой» разработки (Agile Development Method)*. В группу «живых» методов входят, помимо экстремального программирования, методы SCRUM, DSDM (Dynamic Systems Development Method, метод разработки динамичных систем), Feature-Driven Development (разработка, управляемая функциями системы) и др.

Основные принципы «живой» разработки ПО зафиксированы в манифесте «живой» разработки [5], появившемся в 2000 году.

- Люди, участвующие в проекте, и их общение более важны, чем процессы и инструменты.
- Работающая программа более важна, чем исчерпывающая документация.
- Сотрудничество с заказчиком более важно, чем обсуждение деталей контракта.
- Отработка изменений более важна, чем следование планам.

«Живые» методы появились как протест против чрезмерной бюрократизации разработки ПО, обилия побочных, не являющихся необходимыми для получения конечного результата документов, которые приходится оформлять при проведении проекта в соответствии с большинством «тяжелых» процессов, дополнительной работы по поддержке фиксированного процесса организации, как это требуется в рамках, например, CMM. Большая часть таких работ и документов не имеет прямого отношения к разработке ПО и обеспечению его качества, а предназначена для соблюдения формальных пунктов контрактов на разработку, получения и подтверждения сертификатов на соответствие различным стандартам.

«Живые» методы позволяют большую часть усилий разработчиков сосредоточить собственно на задачах разработки и удовлетворения реальных потребностей пользователей. Отсутствие кипы документов и необходимости поддерживать их в связном состоянии позволяет более быстро и качественно реагировать на изменения в требованиях и в окружении, в котором придется работать будущей программе.

Тем не менее, XP имеет свою схему процесса разработки (хотя, вообще говоря, широко используемое понимание «процесса разработки» как достаточно жесткой схемы действий противоречит самой идее «живости» разработки), приведенную на Рис. 15.

По утверждению авторов XP, эта методика представляет собой не столько следование каким-то общим схемам действий, сколько применение комбинации следующих техник. При этом каждая техника важна, и без ее использования разработка считается идущей не по XP, согласно утверждению Кента Бека (Kent Beck) [4], одного из авторов этого подхода, наряду с Уордом Каннингемом (Ward Cunningham), и Роном Джефрисом (Ron Jeffries).

- **Живое планирование (planning game)**
  Его задача — как можно быстрее определить объем работ, которые нужно сделать до следующей версии ПО. Решение принимается, в первую очередь, на основе приоритетов заказчика (т.е. его потребностей, того, что нужно ему от системы для более успешного

ведения своего бизнеса) и, во вторую, на основе технических оценок (т.е. оценок трудоемкости разработки, совместимости с остальными элементами системы и пр.). Планы изменяются, как только они начинают расходиться с действительностью или пожеланиями заказчика.

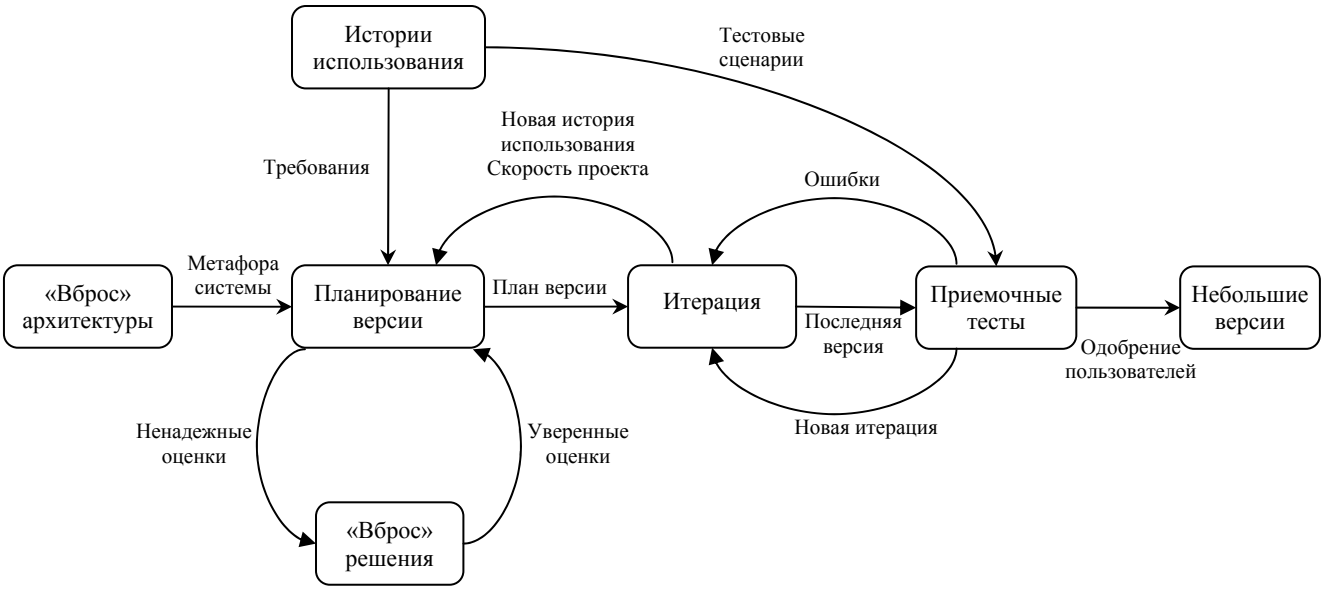


Рисунок 15. Схема потока работ в XP.

- **Частая смена версий (small releases)**
  Самая первая работающая версия должна появиться как можно быстрее и тут же должна начать использоваться. Следующие версии подготавливаются через достаточно короткие промежутки времени (от нескольких часов при небольших изменениях в небольшой программе, до месяца-двух при серьезной переработке крупной системы).
- **Метафора (metaphor) системы**
  Метафора в достаточно простом и понятном команде виде должна описывать основной механизм работы системы. Это понятие напоминает архитектуру, но должно гораздо проще, всего в виде одной-двух фраз описывать основную суть принятых технических решений.
- **Простые проектные решения (simple design)**
  В каждый момент времени система должна быть сконструирована настолько просто, насколько это возможно. Не надо добавлять функции заранее — только после явной просьбы об этом. Вся лишняя сложность удаляется, как только обнаруживается.
- **Разработка на основе тестирования (test-driven development)**
  Разработчики сначала пишут тесты, потом пытаются реализовать свои модули так, чтобы тесты срабатывали. Заказчики заранее пишут тесты, демонстрирующие основные возможности системы, чтобы можно было увидеть, что система действительно заработала.
- **Постоянная переработка (refactoring)**
  Программисты постоянно перерабатывают систему для устранения излишней сложности, увеличения понятности кода, повышения его гибкости, но без изменений в его поведении, что проверяется прогоном после каждой переделки тестов. При этом предпочтение отдается более элегантным и гибким решениям, по сравнению с просто дающими нужный результат. Неудачно переработанные компоненты должны выявляться при выполнении тестов и откатываться к последнему целостному состоянию (вместе с зависимыми от них компонентами).
- **Программирование парами (pair programming)**
  Кодирование выполняется двумя программистами на одном компьютере. Объединение в пары произвольно и меняется от задачи к задаче. Тот, в чьих руках клавиатура, пытается наилучшим способом решить текущую задачу. Второй программист анализирует работу

первого и дает советы, обдумывает последствия тех или иных решений, новые тесты, менее прямые, но более гибкие решения.

- **Коллективное владение кодом (collective ownership)**
  В любой момент любой член команды может изменить любую часть кода. Никто не должен выделять свою собственную область ответственности, вся команда в целом отвечает за весь код.
- **Постоянная интеграция (continuous integration)**
  Система собирается и проходит интеграционное тестирование как можно чаще, по несколько раз в день, каждый раз, когда пара программистов оканчивает реализацию очередной функции.
- **40-часовая рабочая неделя**
  Сверхурочная работа рассматривается как признак больших проблем в проекте. Не допускается сверхурочная работа 2 недели подряд — это истощает программистов и делает их работу значительно менее продуктивной.
- **Включение заказчика в команду (on-site customer)**
  В составе команды разработчиков постоянно находится представитель заказчика, который доступен в течение всего рабочего дня и способен отвечать на все вопросы о системе. Его обязанностью являются достаточно оперативные ответы на вопросы любого типа, касающиеся функций системы, ее интерфейса, требуемой производительности, правильной работы системы в сложных ситуациях, необходимости поддерживать связь с другими приложениями и пр.
- **Использование кода как средства коммуникации**
  Код рассматривается как важнейшее средство общения внутри команды. Ясность кода — один из основных приоритетов. Следование стандартам кодирования, обеспечивающим такую ясность, обязательно. Такие стандарты, помимо ясности кода, должны обеспечивать минимальность выражений (запрет на дублирование кода и информации) и должны быть приняты всеми членами команды.
- **Открытое рабочее пространство (open workspace)**
  Команда размещается в одном, достаточно просторном помещении, для упрощения коммуникации и возможности проведения коллективных обсуждений при планировании и принятии важных технических решений.
- **Изменение правил по необходимости (just rules)**
  Каждый член команды должен принять перечисленные правила, но при возникновении необходимости команда может поменять их, если все ее члены пришли к согласию по поводу этого изменения.

Как видно из применяемых техник, XP рассчитано на использование в рамках небольших команд (не более 10 программистов), что подчеркивается и авторами этой методики. Больший размер команды разрушает необходимую для успеха простоту коммуникации и делает невозможным применение многих перечисленных приемов.

Достоинствами XP, если его удается применить, является большая гибкость, возможность быстро и аккуратно вносить изменения в ПО в ответ на изменения требований и отдельные пожелания заказчиков, высокое качество получающегося в результате кода и отсутствие необходимости убеждать заказчиков в том, что результат соответствует их ожиданиям.

Недостатками этого подхода являются невыполнимость в таком стиле достаточно больших и сложных проектов, невозможность планировать сроки и трудоемкость проекта на достаточно долгую перспективу и четко предсказать результаты длительного проекта в терминах соотношения качества результата и затрат времени и ресурсов. Также можно отметить неприспособленность XP для тех случаев, в которых возможные решения не находятся сразу на основе ранее полученного опыта, а требуют проведения предварительных исследований

XP как совокупность описанных техник впервые было использовано в ходе работы на проектом C3 (Chrysler Comprehensive Compensation System, разработка системы учета выплат

работникам компании Daimler Chrysler). Из 20-ти участников этого проекта 5 (в том числе упомянутые выше 3 основных автора XP) опубликовали еще во время самого проекта и в дальнейшем 3 книги и огромное количество статей, посвященных XP. Этот проект неоднократно упоминается в различных источниках как пример использования этой методики [6,7,8]. Приведенные ниже данные собраны на основе упомянутых статей [9], за вычетом не подтверждающихся сведений, и иллюстрируют проблемы некоторых техник XP при их применении в достаточно сложных проектах.

Проект стартовал в январе 1995 года. С марта 1996 года, после включения в него Кента Бека, он проходил с использованием XP. К этому времени он уже вышел за рамки бюджета и планов поэтапной реализации функций. Команда разработчиков была сокращена, и в течение примерно полугода после этого проект развивался довольно успешно. В августе 1998 года появился прототип, который мог обслуживать около 10000 служащих. Первоначально предполагалось, что проект завершится в середине 1999 года и результирующее ПО будет использоваться для управления выплатами 87000 служащим компании. Он был остановлен в феврале 2000 года после 4-х лет работы по XP в связи с полным несоблюдением временных рамок и бюджета. Созданное ПО ни разу не использовалось для работы с данными о более чем 10000 служащих, хотя было показано, что оно справится с данными 30000 работников компании. Человек, игравший роль включенного в команду заказчика в проекте, уволился через несколько месяцев такой работы, не выдержав нагрузки, и так и не получил адекватной замены до конца проекта.

Литература к Лекции 3

[1] У. Ройс. Управление проектами по созданию программного обеспечения. М.: Лори, 2002.

[2] А. Якобсон, Г. Буч, Дж. Рамбо. Унифицированный процесс разработки программного обеспечения. СПб.: Питер, 2002.

[3] Kroll, The Spirit of the RUP. *www-106.ibm.com/developerworks/rational/library/content/RationalEdge/dec01/ TheSpiritoftheRUPDec01.pdf*

[4] К. Бек. Экстремальное программирование. СПб.: Питер, 2002.

[5] http://www.agilemanifesto.org/

[6] K. Beck, et. al. Chrysler goes to “Extremes”. Distributed Computing, 10/1998.

[7] A. Cockburn. Selecting a Project’s Methodology. IEEE Software, 04/2000.

[8] L. Williams, R. R. Kessler, W. Cunningham, R. Jeffries. Strengthening the Case for Pair Programming. IEEE Software 4/2000.

[9] G. Keefer. Extreme Programming Considered Harmful for Reliable Software Development. AVOCA Technical Report, 2002.
Доступен как http://www.avoca-vsm.com/Dateien-Download/ExtremeProgramming.pdf.

Лекция 4. Анализ предметной области и требования к ПО

Аннотация

Рассматриваются вопросы, связанные с анализом предметной области и выделением требований к разрабатываемой программной системе, а также основные графические модели, используемые в этих деятельностях — диаграммы потоков данных и вариантов использования.

Ключевые слова

Анализ предметной области, схема Захмана, модели предметной области, диаграммы потоков данных, диаграммы сущностей и связей, функции ПО, требования к ПО, варианты использования, действующие лица, диаграммы вариантов использования.

Текст лекции

Анализ предметной области

Для того, чтобы разработать программную систему, приносящую реальные выгоды определенным пользователям, необходимо сначала выяснить, какие же задачи она должна решать для этих людей и какими свойствами обладать.

Требования к ПО определяют, какие свойства и характеристики оно должно иметь для удовлетворения потребностей пользователей и других заинтересованных лиц. Однако сформулировать требования к сложной системе не так легко. В большинстве случаев будущие пользователи могут перечислить набор свойств, который они хотели бы видеть, но никто не даст гарантий, что это — исчерпывающий список. Кроме того, часто сама формулировка этих свойств будет непонятна большинству программистов: могут прозвучать фразы типа «должно использоваться и частотное, и временное уплотнение каналов», «передача клиента должна быть мягкой», «для обычных швов отмечайте бригаду, а для доверительных — конкретных сварщиков», и это еще не самые тяжелые для понимания примеры.

Чтобы ПО было действительно полезным, важно, чтобы оно удовлетворяло реальные потребности людей и организаций, которые часто отличаются от непосредственно выражаемых пользователями желаний. Для выявления этих потребностей, а также для выяснения смысла высказанных требований приходится проводить достаточно большую дополнительную работу, которая называется ***анализом предметной области*** или ***бизнес-моделированием,*** если речь идет о потребностях коммерческой организации. В результате этой деятельности разработчики должны научиться понимать язык, на котором говорят пользователи и заказчики, выявить цели их деятельности, определить набор задач, решаемых ими. В дополнение стоит выяснить, какие вообще задачи нужно уметь решать для достижения этих целей, выяснить свойства результатов, которые хотелось бы получить, а также определить набор сущностей, с которыми приходится иметь дело при решении этих задач. Кроме того, анализ предметной области позволяет выявить места возможных улучшений и оценить последствия принимаемых решений о реализации тех или иных функций.

После этого можно определять область ответственности будущей программной системы — какие именно из выявленных задач будут ею решаться, при решении каких задач она может оказать существенную помощь и чем именно. Определив эти задачи в рамках общей системы задач и деятельностей пользователей, можно уже более точно сформулировать требования к ПО.

Анализом предметной области занимаются ***системные аналитики*** или ***бизнес-аналитики***, которые передают полученные ими знания другим членам проектной команды, сформулировав их на более понятном разработчикам языке. Для передачи этих знаний обычно служит некоторый набор моделей, в виде графических схем и текстовых документов.

Анализ деятельности крупной организации, такой, как банк с сетью региональных отделений, нефтеперерабатывающий завод или компания, производящая автомобили, дает огромные объемы информации. Из этой информации надо уметь отбирать существенную, а также надо уметь находить в ней пробелы — области деятельности, информации по которым недостаточно для

четкого представления о решаемых задачах. Значит, всю получаемую информацию надо каким-то образом систематизировать. Для систематизации сбора информации о больших организациях и дальнейшей разработки систем, поддерживающих их деятельность, применяется *схема Захмана* (автор — John Zachman, [1,2]) или *архитектурная схема предприятия (enterprise architecture framework)*.

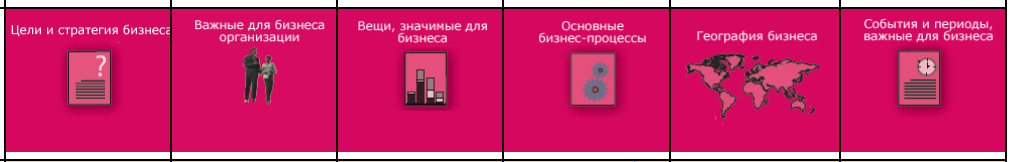
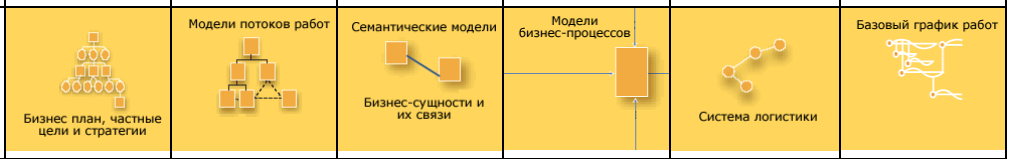
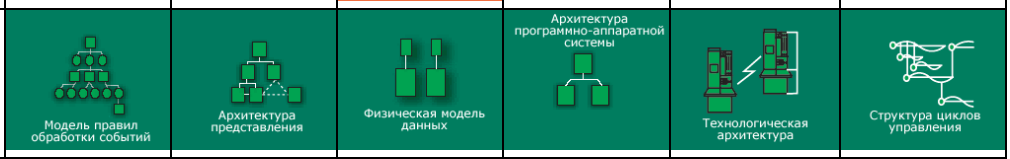
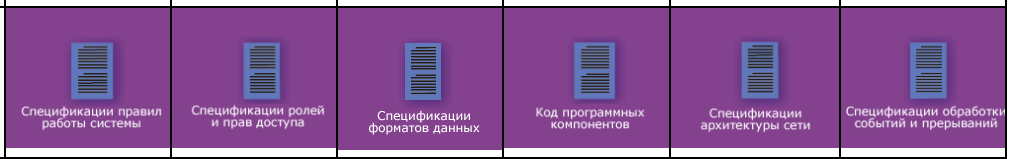
	Мотивация	Люди	Данные	Функции	Место	Время
Контекст						
Модель бизнеса						
Системная модель						
Технологическая модель						
Детальное представление						
Работающая организация	Стратегия и тактика	Структура организации	Данные	Выполняемые функции	Географическое расположение и сети	Планы

Таблица 5. Схема Захмана. Приведены примеры моделей для отдельных клеток.

В основе схемы Захмана лежит следующая идея: деятельность даже очень большой организации можно описать, используя ответы на простые вопросы — зачем, кто, что, как, где и когда, — и разные уровни рассмотрения. Обозначенные 6 вопросов определяют 6 аспектов рассмотрения.

- Цели организации и базовые правила, по которым она работает.
- Персонал, подразделения и другие элементы организационной структуры, связи между ними.
- Сущности и данные, с которыми имеет дело организация.
- Выполняемые организацией и различными ее подразделениями функции и операции над данными.
- Географическое распределение элементов организации и связи между географически разделенными ее частями.
- Временные характеристики и ограничения на деятельность организации, значимые для ее деятельности события.

Также выделены несколько уровней рассмотрения, из которых при бизнес-моделировании особенно важны три верхних.

- Самый крупный — уровень организации в целом, рассматриваемой в ее развитии совместно с окружением, уровень общего планирования ее деятельности. Этот уровень содержит долговременные цели и задачи организации как цельной системы, основные связи организации с внешним миром и основные виды ее деятельности.

- Уровень бизнеса, на котором организация рассматривается во всех аспектах как отдельная сущность, имеющая определенную структуру, которая соответствует ее основным задачам.
- Системный уровень, на котором определяются концептуальные модели всех аспектов организации, без привязки к конкретным их воплощениям и реализациям, например, логическая модель данных в виде набора сущностей и связей между ними, логическая архитектура системы автоматизации в виде набора узлов, с привязанными к ним функциями и пр.

Наиболее удобной формой представления информации при анализе предметной области являются графические диаграммы различного рода. Они позволяют достаточно быстро зафиксировать полученные знания, быстро восстанавливать их в памяти и успешно объясняться с заказчиками и другими заинтересованными лицами. Набросать рисунок из прямоугольников и связывающих их стрелок обычно можно гораздо быстрее, чем записать соответствующий объем информации, и на рисунке за один взгляд видно гораздо больше, чем в тексте. Изредка встречаются люди, лучше ориентирующиеся в текстах и более адекватно их понимающие, но чаще рисунки все же более удобны для иллюстрации мыслей и объяснения сложных вещей.

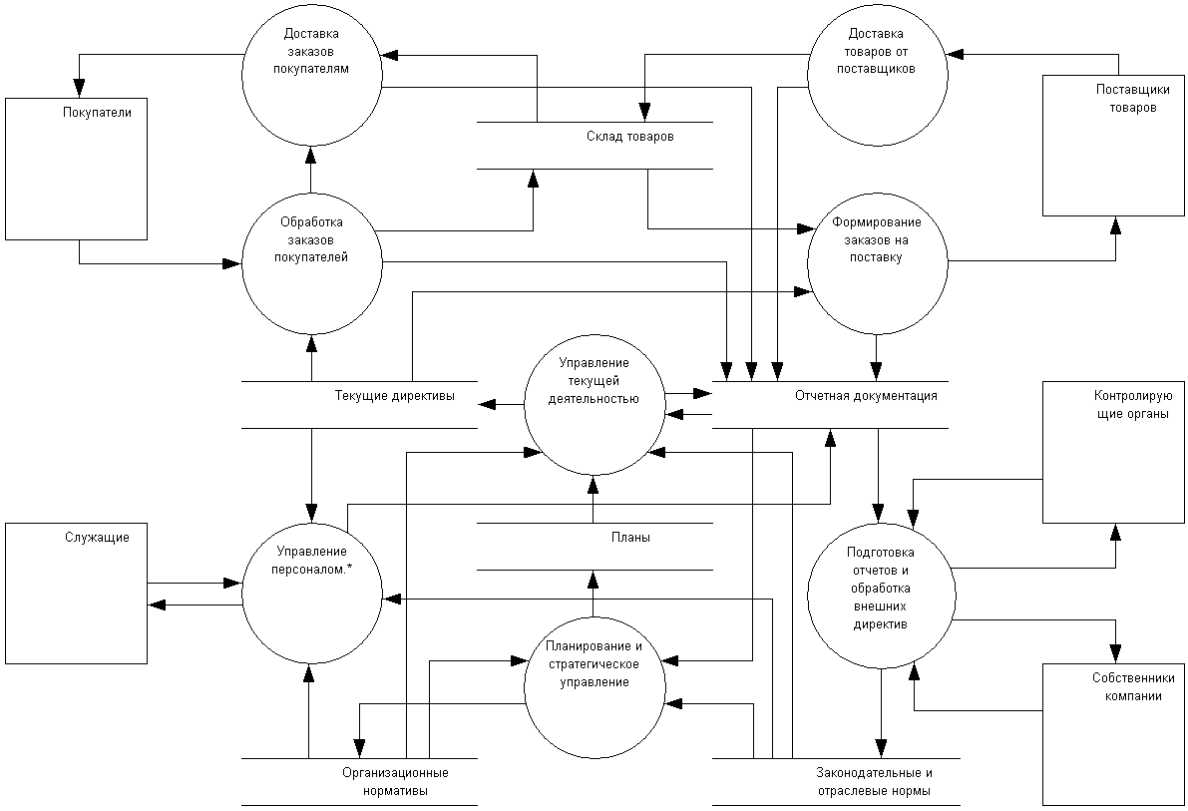


Рисунок 16. Схема деятельности компании в нотации Йордана-ДеМарко.

Часто для описания поведения сложных систем и деятельности крупных организаций используются ***диаграммы потоков данных (data flow diagrams)***. Эти диаграммы содержат 4 вида графических элементов: *процессы*, представляющие собой любые трансформации данных в рамках описываемой системы, *хранилища данных*, *внешние* по отношению к системе *сущности* и *потоки данных* между элементами трех предыдущих видов.

Используются несколько систем обозначений для перечисленных элементов, наиболее известны нотация Йордана-ДеМарко (Yourdon-DeMarco, [3,4]) и нотация Гэйна-Сарсона (Gane-Sarson, [5]), обе предложенные в 1979 году. Рис. 16 показывает диаграмму потоков данных, которая описывает деятельность компании, управляющей небольшим магазином. Эта диаграмма изображена в нотации Йордана-ДеМарко: процессы изображаются кружками, внешние сущности — прямоугольниками, а хранилища данных — двумя горизонтальными параллельными линиями. На Рис. 17 изображена та же диаграмма в нотации Гейна-Сарсона: на ней процессы —

прямоугольники со скругленными углами, внешние сущности — прямоугольники с тенью, а хранилища данных — вытянутые горизонтально прямоугольники без правого ребра.

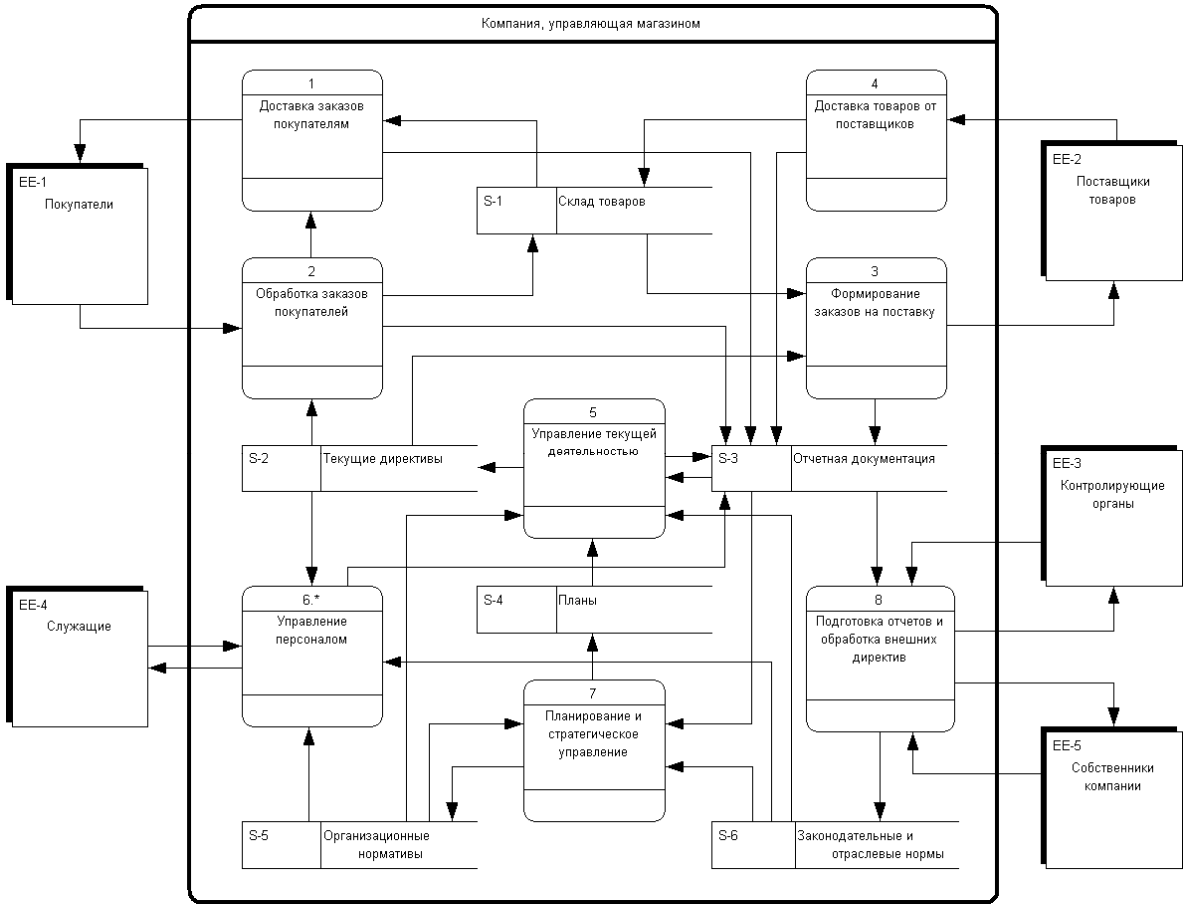


Рисунок 17. Схема деятельности компании в нотации Гэйна-Сарсона.

Процессы на диаграммах потоков данных могут уточняться: если некоторый процесс устроен достаточно сложно, для него можно нарисовать отдельную диаграмму, описывающую потоки данных внутри этого процесса. На ней показываются те элементы, с которыми этот процесс связан потоками данных, и составляющие его более мелкие процессы и хранилища. Таким образом, возникает иерархическая структура процессов. Обычно на самом верхнем уровне находится один процесс, представляющий собой систему в целом, и набор внешних сущностей, с которыми она взаимодействует.

На Рис. 18 показана возможная детализация процесса «Управление персоналом».

Диаграммы потоков данных появились как один из первых инструментов представления деятельности сложных систем при использовании *структурного анализа*. Для представления структуры данных в этом подходе используются ***диаграммы сущностей и связей (entity-relationship diagrams, ER diagrams)*** [6], изображающие набор *сущностей* предметной области и *связей* между ними. И сущности, и связи на таких диаграммах могут иметь атрибуты. Пример такой диаграммы представлен на Рис. 19.

Хотя методы структурного анализа могут значительно помочь при анализе систем и организаций, дальнейшая разработка системы, поддерживающей их деятельность, с использованием объектно-ориентированного подхода часто требует дополнительной работы по переводу полученной информации в объектно-ориентированные модели.

Методы объектно-ориентированного анализа предназначены для обеспечения более удобной передачи информации между моделями анализируемых систем и моделями разрабатываемого ПО. В качестве графических моделей в этих методах вместо диаграмм потоков данных используются рассматривавшиеся при обсуждении RUP диаграммы вариантов использования, а вместо диаграмм сущностей и связей — диаграммы классов.

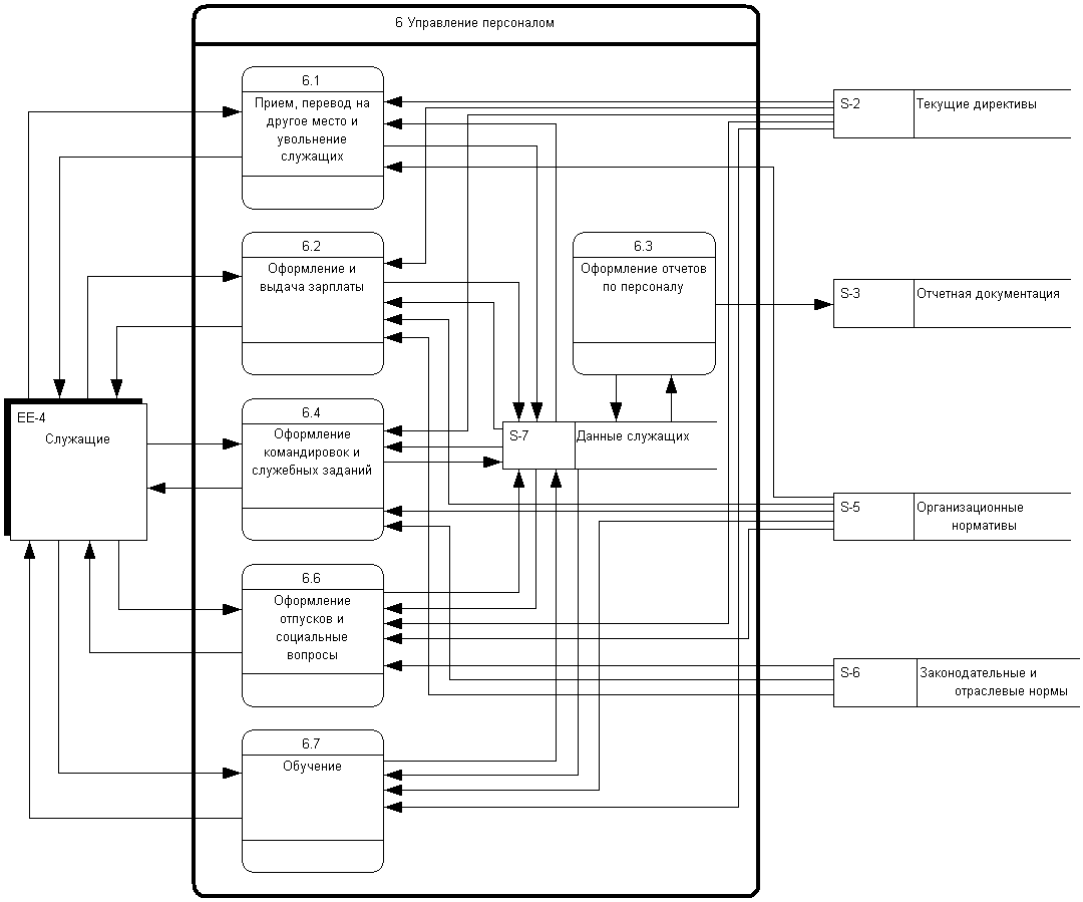


Рисунок 18. Детализация процесса "Управление персоналом".

Однако диаграммы вариантов использования несут несколько меньше информации по сравнению с соответствующими диаграммами потоков данных: на них процессы и хранилища в соответствии с принципом объединения данных и методов работы с ними объединяются в варианты использования, и остаются только связи между вариантами использования и действующими лицами (аналогом внешних сущностей). Для представления остальной информации каждый вариант использования может дополняться набором разнообразных диаграмм UML — диаграммами деятельностей, диаграммами сценариев, и пр. Обо всех этих видах диаграмм будет рассказано в лекции, посвященной архитектуре программного обеспечения.

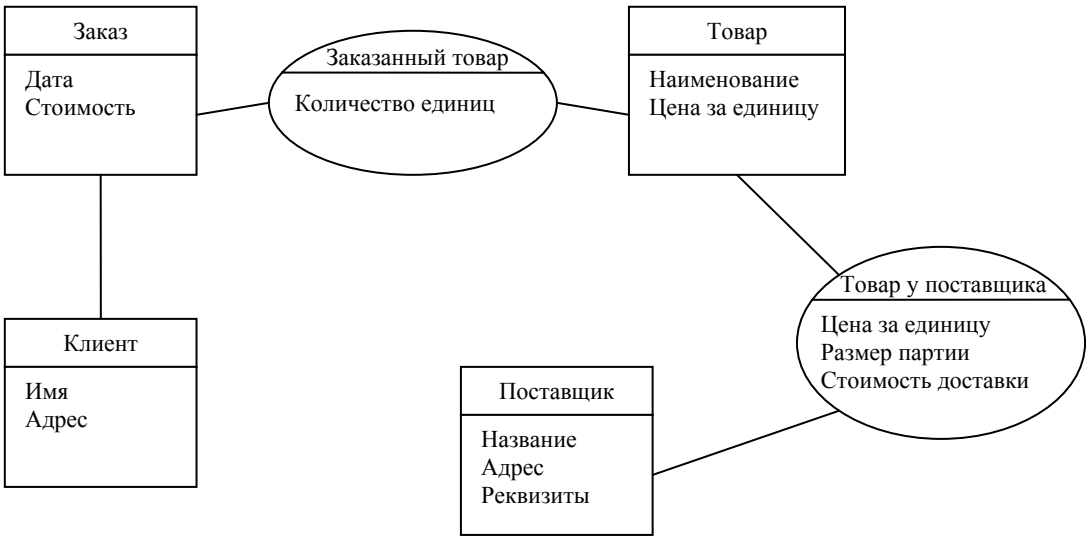


Рисунок 19. Модель сущностей и связей.

Выделение и анализ требований

После получения общего представления о деятельности и целях организаций, в которых будет работать будущая программная система, и о ее предметной области, можно определить более четко, какие именно задачи система будет решать. Кроме того, важно понимать, какие из задач стоят наиболее остро и обязательно должны быть поддержаны уже в первой версии, а какие могут быть отложены до следующих версий или вообще вынесены за рамки области ответственности системы. Эта информация выявляется при анализе потребностей возможных пользователей и заказчиков.

Потребности определяются на основе наиболее актуальных проблем и задач, которые пользователи и заказчики видят перед собой. При этом требуется аккуратное выявление значимых проблем, определение того, насколько хорошо они решаются при текущем положении дел, и расстановка приоритетов при рассмотрении недостаточно хорошо решаемых, поскольку чаще всего решить сразу все проблемы невозможно.

Формулировка потребностей может быть разбита на следующие этапы.

1. Выделить одну-две-три основных проблемы.
2. Определить причины возникновения проблем, оценить степень их влияния и выделить наиболее существенные из проблем, влекущие появление остальных.
3. Определить ограничения на возможные решения.

Формулировка потребностей не должна накладывать лишних ограничений на возможные решения, удовлетворяющие им. Нужно попытаться сформулировать, что именно является проблемой, а не предлагать сразу возможные решения.

Например, формулировки «система должна использовать СУБД Oracle для хранения данных», «нужно, чтобы при вводе неверных данных раздавался звуковой сигнал» не очень хорошо описывают потребности. Исключением в первом случае может быть особая ситуация, например, если СУБД Oracle уже используется для хранения других данных, которые должны быть интегрированы с рассматриваемыми: при этом ее использование становится внешним ограничением. Соответствующие потребности лучше описать так: «нужно организовать надежное и удобное для интеграции с другими системами хранение данных», «необходимо предотвращать попадание некорректных данных в хранилище».

При выявлении потребностей пользователей анализируются модели деятельности пользователей и организаций, в которых они работают, для выявления проблемных мест. Также используются такие приемы, как анкетирование, демонстрация возможных сеансов работы будущей системы, интерактивные опросы, где пользователям предоставляется возможность самим предложить варианты внешнего вида системы и ее работы или поменять предложенные кем-то другим, демонстрация прототипа системы и др.

После выделения основных потребностей нужно решить вопрос о разграничении области ответственности будущей системы, т.е. определить, какие из потребностей надо пытаться удовлетворить в ее рамках, а какие — нет.

При этом все заинтересованные лица делятся на пользователей, которые будут непосредственно использовать создаваемую систему для решения своих задач, и вторичных заинтересованных лиц, которые не решают своих задач с ее помощью, но чьи интересы так или иначе затрагиваются ею. Потребности пользователей нужно удовлетворить в первую очередь и на это нужно выделить больше усилий, а интересы вторичных заинтересованных лиц должны быть только адекватно учтены в итоговой системе.

На основе выделенных потребностей пользователей, отнесенных к области ответственности системы, формулируются возможные ***функции*** будущей системы, которые представляют собой услуги, предоставляемые системой и удовлетворяющие потребности одной или нескольких групп пользователей (или других заинтересованных лиц). Идеи для определения таких функций можно брать из имеющегося опыта разработчиков (наиболее часто используемый источник) или из результатов мозговых штурмов и других форм выработки идей.

Формулировка функций должна быть достаточно короткой, ясной для пользователей, без лишних деталей. Например:

- Все данные о сделках и клиентах будут сохраняться в базе данных.
- Статус выполнения заказа клиент сможет узнать через Интернет.
- Система будет поддерживать до 10000 одновременно работающих пользователей.
- Расписание проведения ремонтных работ будет строиться автоматически.

Предлагая те или иные функции, нужно уметь аккуратно оценивать их влияние на структуру и деятельность организаций, в рамках которых будет использоваться ПО. Это можно сделать, имея полученные при анализе предметной области модели их текущей деятельности.

Имея набор функций, достаточно хорошо поддерживающий решение наиболее существенных задач, с которыми придется работать разрабатываемой системе, можно составлять требования к ней, представляющие собой детализацию работы этих функций. Соотношение между проблемами, потребностями, функциями и требованиями изображено на Рис. 20.

Рисунок 20. Соотношение между проблемами, потребностями, функциями и требованиями.

При этом часто нужно учитывать, что ПО является частью программно-аппаратной системы, требования к которой надо преобразовать в требования к программной и аппаратной ее составляющим. В последнее время, в связи со значительным падением цен на мощное аппаратное обеспечение общего назначения, фокус внимания переместился, в основном, на программное обеспечение. Во многих проектах аппаратная платформа определяется из общих соображений, а поддержку большинства нужных функций осуществляет ПО.

Каждое требование раскрывает детали поведения системы при выполнении ею некоторой функции в некоторых обстоятельствах. При этом часть требований исходит из потребностей и пожеланий заинтересованных лиц и решений, удовлетворяющих эти потребности и пожелания, а часть — из внешних ограничений, накладываемых на систему, например, основными законами той предметной области, в рамках которой системе придется работать, государственным законодательством, корпоративной политикой и пр.

Еще до перехода от функций к требованиям полезно расставить приоритеты и оценить трудоемкость их реализации и рискованность. Это позволит отказаться от реализации наименее важных и наиболее трудоемких, не соответствующих бюджету проекта функций еще до их детальной проработки, а также выявить возможные проблемные места проекта — наиболее трудоемкие и неясные из вошедших в него функций.

Правила работы с требования к ПО и более общими системными требованиями (к программно-аппаратной системе), определяются следующими двумя стандартами IEEE.

- **IEEE 830-1998 Recommended Practice for Software Requirements Specifications** [7] (рекомендуемые методы спецификации требований к ПО)**.**
  Описывает структуру документов для фиксации требований к ПО.

Кроме того, он определяет характеристики, которыми должен обладать правильно составленный набор требований.

- Корректность или адекватность (соответствие реальным потребностям).
- Недвусмысленность (однозначность понимания).
- Полнота (отражение всех выделенных потребностей и всех возможных ситуаций, в которых придется работать системе).
- Непротиворечивость (согласованность между различными элементами).
- Упорядоченность по важности и стабильности.
- Проверяемость (выполнение каждого требования нужно уметь проверять некоторым достаточно эффективным способом — непроверяемые требования должны быть удалены из рассмотрения или сведены к проверяемым вариантам).
- Модифицируемость (оформление в удобных для внесения изменений структуре и стилях).
- Прослеживаемость в ходе разработки (возможность увязать требование с подсистемами, модулями и операциям, ответственными за его выполнение, и с тестами, проверяющими его выполнение).

- **IEEE 1233-1998, 2002 Guide for Developing System Requirements Specifications** [8] (руководство по разработке спецификаций требований к системам)**.**
  Описывает правила построения требований для программно-аппаратных систем в целом. Он выделяет следующие необходимые свойства набора требований.
  - Однократное упоминание отдельных требований.
  - Отсутствие пересечений между требованиями.
  - Явное указание связей между требованиями.
  - Полнота.
  - Непротиворечивость.
  - Определение ограничений, области действия и контекста для каждого требования.
  - Модифицируемость.
  - Конфигурируемость, удобство поддержки.
  - Подходящий для определения системы уровень абстракции.

  Кроме того, следующие свойства считаются необходимыми для отдельного требования.
  - Абстрактность — формулировка, независимая от возможных реализаций.
  - Недвусмысленность.
  - Прослеживаемость.
  - Проверяемость.

  Стандарт предписывает определять следующие атрибуты для каждого требования.
  - Уникальный идентификатор.
  - Приоритет, важность реализации с точки зрения пользователей.
  - Критичность для построения и успешности системы с точки зрения аналитиков.
  - Осуществимость с точки зрения готовности пользователей к новой функции, имеющихся технологий и стоимости реализации.
  - Риски высокой стоимости, последствий использования для окружающей среды и пользователей, конфликтов со стандартами и законодательством.
  - Источник (т.е. кто предложил это требование).
  - Тип требования. Возможные типы определяются так (многие из них соответствуют атрибутам качества, рассматриваемым в следующей лекции):
    - Требования на входные данные.
    - Требования на выходные данные.

- Надежность (reliability, например, среднее время работы между отказами).
- Работоспособность (availability, например, необходимое отношение времени функционирования к полному времени работы).
- Удобство сопровождения (maintainability, например, удобство замены компонента).
- Производительность (performance, например, среднее время ожидания ответа).
- Доступность (accessibility, например, разные способы доступа для новичков и опытных пользователей).
- Ограничения окружающей среды (например, максимальный уровень задымленности, при котором гарантируется работоспособность).
- Эргономичность (ergonomic, например, использование набора цветов, понижающих утомляемость глаз).
- Безопасность (safety, например, допустимые уровни электромагнитного излучения различных частот).
- Защищенность (security, например, ограничения доступа для разных пользователей).
- Требования к оборудованию (например, использование обычной электросети).
- Транспортируемость (transportability, например, ограничения веса).
- Удобство обучения (например, включение обучающих материалов).
- Документированность (например, наличие встроенной документации).
- Внешние интерфейсы (например, поддержка стандартных форматов документов).
- Тестируемость (например, поддержка удаленной диагностики).
- Условия необходимого качества (например, максимально допустимая погрешность производимых измерений).
- Следование корпоративным и законодательным нормам (например, законам об охране труда).
- Совместимость с известными системами.
- Следование стандартам и технологическим нормам.
- Конвертация данных (например, из старой версии системы).
- Возможности роста (например, возможное увеличение числа пользователей).
- Удобство развертывания (например, время, необходимое для приведения в работоспособное состояние).

В дополнение к перечисленному стандарт IEEE 1233 выделяет следующие ошибки, которых необходимо избегать при определении требований.

- Описание возможных решений вместо требований. Эта информация важна, но должна оформляться в отдельных документах.
- Слишком детальные спецификации, описывающие требования к слишком мелким элементам системы или описывающие требования, в точности соответствующие характеристикам определенных систем.
- Слишком сильные ограничения, не вытекающие из реальных потребностей.
- Нечеткие требования, которые могут быть непроверяемыми и субъективными («минимизировать уровень погрешности», «удобный для пользователей интерфейс»), или сформулированы в виде, открытом для пополнения неопределенными элементами (с указанием «и т.д.» или «включая, но не ограничиваясь следующим...»).
- Несформулированные предположения о режимах работы, свойствах окружения, о готовности других систем или принятии законов и стандартов, находящихся в стадии разработки.

Варианты использования

Наиболее широко распространенными техниками фиксации требований в настоящий момент являются структурированные текстовые документы и диаграммы *вариантов использования*, о которых уже заходила речь при обсуждении RUP.

Вариантом использования (use case) называют некоторый сценарий действий системы, который обеспечивает ощутимый и значимый для ее пользователей результат. На практике в виде

одного варианта использования оформляется сценарий действий системы, который будет, скорее всего, неоднократно возникать во время ее работы и имеет достаточно четко определенные условия начала выполнения и завершения.

Примеры вариантов использования:

- Покупатель в Интернет-магазине выбирает товар. Для этого он может выбрать категорию товара, фирму-изготовителя или группу таких фирм и отфильтровать оставшиеся товары по цене, габаритам и цвету. Определившись, он выбирает товар, кликая на соответствующем значке мышкой.
- Оператор системы контроля качества газопровода ищет участки газопровода с повышенным риском возникновения аварии. Для этого он выбирает группу ранее случившихся аварий, фильтруя их по дате, нанесенному ущербу, типу аварии и запускает процедуру анализа характеристик соответствующих участков газопровода на совпадение, по крайней мере, двух характеристик. При таком анализе учитываются изготовитель труб и их партия, история хранения труб на складах, землепроходческая бригада, бригада сварщиков, показатели нескольких последних проведенных инспекций, показатели химической активности грунтов, наличие близлежащих предприятий, влияющих на химические и электрические характеристики грунтов. После этого на карте выделяются участки, характеристики которых также попадают под найденный «шаблон аварии».

В языке UML вариант использования изображается в виде овала, помеченного именем представляемого варианта. Варианты использования могут быть связаны с участвующими в них ***действующими лицами (actors)***, изображаемыми в виде человечков и представляющими различные роли пользователей системы или внешние системы, взаимодействующие с ней.

Варианты использования могут быть связаны друг с другом тремя видами связей: *обобщением (generalization)*, *расширением (extend relationship)* и *включением (include relationship)*. Действующие лица также могут быть связаны друг с другом с помощью связей *обобщения (generalization)*.

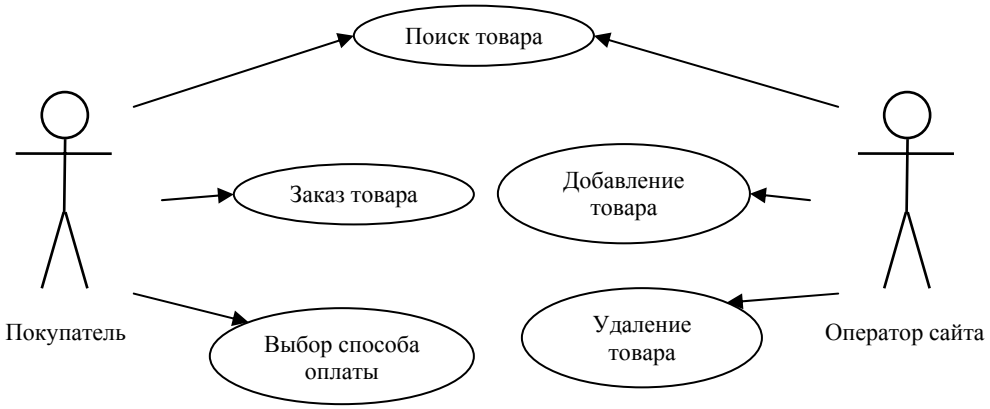


Рисунок 21. Набросок диаграммы вариантов использования для Интернет-магазина.

Если несколько вариантов использования имеют много общего в структуре выполняемых в их рамках сценариев и в достигаемых целях, можно выделить ***обобщающий*** их вариант использования, содержащий общие части описываемого ими поведения. Обобщаемые варианты ***уточняют*** обобщающий их вариант. При этом обычно сценарий работы обобщаемого варианта состоит из нескольких кусков — последовательности действий, выполняемых в рамках сценария работы общего варианта использования, перемежаются с последовательностями, специфическими для частного. Например, если система регистрации заказов в магазине позволяет оформить заказ (данные о котором в дальнейшем будут присутствовать в системе) как при помощи сайта магазина, так и по телефону, то варианты использования «Заказ товара через сайт» и «Заказ товара по телефону» могут быть обобщены в варианте «Заказ товара».

Вариант использования А ***расширяет (extends)*** другой вариант использования В, если в ходе сценария работы А при определенных условиях надо включить полный сценарий работы В.

Например, оператор сайта магазина может удалить товар, введя его идентификатор; а если идентификатор ему не известен, а известна лишь марка товара и производитель, он должен сначала найти такой товар и определить идентификатор в его описании, а затем уже удалить товар. Соответственно, вариант использования «Удаление товара» будет расширять вариант использования «Поиск товара».

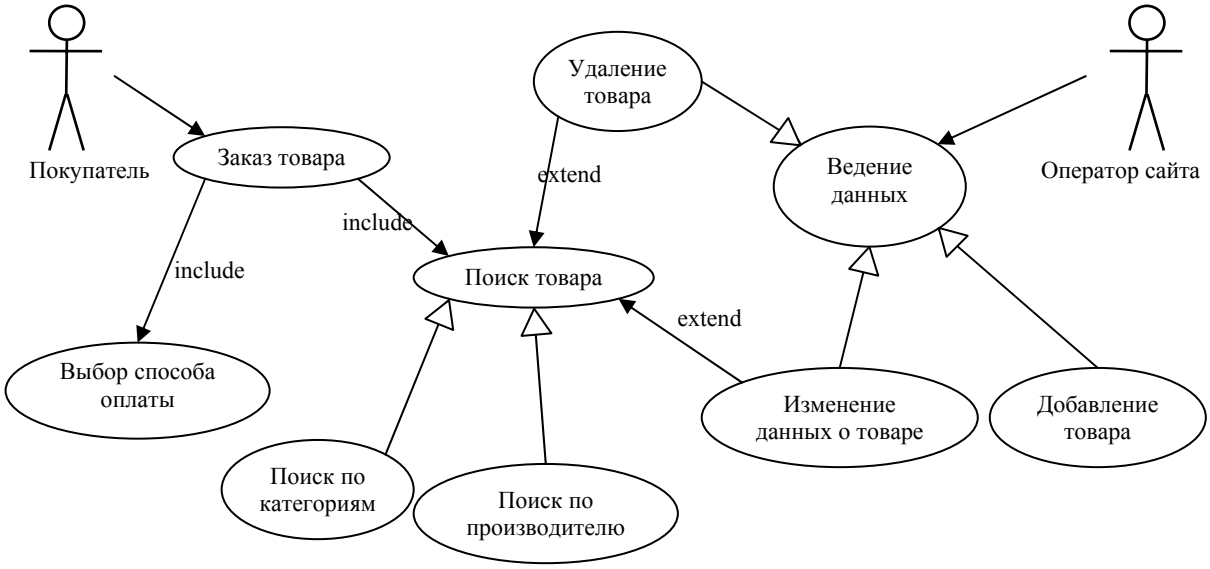


Рисунок 22. Доработанная диаграмма вариантов использования для Интернет-магазина.

Вариант использования A ***включает (includes, или использует, uses)*** вариант использования B, если A всегда в некоторый момент включает полностью сценарий работы B. Например, при оформлении заказа покупатель всегда должен определить способ его оплаты. Значит, вариант использования «Заказ товара» включает вариант «Определение способа оплаты».

Обобщение между действующими лицами вводится, если задачи, решаемые одним действующим лицом с помощью данной системы, являются подмножеством задач, решаемых другим действующим лицом. Например, обычный оператор сайта может иметь права только на внесение дополнений и изменений в данные, но не иметь прав на приостановку работы сайта и изменение структуры, которые имеет администратор сайта. В то же время администратор может делать все, что может обычный оператор сайта. Соответственно, администратор сайта является специальным частным случаем оператора.

Хорошо описанный вариант использования имеет следующие атрибуты [9].

1. Имя, ясно говорящее о назначении варианта использования.
2. Описание. Несколько предложений, описывающих этот вариант использования.
3. Частота. Насколько часто данный вариант использования возникает.
4. Предусловия. Все условия запуска варианта использования.
5. Постусловия. Все условия, которые должны быть выполнены после успешного выполнения варианта использования.
6. Основной сценарий работы, который используется в большинстве случаев.
7. Альтернативные сценарии, возникающие иногда. Для каждого альтернативного сценария указываются условия его запуска.
8. (Необязательно) Задействованные действующие лица.
9. (Необязательно) Расширяемые варианты использования.
10. (Необязательно) Включаемые варианты использования.
11. (Необязательно) Статус: «в разработке», «готов к проверке», «в процессе проверки», «подтвержден», «отвергнут».
12. (Необязательно) Допущения об окружении и ходе работы системы, использованные при разработке данного варианта. В полностью готовом варианте эти допущения либо должны

быть подтверждены и стать ограничениями системы, либо должны давать начало различным сценариям работы.

Кроме того, варианты использования могут дополняться диаграммами других видов — прежде всего, сценарными диаграммами и диаграммами активностей, описывающими последовательности действий участвующих компонентов, диаграммами состояний и переходов компонентов и диаграммами классов этих компонентов, и др. Все эти виды диаграмм будут рассматриваться в лекции, посвященной архитектуре ПО.

Литература к Лекции 4

[1] J. A. Zachman. A Framework for Information Systems Architecture. IBM Systems Journal, vol. 26, no. 3, pp. 276-292, 1987.

[2] J. F. Sowa and J. A. Zachman. Extending and Formalizing the Framework for Information Systems Architecture. IBM Systems Journal, vol. 31, no. 3, pp. 590-616, 1992.

[3] E. Yourdon. Modern Structured Analysis. Prentice Hall, 1988.

[4] T. DeMarco. Structured Analysis and System Specification. A Yourdon Book, Yourdon Inc., NY, 1979.

[5] C. Sarson, T. Gane. Structured Systems Analysis. Englewood Cliffs, NJ.: Prentice-Hall, 1979.

[6] P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. ACM Transactions on Database Systems I (I). March 1976, pp. 8-46.

[7] IEEE 830-1998. Recommended Practice for Software Requirements Specifications. New York: IEEE, 1998.

[8] IEEE 1233-1998. Guide for Developing System Requirements Specifications. New York: IEEE, 1998.

[9] А. Коберн. Современные методы описания требований к системам. М.: Лори, 2002.

[10] И. Соммервилл. Инженерия программного обеспечения. М.: Вильямс, 2002.

[11] Э. Дж. Брауде. Технология разработки программного обеспечения. СПб.: Питер, 2004.

[12] Д. Леффингуэлл, Д. Уидриг. Принципы работы с требованиями к программному обеспечению. Унифицированный подход. М.: Вильямс, 2002.

[13] А. Якобсон, Г. Буч, Дж. Рамбо. Унифицированный процесс разработки программного обеспечения. СПб.: Питер, 2002.

Лекция 5. Качество ПО и методы его контроля

Аннотация

Рассматривается понятие качества ПО, характеристики и атрибуты качества, связь атрибутов качества с требованиями. Дается краткий обзор различных методов контроля качества ПО, с более детальным рассмотрением тестирования и проверки свойств на моделях.

Ключевые слова

Качество ПО, функциональность, надежность, удобство использования, производительность, удобство сопровождения, переносимость, методы контроля качества, тестирование, проверка свойств ПО на моделях, ошибки в ПО.

Текст лекции

Качество программного обеспечения

Как проверить, что требования определены достаточно полно и описывают все, что ожидается от будущей программной системы? Это можно сделать, проследив, все ли необходимые аспекты *качества ПО* отражены в них. Именно понятие качественного ПО соответствует представлению о том, что программа достаточно успешно справляется со всеми возложенными на нее задачами и не приносит проблем ни конечным пользователям, ни их начальству, ни службе поддержки, ни специалистам по продажам. Да и самим разработчикам создание качественной программы приносит гораздо больше удовольствия.

Если попросить группу людей высказать своё мнение по поводу того, что такое качественное ПО, можно получить следующие варианты ответов.

- Его легко использовать.
- Оно демонстрирует хорошую производительность.
- В нем нет ошибок.
- Оно не портит пользовательские данные при сбоях.
- Его можно использовать на разных платформах.
- Оно может работать 24 часа в сутки и 7 дней в неделю.
- В него легко добавлять новые возможности.
- Оно удовлетворяет потребности пользователей.
- Оно хорошо документировано.

Все это действительно имеет непосредственное отношение к качеству ПО. Но эти ответы выделяют характеристики, важные для конкретного пользователя, разработчика или группы таких лиц. Для того, чтобы удовлетворить потребности всех сторон (конечных пользователей, заказчиков, разработчиков, администраторов систем, в которых оно будет работать, регулирующих организаций и пр.), для достижения прочного положения разрабатываемого ПО на рынке и повышения потенциала его развития необходим учет всей совокупности характеристик ПО, важных для всех заинтересованных лиц.

Приведенные выше ответы показывают, что качество ПО может быть описано большим набором разнородных характеристик. Такой подход к описанию сложных понятий называется *холистическим* (от греческого слова ολοσ, целое). Он не дает единой концептуальной основы для рассмотрения затрагиваемых вопросов, какую дает целостная система представлений (например, Ньтоновская механика в физике или классическая теория вычислимости на основе машин Тьюринга), но позволяет, по крайней мере, не упустить ничего существенного.

Качество программного обеспечения определяется в стандарте ISO 9126 [1] как вся совокупность его характеристик, относящихся к возможности удовлетворять высказанные или подразумеваемые потребности всех заинтересованных лиц.

Тот же стандарт ISO 9126 [1-4] дает следующее представление качества.

Различаются понятия ***внутреннего качества***, связанного с характеристиками ПО самого по себе, без учета его поведения; ***внешнего качества***, характеризующего ПО с точки зрения его поведения; и ***качества ПО при использовании*** в различных контекстах — того качества, которое ощущается пользователями при конкретных сценариях работы ПО. Для всех этих аспектов качества введены метрики, позволяющие оценить их. Кроме того, для создания добротного ПО существенно *качество технологических процессов* его разработки. Взаимоотношения между этими аспектами качества по схеме, принятой ISO 9126, показано на Рис. 23.

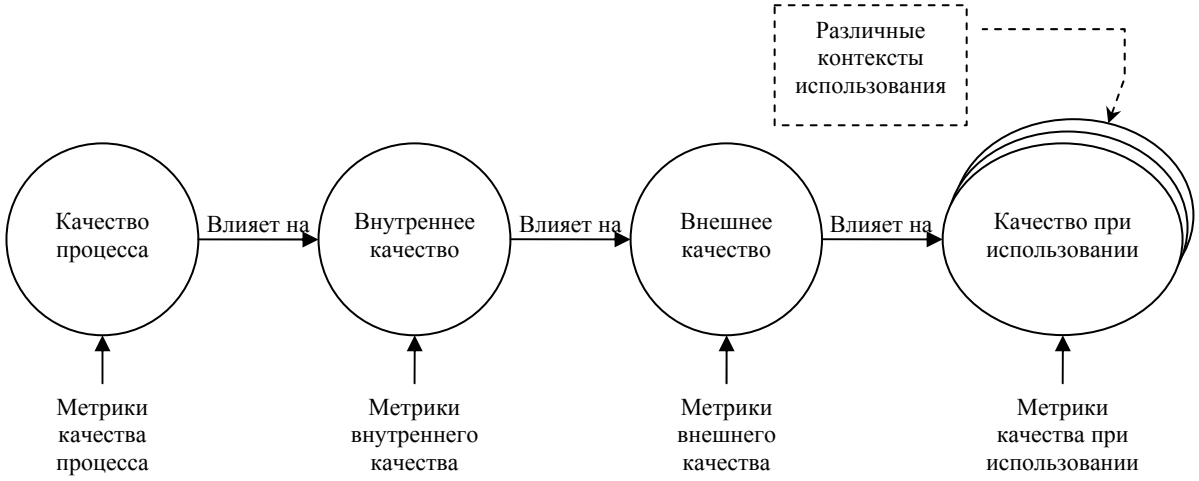


Рисунок 23. Основные аспекты качества ПО по ISO 9126.

Общие принципы обеспечения качества процессов производства во всех отраслях экономики регулируются набором стандартов ISO 9000. Наиболее важные для разработки ПО стандарты в его составе следующие.

- **ISO 9000:2000 Quality management systems — Fundamentals and vocabulary** [5].
  Системы управления качеством — Основы и словарь. (Аналог ГОСТ Р-2001).
- **ISO 9001:2000 Quality management systems — Requirements. Models for quality assurance in design, development, production, installation, and servicing** [6].
  Системы управления качеством — Требования. Модели для обеспечения качества при проектировании, разработке, коммерциализации, установке и обслуживании.
  Определяет общие правила обеспечения качества результатов во всех процессах жизненного цикла. (Аналог ГОСТ Р-2001).
  - Этот стандарт выделяет следующие процессы
    - Управление качеством.
    - Управление ресурсами.
    - Развитие системы управления.
    - Исследования рынка.
    - Проектирование продуктов.
    - Приобретения.
    - Производство.
    - Оказание услуг.
    - Защита продуктов.
    - Оценка потребностей заказчиков.
    - Поддержка коммуникаций с заказчиками.
    - Поддержка внутренних коммуникаций.
    - Управление документацией.
    - Ведение записей о деятельности.
    - Планирование.
    - Обучение персонала.
    - Внутренние аудиты.
    - Оценки управления.

        - Мониторинг и измерения.
        - Управление несоответствиями.
        - Постоянное совершенствование.
        - Управление и развитие системы в целом.
    - Для каждого процесса требуется иметь планы развития процесса, состоящие как минимум из следующих разделов.
        - Проектирование процесса.
        - Документирование процесса.
        - Реализация процесса.
        - Поддержка процесса.
        - Мониторинг процесса.
        - Управление процессом.
        - Усовершенствование процесса.
    - Помимо поддержки и развития системы процессов, нацеленных на удовлетворение нужд заказчиков и пользователей, ISO 9001 требует:
        - Определить, документировать и развивать собственную систему качества на основе измеримых показателей.
        - Использовать эту систему качества как средство управления процессами, нацеливая их на большее удовлетворение нужд заказчиков, планируя и постоянно отслеживая качество результатов всех видов деятельности, в том числе и самого управления.
        - Обеспечить использование качественных ресурсов, качественного (компетентного, профессионального) персонала, качественной инфраструктуры и качественного окружения.
        - Постоянно контролировать соблюдение требований к качеству на практике, во всех процессах проектирования, производства, предоставления услуг и при приобретениях.
        - Предусмотреть процесс устранения дефектов, определить и контролировать качество результатов этого процесса.

  Ранее использовавшиеся стандарты ISO 9002:1994 Quality systems — Model for quality assurance in production, installation and servicing и ISO 9003:1994 Quality systems — Model for quality assurance in final inspection and test в 2000 году были заменены соответствующими им частями ISO 9001.

- **ISO 9004:2000 Quality management systems — Guidelines for performance improvements** [7].
  Системы управления качеством. Руководство по улучшению деятельности. (Аналог ГОСТ Р-2001).
- **ISO/IEC 90003:2004 Software engineering — Guidelines for the application of ISO 9001:2000 to computer software** [8]**.**
  Руководящие положения по применению стандарта ISO 9001 при разработке, поставке и обслуживании программного обеспечения.
  Этот стандарт конкретизирует положения ISO 9001 для разработки программных систем, с упором на обеспечение качества при процессе проектирования. Он также определяет некоторый набор техник и процедур, которые рекомендуется применять для контроля и обеспечения качества разрабатываемых программ.

Стандарт ISO 9126 [1-4] предлагает использовать для описания внутреннего и внешнего качества ПО многоуровневую модель. На верхнем уровне выделено 6 основных характеристик качества ПО. Каждая характеристика описывается при помощи нескольких входящих в нее *атрибутов*. Для каждого атрибута определяется набор метрик, позволяющих его оценить. Множество характеристик и атрибутов качества согласно ISO 9126 показано на Рис. 24.

Рисунок 24. Характеристики и атрибуты качества ПО по ISO 9126.

Ниже приведены определения этих характеристик и атрибутов по стандарту ISO 9126:2001.

- ***Функциональность (functionality)***.
  Способность ПО в определенных условиях решать задачи, нужные пользователям. Определяет, что именно делает ПО, какие задачи оно решает.
  - ***Функциональная пригодность (suitability)***.
    Способность решать нужный набор задач.
  - ***Точность (accuracy)***.
    Способность выдавать нужные результаты.
  - ***Способность к взаимодействию (interoperability)***.
    Способность взаимодействовать с нужным набором других систем.
  - ***Соответствие стандартам и правилам (compliance)***.
    Соответствие ПО имеющимся индустриальным стандартам, нормативным и законодательным актам, другим регулирующим нормам.
  - ***Защищенность (security)***.
    Способность предотвращать неавторизированный, т.е. без указания лица, пытающегося его осуществить, и не разрешенный доступ к данным и программам.
- ***Надежность (reliability)***.
  Способность ПО поддерживать определенную работоспособность в заданных условиях.
  - ***Зрелость, завершенность (maturity)***.
    Величина, обратная частоте отказов ПО. Обычно измеряется средним временем работы без сбоев и величиной, обратной вероятности возникновения отказа за данный период времени.
  - ***Устойчивость к отказам (fault tolerance)***
    Способность поддерживать заданный уровень работоспособности при отказах и нарушениях правил взаимодействия с окружением.
  - ***Способность к восстановлению (recoverability)***.
    Способность восстанавливать определенный уровень работоспособности и целостность данных после отказа, необходимые для этого время и ресурсы.
  - ***Соответствие стандартам надежности (reliability compliance)***.
    Этот атрибут добавлен в 2001 году.

- ***Удобство использования (usability) или практичность***.
  Способность ПО быть удобным в обучении и использовании, а также привлекательным для пользователей.
  - ***Понятность (understandability)***.
    Показатель, обратный к усилиям, которые затрачиваются пользователями на восприятие основных понятий ПО и осознание их применимости для решения своих задач.
  - ***Удобство обучения (learnability)***.
    Показатель, обратный усилиям, затрачиваемым пользователями на обучение работе с ПО.
  - ***Удобство работы (operability)***.
    Показатель, обратный усилиям, предпринимаемым пользователями для решения своих задач с помощью ПО.
  - ***Привлекательность (attractiveness)***.
    Способность ПО быть привлекательным для пользователей. Этот атрибут добавлен в 2001.
  - ***Соответствие стандартам удобства использования (usability compliance)***.
    Этот атрибут добавлен в 2001.
- ***Производительность (efficiency) или эффективность***.
  Способность ПО при заданных условиях обеспечивать необходимую работоспособность по отношению к выделяемым для этого ресурсам. Можно определить ее и как отношение получаемых с помощью ПО результатов к затрачиваемым на это ресурсам всех типов.
  - ***Временная эффективность (time behaviour)***.
    Способность ПО выдавать ожидаемые результаты, а также обеспечивать передачу необходимого объема данных за отведенное время.
  - ***Эффективность использования ресурсов (resource utilisation)***.
    Способность решать нужные задачи с использованием определенных объемов ресурсов определенных видов. Имеются в виду такие ресурсы, как оперативная и долговременная память, сетевые соединения, устройства ввода и вывода, и пр.
  - ***Соответствие стандартам производительности (efficiency compliance)***.
    Этот атрибут добавлен в 2001.
- ***Удобство сопровождения (maintainability)***.
  Удобство проведения всех видов деятельности, связанных с сопровождение программ.
  - ***Анализируемость (analyzability) или удобство проведения анализа***.
    Удобство проведения анализа ошибок, дефектов и недостатков, а также удобство анализа необходимости изменений и их возможных последствий.
  - ***Удобство внесения изменений (changeability)***.
    Показатель, обратный трудозатратам на выполнение необходимых изменений.
  - ***Стабильность (stability)***.
    Показатель, обратный риску возникновения неожиданных эффектов при внесении необходимых изменений.
  - ***Удобство проверки (testability)***.
    Показатель, обратный трудозатратам на проведение тестирования и других видов проверки того, что внесенные изменения привели к нужным результатам.
  - ***Соответствие стандартам удобства сопровождения (maintainability compliance)***.
    Этот атрибут добавлен в 2001.
- ***Переносимость (portability)***.
  Способность ПО сохранять работоспособность при переносе из одного окружения в другое, включая организационные, аппаратные и программные аспекты окружения.
  Иногда эта характеристика называется в русскоязычной литературе мобильностью. Однако термин «мобильность» стоит зарезервировать для перевода «mobility» — способности ПО и

компьютерной системы в целом сохранять работоспособность при ее физическом перемещении в пространстве.

- ***Адаптируемость (adaptability)***.
  Способность ПО приспосабливаться к различным окружениям без проведения для этого действий, помимо заранее предусмотренных.
- ***Удобство установки (installability)***.
  Способность ПО быть установленным или развернутым в определенном окружении.
- ***Способность к сосуществованию (coexistence)***.
  Способность ПО сосуществовать с другими программами в общем окружении, деля с ними ресурсы.
- ***Удобство замены (replaceability) другого ПО данным***.
  Возможность применения данного ПО вместо других программных систем для решения тех же задач в определенном окружении.
- ***Соответствие стандартам переносимости (portability compliance)***.
  Этот атрибут добавлен в 2001.

Перечисленные атрибуты относятся к внутреннему и внешнему качеству ПО согласно ISO 9126. Для описания качества ПО при использовании стандарт ISO 9126-4 [4] предлагает другой, более узкий набор характеристик.

- ***Эффективность (effectiveness)***.
  Это способность ПО предоставлять пользователям возможность решать их задачи с необходимой точностью при использовании в заданном контексте.
- ***Продуктивность (productivity)***.
  Способность ПО предоставлять пользователям определенные результаты в рамках ожидаемых затрат ресурсов.
- ***Безопасность (safety)***.
  Способность ПО обеспечивать необходимо низкий уровень риска нанесения ущерба жизни и здоровью людей, бизнесу, собственности или окружающей среде.
- ***Удовлетворение пользователей (satisfaction)***.
  Способность ПО приносить удовлетворение пользователям при использовании в заданном контексте.

Помимо перечисленных характеристик и атрибутов качества стандарт ISO 9126:2001 определяет наборы метрик для оценки каждого атрибута. Приведем следующие примеры таких метрик.

- *Полнота реализации функций* — процент реализованных функций по отношению к перечисленным в требованиях. Используется для измерения функциональной пригодности.
- *Корректность реализации функций* — правильность их реализации по отношению к требованиям. Используется для измерения функциональной пригодности.
- *Отношение числа обнаруженных дефектов к прогнозируемому*. Используется для определения зрелости.
- *Отношение числа проведенных тестов к общему их числу*. Используется для определения зрелости.
- *Отношение числа доступных проектных документов к указанному в их списке*. Используется для измерения удобства проведения анализа.
- *Наглядность и полнота документации*. Используется для оценки понятности.

Перечисленные характеристики и атрибуты качества ПО позволяют систематически описывать требования к нему, определяя, какие свойства ПО по данной характеристике хотят видеть заинтересованные стороны. Таким образом, требования должны определять следующее.

- Что ПО должно делать, например:
  позволять клиенту оформить заказы и обеспечить их доставку;

обеспечивать контроль качества строительства и отслеживать проблемные места;
поддерживать нужные характеристики автоматизированного процесса производства, предотвращая аварии и оптимальным образом используя имеющиеся ресурсы.

- Насколько оно должно быть надежно, например:
  работать 7 дней в неделю и 24 часа в сутки;
  допускается неработоспособность в течение не более 3 часов в год;
  никакие введенные пользователями данные при отказе не должны теряться.
- Насколько им должно быть удобно пользоваться, например:
  покупатель должен, зная название товара и имея средние навыки работы в Интернет, находить нужный ему товар за не более чем 2 минуты;
  инженер по специальности «строительство мостов» должен в течение одного дня уметь разобраться в 80% функций системы.
- Насколько оно должно быть эффективно, например:
  поддерживать обслуживание до 10000 запросов в секунду;
  время отклика на запрос при максимальной загрузке не должно превышать 3 с;
  время реакции на изменение параметров процесса производства не должно превышать 0.1 с;
  на обработку одного запроса не должно тратиться более 1 MB оперативной памяти.
- Насколько удобно должно быть его сопровождение, например:
  добавление в систему нового вида запросов не должно требовать более 3 человеко-дней;
  добавление поддержки нового этапа процесса производства не должно стоить более $20000.
- Насколько оно должно быть переносимо, например:
  ПО должно работать на операционных системах Linux, Windows XP и MacOS X;
  ПО должно работать с документами в форматах MS Word 97 и HTML;
  ПО должно сохранять файлы отчетов в форматах MS Word 2000, MS Excel 2000, HTML, RTF и в виде обычного текста;
  ПО должно сопрягаться с существующей системой записи данных о заказах.

Приведенные атрибуты качества закреплены в стандартах, но это не значит, что они вполне исчерпывают понятие качества ПО. Так, в стандарте ISO 9126 отсутствуют характеристики, связанные с ***мобильностью ПО (mobility)***, т.е. способностью программы работать при физических перемещениях машины, на которой она работает. Вместо надежности многие исследователи предпочитают рассматривать более общее понятие ***добротности (dependability)***, описывающее способность ПО поддерживать определенные показатели качества по основным характеристикам (функциональности, производительности, удобству использования) с заданными вероятностями выхода за их рамки и определенным максимальным ущербом от возможных нарушений. Кроме того, активно исследуются понятия удобства использования, безопасности и защищенности ПО, — они кажутся большинству специалистов гораздо более сложными, чем это описывается данным стандартом.

Методы контроля качества

Как контролировать качество системы? Как точно узнать, что программа делает именно то, что нужно, и ничего другого? Как определить, что она достаточно надежна, переносима, удобна в использовании? Ответы на эти вопросы можно получить с помощью процессов верификации и валидации.

- ***Верификация*** обозначает проверку того, что ПО разработано в соответствии со всеми требованиями к нему, или что результаты очередного этапа разработки соответствуют ограничениям, сформулированным на предшествующих этапах.
- ***Валидация*** — это проверка того, что сам продукт правилен, т.е. подтверждение того, что он действительно удовлетворяет потребностям и ожиданиям пользователей, заказчиков и других заинтересованных сторон.

Эффективность верификации и валидации, как и эффективность разработки ПО в целом, зависит от полноты и корректности формулировки требований к программному продукту.

Основой любой системы обеспечения качества являются методы его обеспечения и контроля. ***Методы обеспечения качества*** [9] представляют собой техники, гарантирующие достижение определенных показателей качества при их применении. Мы будем рассматривать подобные методы на протяжении всего курса.

Методы контроля качества позволяют убедиться, что определенные характеристики качества ПО достигнуты. Сами по себе они не могут помочь их достижению, они лишь помогают определить, удалось ли получить в результате то, что хотелось, или нет, а также найти ошибки, дефекты и отклонения от требований. Методы контроля качества ПО можно классифицировать следующим образом.

- Методы и техники, связанные с выяснением свойств ПО во время его работы.
  Это, прежде всего, все виды *тестирования*, а также *профилирование* и измерение количественных показателей качества, которые можно определить по результатам работы ПО — эффективности по времени и другим ресурсам, надежности, доступности и пр.
- Методы и техники определения показателей качества на основе симуляции работы ПО с помощью моделей разного рода.
  К этому виду относятся *проверка на моделях (model checking)*, а также *прототипирование (макетирование)*, используемое для оценки качества принимаемых решений.
- Методы и техники, нацеленные на выявление нарушений формализованных правил построения исходного кода ПО, проектных моделей и документации.
  К методам такого рода относится *инспектирование кода*, заключающееся в целенаправленном поиске определенных дефектов и нарушений требований в коде на основе набора шаблонов, автоматизированные методы поиска ошибок в коде, не основанные на его выполнении, методы проверки документации на согласованность и соответствие стандартам.
- Методы и техники обычного или формализованного анализа проектной документации и исходного кода для выявления их свойств.
  К этой группе относятся многочисленные методы *анализа архитектуры ПО*, о которых пойдет речь в следующей лекции, методы формального доказательства свойств ПО и формального анализа эффективности применяемых алгоритмов.

Далее мы несколько подробнее рассмотрим тестирование и проверку на моделях как примеры методов контроля качества.

Тестирование

Тестирование — это проверка соответствия ПО требованиям, осуществляемая с помощью наблюдения за его работой в специальных, искусственно построенных ситуациях. Такого рода ситуации называют тестовыми или просто ***тестами***.

Тестирование — конечная процедура. Набор ситуаций, в которых будет проверяться тестируемое ПО, всегда конечен. Более того, он должен быть настолько мал, чтобы тестирование можно было провести в рамках проекта разработки ПО, не слишком увеличивая его бюджет и сроки. Это означает, что при тестировании всегда проверяется очень небольшая доля всех возможных ситуаций. По этому поводу Дейкстра (Dijkstra) заметил, что тестирование позволяет точно определить, что в программе есть ошибка, но не позволяет утверждать, что там нет ошибок.

Тем не менее, тестирование может использоваться для достаточно уверенного вынесения оценок о качестве ПО. Для этого необходимо иметь ***критерии полноты тестирования***, описывающие важность различных ситуаций для оценки качества, а также эквивалентности и зависимости между ними. Этот критерий может утверждать, что все равно в какой из ситуаций, A или B, проверять правильность работы ПО, или, если программа правильно работает в ситуации A, то, скорее всего, в ситуации B все тоже будет правильно. Часто критерий полноты тестирования задается при помощи определения эквивалентности ситуаций, дающей конечный

набор классов ситуаций. В этом случае считают, что все равно, какую из ситуаций одного класса использовать в качестве теста. Такой критерий называют ***критерием тестового покрытия***, а процент классов эквивалентности ситуаций, случившихся во время тестирования, — достигнутым ***тестовым покрытием***.

Таким образом, основные задачи тестирования: построить такой набор ситуаций, который был бы достаточно представителен и позволял бы завершить тестирование с достаточной степенью уверенности в правильности ПО вообще, и убедиться, что в конкретной ситуации ПО работает правильно, в соответствии с требованиями.

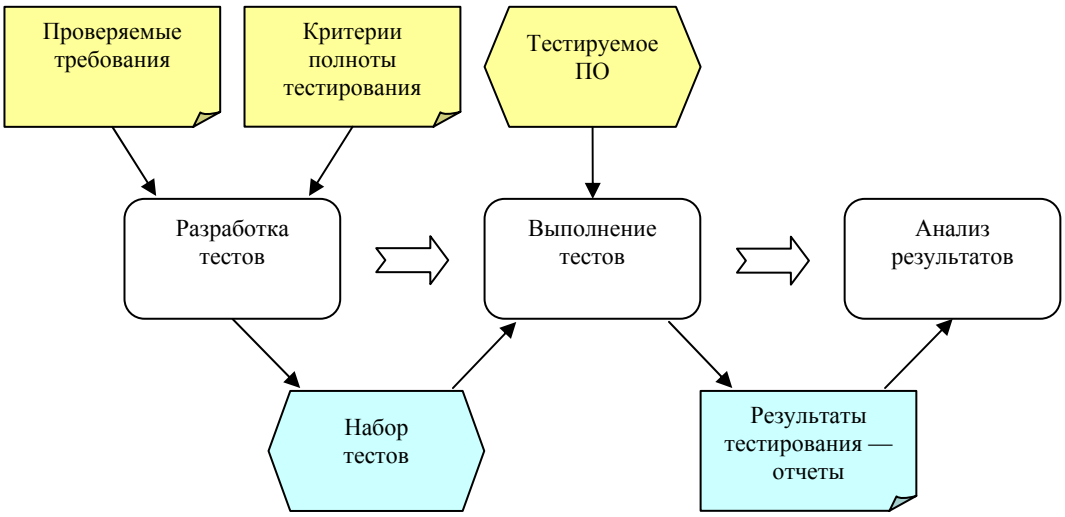


Рисунок 25. Схема процесса тестирования.

Тестирование — наиболее широко применяемый метод контроля качества. Для оценки многих атрибутов качества не существует других эффективных способов, кроме тестирования.

Организация тестирования ПО регулируется следующими стандартами.

- IEEE 829-1998 Standard for Software Test Documentation.
  Описывает виды документов, служащих для подготовки тестов.
- IEEE 1008-1987 (R1993, R2002) Standard for Software Unit Testing.
  Описывает организацию модульного тестирования (см. ниже).
- ISO/IEC 12119:1994 (аналог AS/NZS 4366:1996 и ГОСТ Р-2000, также принят IEEE под номером IEEE 1465-1998) Information Technology. Software packages — Quality requirements and testing.
  Описывает требования к процедурам тестирования программных систем.

Тестировать можно соблюдение любых требований, соответствие которым выявляется во время работы ПО. Из характеристик качества по ISO 9126 этим свойством не обладают только атрибуты удобства сопровождения. Поэтому выделяют виды тестирования, связанные с проверкой определенных характеристик и атрибутов качества — тестирование функциональности, надежности, удобства использования, переносимости и производительности, а также тестирование защищенности, функциональной пригодности и пр. Кроме того, особо выделяют ***нагрузочное*** или ***стрессовое тестирование***, проверяющее работоспособность ПО и показатели его производительности в условиях повышенных нагрузок — при большом количестве пользователей, интенсивном обмене данными с другими системами, большом объеме передаваемых или используемых данных, и пр.

На основе исходных данных, используемых для построения тестов, тестирование делят на следующие виды.

- ***Тестирование черного ящика***, нацеленное на проверку требований. Тесты для него и критерий полноты тестирования строятся на основе требований и ограничений, четко зафиксированных в спецификациях, стандартах, внутренних нормативных документах. Часто такое тестирование называется ***тестированием на соответствие (conformance***

testing). Частным случаем его является ***функциональное тестирование*** — тесты для него, а также используемые критерии полноты проведенного тестирования определяют на основе требований к функциональности.
Еще одним примером тестирования на соответствие является ***аттестационное*** или ***квалификационное тестирование***, по результатам которого программная система получает (или не получает) официальный документ, подтверждающий ее соответствие определенным требованиям и стандартам.

- ***Тестирование белого ящика***, оно же ***структурное тестирование*** — тесты создаются на основе знаний о структуре самой системы и о том, как она работает. Критерии полноты основаны на проценте элементов кода, которые отработали в ходе выполнения тестов. Для оценки степени соответствия требованиям могут привлекаться дополнительные знания о связи требований с определенными ограничениями на значения внутренних данных системы (например, на значения параметров вызовов, результатов и локальных переменных).
- Тестирование, нацеленное на определенные ошибки. Тесты для такого тестирования строятся так, чтобы гарантированно выявлять определенные виды ошибок. Полнота тестирования определяется на основе количества проверенных ситуаций по отношению к общему числу ситуаций, которые мы пытались проверить. К этому виду относится, например, ***тестирование на отказ (smoke testing)***, в ходе которого просто пытаются вывести систему из строя, давая ей на вход как обычные данные, так и некорректные, с нарочно внесенными ошибками.
Другим примером служит метод оценки полноты тестирования при помощи набора ***мутантов*** — программ, совпадающих с тестируемой всюду, кроме нескольких мест, где специально внесены некоторые ошибки. Чем больше мутантов не проходит успешно через данный набор тестов, тем полнее и качественнее проводимое с его помощью тестирование.

Еще одна классификация видов тестирования основана на том уровне, на который оно нацелено. Эти же разновидности тестирования можно связать с фазой жизненного цикла, на которой они выполняются.

- ***Модульное тестирование (unit testing)*** предназначено для проверки правильности отдельных модулей, вне зависимости от их окружения. При этом проверяется, что если модуль получает на вход данные, удовлетворяющие определенным критериям корректности, то и результаты его корректны. Для описания критериев корректности входных и выходных данных часто используют ***программные контракты — предусловия***, описывающие для каждой операции, на каких входных данных она предназначена работать, ***постусловия***, описывающие для каждой операции, как должны соотноситься входные данные с возвращаемыми ею результатами, и ***инварианты***, определяющие критерии целостности внутренних данных модуля.
Модульное тестирование является важной составной частью ***отладочного тестирования***, выполняемого разработчиками для отладки написанного ими кода.
- ***Интеграционное тестирование (integration testing)*** предназначено для проверки правильности взаимодействия модулей некоторого набора друг с другом. При этом проверяется, что в ходе совместной работы модули обмениваются данными и вызовами операций, не нарушая взаимных ограничений на такое взаимодействие, например, предусловий вызываемых операций. Интеграционное тестирование также используется при отладке, но на более позднем этапе разработки.
- ***Системное тестирование (system testing)*** предназначено для проверки правильности работы системы в целом, ее способности правильно решать поставленные пользователями задачи в различных ситуациях.
Системное тестирование выполняется через внешние интерфейсы ПО и тесно связано с ***тестированием пользовательского интерфейса*** (или через пользовательский интерфейс), проводимым при помощи имитации действий пользователей над элементами этого интерфейса. Частными случаями этого вида тестирования являются ***тестирование***

графического пользовательского интерфейса (Graphical User Interface, GUI) и ***пользовательского интерфейса Web-приложений*** (WebUI).
Если интеграционное и модульное тестирование чаще всего проводят, воздействуя на компоненты системы при помощи операций предоставляемого ими программного интерфейса (Application Programming Interface, API), то на системном уровне без использования пользовательского интерфейса не обойтись, хотя тестирование через API в этом случае также вполне возможно.

Основной недостаток тестирования состоит в том, что проводить его можно, только когда проверяемый элемент программы уже разработан. Снизить влияние этого ограничения можно, подготавливая тесты (а это — наиболее трудоемкая часть тестирования) на основе требований заранее, когда исходного кода еще нет. Подход опережающей разработки тестов с успехом используется, например, в рамках XP.

Проверка на моделях

Проверка свойств на моделях (model checking) [10] — проверка соответствия ПО требованиям при помощи формализации проверяемых свойств, построения формальных моделей проверяемого ПО (чаще всего в виде автоматов различных видов) и автоматической проверки выполнения этих свойств на построенных моделях. Проверка свойств на моделях позволяет проверять достаточно сложные свойства автоматически, при минимальном участии человека. Однако она оставляет открытым вопрос о том, насколько выявленные свойства модели можно переносить на само ПО.

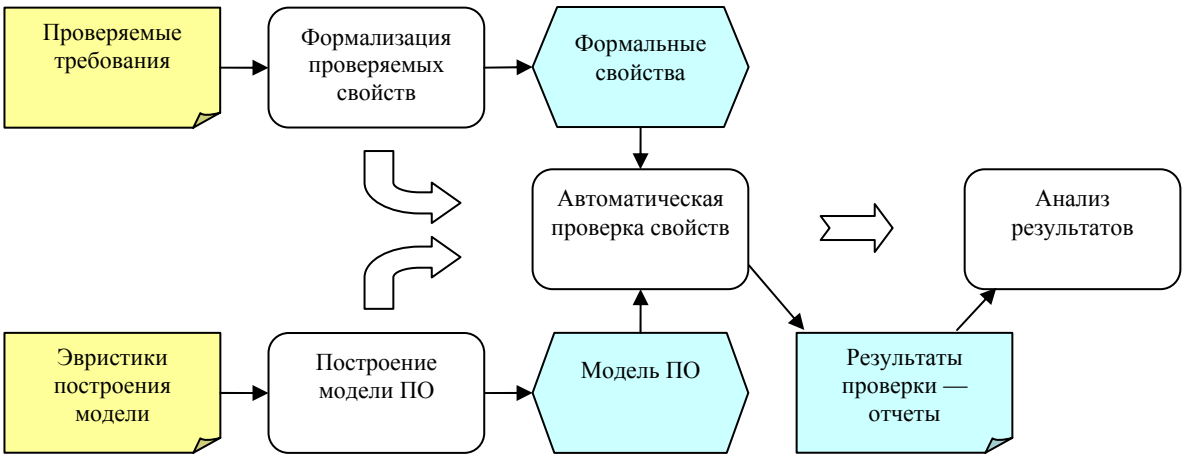


Рисунок 26. Схема процесса проверки свойств ПО на моделях.

Обычно при помощи проверки свойств на моделях анализируют два вида свойств алгоритмов, использованных при построении ПО. ***Свойства безопасности (safety properties)*** утверждают, что нечто нежелательное никогда не случится в ходе работы ПО. ***Свойства живучести (liveness properties)*** утверждают, наоборот, что нечто желательное при любом развитии событий произойдет в ходе его работы.

Примером свойства первого типа служит отсутствие *взаимных блокировок (deadlocks)*. Взаимная блокировка возникает, если каждый из группы параллельно работающих в проверяемом ПО процессов или потоков ожидает прибытия данных или снятия блокировки ресурса от одного из других, а тот не может продолжить выполнение, ожидая того же от первого или от третьего процесса, и т.д.

Примером свойства живучести служит гарантированная доставка сообщения, обеспечиваемая некоторыми протоколами — как бы ни развивались события, если сетевое соединение между машинами будет работать, посланное с одной стороны (процессом на первой машине) сообщение будет доставлено другой стороне (процессу на второй машине).

В классическом подходе к проверке на моделях проверяемые свойства формализуются в виде формул так называемых *временны**х** логик*. Их общей чертой является наличие операторов «всегда

в будущем» и «когда-то в будущем». Заметим, что второй оператор может быть выражен с помощью первого и отрицания — то, что некоторое свойство когда-то будет выполнено, эквивалентно тому, что отрицание этого свойства не будет выполнено всегда. Свойства безопасности легко записываются в виде «всегда будет выполнено отрицание нежелательного свойства», а свойства живости — в виде «когда-то обязательно будет выполнено желаемое».

Проверяемая программа в классическом подходе моделируется при помощи конечного автомата. Проверка, выполняемая автоматически, состоит в том, что для всех достижимых при работе системы состояний этого автомата проверяется нужное свойство. Если оно оказывается выполненным, выдается сообщение об успешности проверки, если нет — выдается трасса, последовательность выполнения отдельных шагов программы, моделируемых переходами автомата, приводящая из начального состояния в такое, в котором нужное свойство нарушается. Эта трасса используется для анализа происходящего и исправления либо программы, либо модели, если ошибка находится в ней.

Основная проблема этого подхода — огромное, а часто и бесконечное, количество состояний в моделях, достаточно хорошо отражающих поведение реальных программ. Для борьбы с комбинаторным взрывом состояний применяются различные методы оптимизации представления автомата, выделения и поиска состояний, существенных для выполнения проверяемого свойства.

Ошибки в программах

Ошибками в ПО, вообще говоря, являются все возможные несоответствия между демонстрируемыми характеристиками его качества и сформулированными или подразумеваемыми требованиями и ожиданиями пользователей.

В англоязычной литературе используется несколько терминов, часто одинаково переводящихся как «ошибка» на русский язык.

- ***defect*** — самое общее нарушение каких-либо требований или ожиданий, не обязательно проявляющееся вовне (к дефектам относятся нарушения стандартов кодирования, недостаточная гибкость системы и пр.).
- ***failure*** — наблюдаемое нарушение требований, проявляющееся при каком-то реальном сценарии работы ПО. Это можно назвать проявлением ошибки.
- ***fault*** — ошибка в коде программы, вызывающая нарушения требований при работе (failures), то место, которое надо исправить. Хотя это понятие используется довольно часто, оно, вообще говоря, не вполне четкое, поскольку для устранения нарушения можно исправить программу в нескольких местах. Что именно надо исправлять, зависит от дополнительных условий, выполнение которых мы хотим при этом обеспечить, хотя в некоторых ситуациях наложение дополнительных ограничений не устраняет неоднозначность.
- ***error*** — используется в двух смыслах.
  Первый — это ошибка в ментальной модели программиста, ошибка в его рассуждениях о программе, которая заставляет его делать ошибки в коде (faults). Это, собственно, ошибка, которую сделал человек в своем понимании свойств программы.
  Второй смысл — это некорректные значения данных (выходных или внутренних), которые возникают при ошибках в работе программы.

Эти понятия некоторым образом связаны с основными источниками ошибок. Поскольку при разработке программ необходимо сначала понять задачу, затем придумать ее решение и закодировать его в виде программы, то, соответственно, основных источников ошибок три.

- Неправильное понимание задач.
  Очень часто люди не понимают, что им пытаются сказать другие. Так же и разработчики ПО не всегда понимают, что именно нужно сделать. Другим источником непонимания служит отсутствие его у самих пользователей и заказчиков — достаточно часто они могут просить сделать несколько не то, что им действительно нужно.

Для предотвращения неправильного понимания задач программной системы служит анализ предметной области.

- Неправильное решение задач.
  Зачастую, даже правильно поняв, что именно нужно сделать, разработчики выбирают неправильный подход к тому, как это делать. Выбираемые решения могут обеспечивать лишь некоторые из требуемых свойств, они могут хорошо подходить для данной задачи в теории, но плохо работать на практике, в конкретных обстоятельствах, в которых должно будет работать ПО.
  Помочь в выборе правильного решения может сопоставление альтернативных решений и тщательный анализ их на предмет соответствия всем требованиям, поддержание постоянной связи с пользователями и заказчиками, предоставление им необходимой информации о выбранных решениях, демонстрация прототипов, анализ пригодности выбираемых решений для работы в том контексте, в котором они будут использоваться.
- Неправильный перенос решений в код.
  Имея правильное решение правильно понятой задачи, люди, тем не менее, способны сделать достаточно много ошибок при воплощении этих решений. Корректному представлению решений в коде могут помешать как обычные опечатки, так и забывчивость программиста или его нежелание отказаться от привычных приемов, которые не дают возможности аккуратно записать принятое решение.
  С ошибками такого рода можно справиться при помощи инспектирования кода, взаимного контроля, при котором разработчики внимательно читают код друг друга, опережающей разработки модульных тестов и тестирования.

Первое место в неформальном состязании за место «самой дорого обошедшейся ошибки в ПО» (см. [11,12]) долгое время удерживала ошибка, приведшая к неудаче первого запуска ракеты Ариан-5 4 июня 1996 года (см. [13]), стоившая около $500 000 000. После произошедшего 14 августа 2003 года обширного отключения электричества на северо-востоке Северной Америки, стоившего экономике США и Канады от 4 до 10 миллиардов долларов [14], это место можно отдать спровоцировавшей его ошибке в системе управления электростанцией. Широко известны также примеры ошибок в системах управления космическими аппаратами, приведшие к их потере или разрушению. Менее известны, но не менее трагичны, ошибки в ПО, управлявшем медицинским и военным оборудованием, некоторые из которых привели к гибели людей.

Стоит отметить, что в большинстве примеров ошибок, имевших тяжелые последствия, нельзя однозначно приписать всю вину за случившееся ровно одному недочету, одному месту в коде. Ошибки очень часто «охотятся стаями». К тяжелым последствиям чаще всего приводят ошибки системного характера, затрагивающие многие аспекты и элементы системы в целом. Это значит, что при анализе такого происшествия обычно выявляется множество частных ошибок, нарушений действующих правил, недочетов в инструкциях и требованиях, которые совместно привели к создавшейся ситуации.

Даже если ограничиться рассмотрением только ПО, часто одно проявление ошибки (failure) может выявить несколько дефектов, находящихся в разных местах. Такие ошибки возникают, как показывает практика, в тех ситуациях, поведение в рамках которых неоднозначно или недостаточно четко определяется требованиями (а иногда и вообще никак не определяется — признак неполного понимания задачи). Поэтому разработчики различных модулей ПО имеют возможность по-разному интерпретировать те части требований, которые относятся непосредственно к их модулям, а также иметь разные мнения по поводу области ответственности каждого из взаимодействующих модулей в данной ситуации. Если различия в их понимании не выявляются достаточно рано, при разработке системы, то становятся «минами замедленного действия» в ее коде.

Например, анализ катастрофы Ариан-5 показал следующее [13].

- Ариан-5 была способна летать при более высоких значениях ускорений и скоростей, чем это могла делать ракета предыдущей серии, Ариан-4.
  Однако большое количество процедур контроля и управления движением по траектории в

коде управляющей системы было унаследовано от Ариан-4. Большинство таких процедур не были специально проверены на работоспособность в новой ситуации, как в силу большого размера кода, который надо было проанализировать, так и потому, что этот код раньше не вызывал проблем, а соотнести его со специфическими характеристиками полета ракет вовремя никто не сумел.

- В одной из таких процедур производилась обработка горизонтальной скорости ракеты. При выходе этой величины за границы, допустимые для Ариан-4, создавалась исключительная ситуация переполнения.
  Надо отметить, что обработка нескольких достаточно однородных величин производилась по-разному — семь переменных могли вызвать исключительную ситуацию данного вида, обработка четырех из них была защищена от этого, а три оставшихся, включая горизонтальную скорость, оставлены без защиты. Аргументом для этого послужило выдвинутое при разработке требование поддерживать загрузку процессора не выше 80%. «Нагружающие» процессор защитные действия для этих переменных не были использованы, поскольку предполагалось, что эти величины будут находиться в нужных пределах в силу физических ограничений на параметры движения ракеты. Обоснований для поддержки именно такой загрузки процессора и того, что отсутствие обработки переполнения выбранных величин будет способствовать этому, найдено не было.
- Когда такая ситуация действительно случилась, т.е. горизонтальная скорость ракеты превысила определенное значение, она не была обработана соответствующим образом, и в результате ею вынужден был заняться модуль, обеспечивающим отказоустойчивость программной системы в целом.
- Этот модуль, в силу отсутствия у него какой-либо возможности обрабатывать такие ошибки специальным образом, применил обычный прием — остановил процесс, в котором возникла ошибка, и запустил другой процесс с теми же исходными данными. Как легко догадаться, эта же ошибка повторилась и во втором процессе.
- Не в силах получить какие-либо осмысленные данные о текущем состоянии полета, система управления использовала ранее полученные, которые уже не соответствовали действительности. При этом были ошибочно включены боковые двигатели «для корректировки траектории», ракета начала болтаться, угол между нею и траекторией движения стал увеличиваться и достиг 20 градусов. В результате она стала испытывать чрезмерные аэродинамические нагрузки и была автоматически уничтожена.

Литература к Лекции 5

[1] ISO/IEC 9126-1:2001. Software engineering — Software product quality — Part 1: Quality model.
[2] ISO/IEC TR 9126-2:2003 Software engineering — Product quality — Part 2: External metrics.
[3] ISO/IEC TR 9126-3:2003 Software engineering — Product quality — Part 3: Internal metrics.
[4] ISO/IEC TR 9126-4:2004 Software engineering — Product quality — Part 4: Quality in use metrics.
[5] ISO 9000:2000 Quality management systems — Fundamentals and vocabulary.
[6] ISO 9001:2000 Quality management systems — Requirements.
[7] ISO 9004:2000 Quality management systems — Guidelines for performance improvements.
[8] ISO/IEC 90003:2004 Software engineering — Guidelines for the application of ISO 9001:2000 to computer software.
[9] В. В. Липаев. Методы обеспечения качества крупномасштабных программных средств. М.: Синтег, 2003.
[10] Э. М. Кларк, О. Грамберг, Д. Пелед. Верификация моделей программ: Model Checking. М.: МЦНМО, 2002.
[11] http://www5.in.tum.de/~huckle/bugse.html
[12] http://infotech.fanshawec.on.ca/gsantor/Computing/FamousBugs.htm
[13] http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html

[14] http://www.elcon.org/Documents/EconomicImpactsOfAugust2003Blackout.pdf
[15] И. Соммервилл. Инженерия программного обеспечения. М.: Вильямс, 2002.
[16] Э. Дж. Брауде. Технология разработки программного обеспечения. СПб.: Питер, 2004.
[17] Б. Боэм, Дж. Браун, Х. Каспар и др. Характеристики качества программного обеспечения. М.: Мир, 1991.

Лекция 6. Архитектура программного обеспечения

Аннотация

Рассматривается понятие архитектуры ПО, влияние архитектуры на свойства ПО, а также методы оценки архитектуры. Рассказывается об основных элементах унифицированного языка моделирования UML.

Ключевые слова

Архитектура ПО, компонент архитектуры, представление архитектуры, сценарий использования, методы оценки архитектуры, диаграммы классов, диаграммы взаимодействия, диаграммы сценариев, диаграммы компонентов, диаграммы развертывания.

Текст лекции

Анализ области решений

Допустим, мы разобрались в предметной области, поняли, что требуется от будущей программной системы, даже зафиксировали настолько полно, насколько смогли, требования и пожелания всех заинтересованных лиц ко всем аспектам качества программы. Что делать дальше?

На этом этапе (а точнее, гораздо раньше, обычно еще в ходе анализа предметной области) исследуются возможные способы решения тех задач, которые поставлены в требованиях. Не всегда бывает нужно специальное изучение этого вопроса — чаще всего имеющийся опыт разработчиков сразу подсказывает, как можно решать поставленные задачи. Однако иногда, все-таки, возникает потребность сначала понять, как это можно сделать и, вообще, возможно ли это сделать и при каких ограничениях.

Таким образом, явно или неявно, проводится ***анализ области решений***. Целью этой деятельности является понимание, можно ли вообще решить стоящие перед разрабатываемой системой задачи, при каких условиях и ограничениях это можно сделать, как они решаются, если решение есть, а если нет — нельзя ли придумать способ его найти или получить хотя бы приблизительное решение, и т.п. Обычно задача хорошо исследована в рамках какой-либо области человеческих знаний, но иногда приходится потратить некоторые усилия на выработку собственных решений. Кроме того, решений обычно несколько и они различаются по некоторым характеристикам, способным впоследствии сыграть важную роль в процессе развития и эксплуатации созданной на их основе программы. Поэтому важно взвесить их плюсы и минусы и определить, какие из них наиболее подходят в рамках данного проекта, или решить, что все они должны использоваться для обеспечения большей гибкости ПО.

Когда определены принципиальные способы решения всех поставленных задач (быть может, в каких-то ограничениях), основной проблемой становится способ организации программной системы, который позволил бы реализовать все эти решения и при этом удовлетворить требованиям, касающимся нефункциональных аспектов разрабатываемой программы. Искомый способ организации ПО в виде системы взаимодействующих компонентов называют *архитектурой*, а процесс ее создания — *проектированием архитектуры ПО*.

Архитектура программного обеспечения

Под ***архитектурой ПО*** понимают набор внутренних структур ПО, которые видны с различных точек зрения и состоят из компонентов, их связей и возможных взаимодействий между компонентами, а также доступных извне свойств этих компонентов [1].

Под ***компонентом*** в этом определении имеется в виду достаточно произвольный структурный элемент ПО, который можно выделить, определив интерфейс взаимодействия между этим компонентом и всем, что его окружает. Обычно при разработке ПО термин «компонент» (см. далее при обсуждении компонентных технологий) имеет несколько другой, более узкий смысл — это единица развертывания, самая маленькая часть системы, которую можно включить или не включить в ее состав. Такой компонент также имеет определенный интерфейс и удовлетворяет

некоторому набору правил, называемому компонентной моделью. Там, где возможны недоразумения, будет указано, в каком смысле употребляется этот термин. В этой лекции до обсуждения UML мы будем использовать преимущественно широкое понимание этого термина, а в дальше — наоборот, узкое.

В определении архитектуры упоминается набор структур, а не одна структура. Это означает, что в качестве различных аспектов архитектуры, различных взглядов на нее выделяются различные структуры, соответствующие разным аспектам взаимодействия компонентов. Примеры таких аспектов — описание типов компонентов и типов статических связей между ними при помощи диаграмм классов, описание композиции компонентов при помощи структур ссылающихся друг на друга объектов, описание поведения компонентов при помощи моделирования их как набора взаимодействующих, передающих друг другу некоторые события, конечных автоматов.

Архитектура программной системы похожа на набор карт некоторой территории. Карты имеют разные масштабы, на них показаны разные элементы (административно-политическое деление, рельеф и тип местности — лес, степь, пустыня, болота и пр., экономическая деятельность и связи), но они объединяются тем, что все представленные на них сведения соотносятся с географическим положением. Точно так же архитектура ПО представляет собой набор структур или представлений, имеющих различные уровни абстракции и показывающих разные аспекты (структуру классов ПО, структуру развертывания, т.е. привязки компонентов ПО к физическим машинам, возможные сценарии взаимодействий компонентов и пр.), объединяемых сопоставлением всех представленных данных со структурными элементами ПО. При этом уровень абстракции данного представления является аналогом масштаба географической карты.

Рассмотрим в качестве примера программное обеспечение авиасимулятора для командной тренировки пилотов. Задачей такой системы в целом является контроль и выработка необходимых для безопасного управления самолетом навыков у команд летчиков. Кроме того, отрабатываются навыки поведения в особых ситуациях, связанных с авариями, частичной потерей управления самолетом, тяжелыми условиями полета, и т.д.

Симулятор должен:

- Моделировать определенные условия полета и создавать некоторые события, к которым относятся следующие.
  - Скоростной и высотный режим полета, запас горючего, их изменения со временем.
  - Модель самолета и ее характеристики по управляемости, возможным режимам полета и скорости реакции на различные команды.
  - Погода за бортом и ее изменения со временем.
  - Рельеф и другие особенности местности в текущий момент, их изменения со временем.
  - Исходный и конечный пункты полета, расстояние и основные характеристики рельефа между ними.
  - Исправность или неисправность элементов системы контроля полета и управления самолетом, показатели системы мониторинга и их изменение со временем.
  - Наличие пролетающих вблизи курса самолета других самолетов, их геометрические и скоростные характеристики.
  - Чрезвычайные ситуации, например, террористы на борту, нарушение герметичности корпуса, внезапные заболевания и «смерть» отдельных членов экипажа.

  При этом вся совокупность условий должна быть непротиворечивой, выглядеть и развиваться подобно реальным событиям. Некоторые условия, например, погода, должны изменяться достаточно медленно, другие события — происходить внезапно и приводить к связанным с ними последствиям (нарушение герметичности корпуса может сопровождаться поломками каких-то элементов системы мониторинга или «смертью» одного из пилотов). Если приборы показывают наличие грозы по курсу и они исправны, через некоторое время летчики должны увидеть грозу за бортом и она может начать оказывать влияние на условия полета.

- Принимать команды, подаваемые пилотами, и корректировать демонстрируемые характеристики полета и работы системы управления самолетом в зависимости от этих команд, симулируемой модели самолета и исправности системы управления. Например, при повороте на некоторый угол вправо, показываемый пилотам «вид из кабины» должен переместиться на соответствующий угол влево со скоростью, соответствующей скорости реакции симулируемой модели самолета и исправности задействованных элементов системы управления.

Понятно, что одним из элементов симулятора служит система визуализации обстановки за бортом — она показывает пилотам «вид за окнами». Пилоты в ходе полета ориентируются по показателям огромного количества датчиков, представленных на приборной панели самолета. Вся их работа тоже должна симулироваться. Наличие и характеристики работы таких датчиков могут зависеть от симулируемой модели, но их расположение, форма и цвет служат слишком важными элементами выработки навыков управления самолетом, поэтому требуется поддерживать эти характеристики близкими к реальным. Представлять их только в виде изображений на некоторой панели неудобно, поскольку они должны располагаться и выглядеть максимально похоже на реальные прототипы. Значит, симулировать можно только небольшое семейство самолетов с практически одним и тем же наборов приборов на приборной панели.

Кроме того, все команды пилотов должны восприниматься соответствующими компонентами симулятора и встраиваться в моделируемые условия. В симулятор должен быть включен генератор событий, зависящий от текущей ситуации, а также интерфейс мониторинга и управления, с помощью которого внешний оператор мог бы создавать определенные события во время симуляции полета наблюдать за всем, что происходит. Все события и действия пилотов должны протоколироваться с помощью камер и микрофонов для дальнейшего разбора полетов.

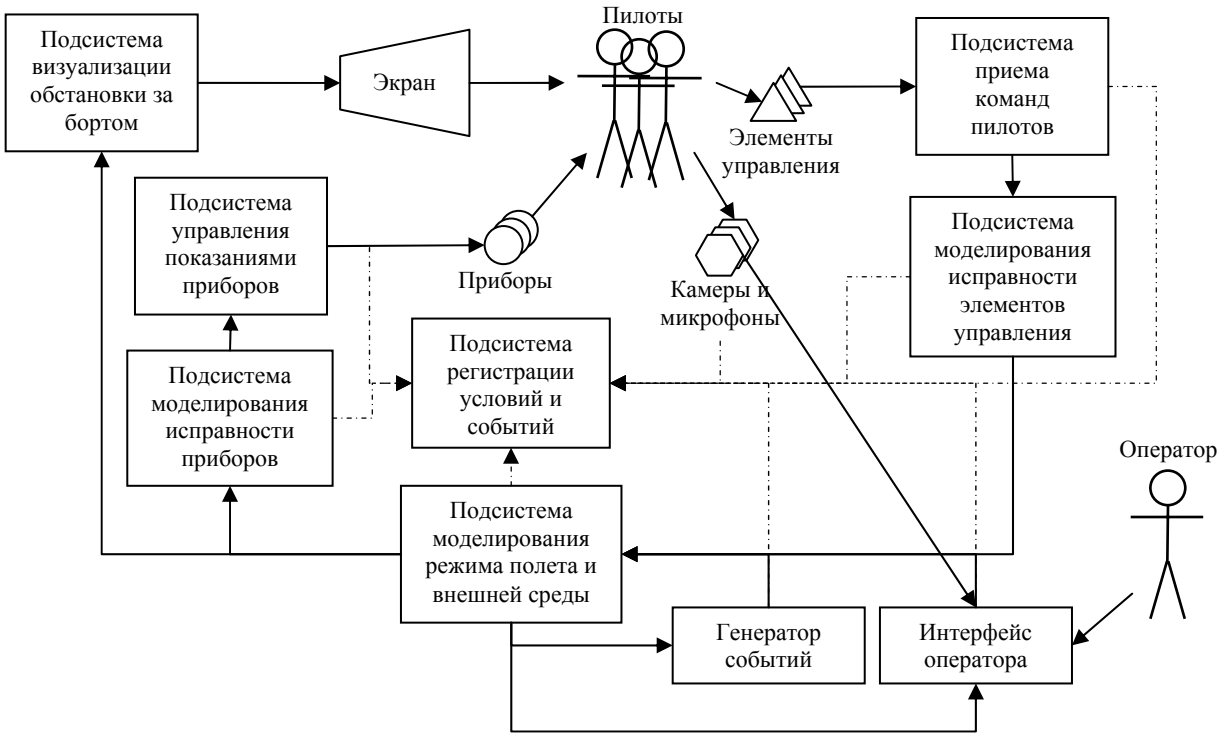


Рисунок 27. Примерная архитектура авиасимулятора.

Рис. 27 показывает набросок архитектуры такого авиасимулятора. Каждый из указанных компонентов решает свои задачи, которые необходимы для работы всей системы. В совокупности они решают все задачи системы в целом. Стрелками показаны потоки данных и управления между компонентами. Пунктирные стрелки изображают потоки данных, передаваемых для протоколирования.

Архитектура определяет большинство характеристик качества ПО в целом. Архитектура служит также основным средством общения между разработчиками, а также между разработчиками и всеми остальными лицами, заинтересованными в данном ПО.

Выбор архитектуры задает способ реализации требований на высоком уровне абстракции. Именно архитектура почти полностью определяет такие характеристики ПО как надежность, переносимость и удобство сопровождения. Она также значительно влияет на удобство использования и эффективность ПО, которые, однако, сильно зависят и от реализации отдельных компонентов. Значительно меньше влияние архитектуры на функциональность — обычно заданную функциональность можно реализовать, использовав совершенно различные архитектуры.

Поэтому выбор между той или иной архитектурой определяется в большей степени именно нефункциональными требованиями и необходимыми свойствами ПО с точки зрения удобства сопровождения и переносимости. При этом для построения хорошей архитектуры надо учитывать возможные противоречия между требованиями к различным характеристикам и уметь выбирать компромиссные решения, дающие приемлемые значения по всем показателям.

Так, для повышения эффективности в общем случае выгоднее использовать монолитные архитектуры, в которых выделено небольшое число компонентов (в пределе — единственный компонент). Этим обеспечивается экономия как памяти, поскольку каждый компонент обычно имеет свои данные, а здесь число компонентов минимально, так и времени работы, поскольку возможность оптимизировать работу алгоритмов обработки данных имеется также только в рамках одного компонента.

С другой стороны, для повышения удобства сопровождения, наоборот, лучше разбить систему на большое число отдельных маленьких компонентов, с тем чтобы каждый из них решал свою небольшую, но четко определенную часть общей задачи. При этом, если возникают изменения в требованиях или проекте, их обычно можно свести к изменению в постановке одной, реже двух или трех таких подзадач и, соответственно, изменять только отвечающие за решение этих подзадач компоненты.

С третьей стороны, для повышения надежности лучше использовать либо небольшой набор простых компонентов, либо дублирование функций, т.е. сделать несколько компонентов ответственными за решение одной подзадачи. Заметим, однако, что ошибки в ПО чаще всего носят неслучайный характер. Они повторяемы, в отличие от аппаратного обеспечения, где ошибки связаны часто со случайными изменениями характеристик среды и могут быть преодолены простым дублированием компонентов без изменения их внутренней реализации. Поэтому при таком обеспечении надежности надо использовать достаточно сильно отличающиеся способы решения одной и той же задачи в разных компонентах.

Другим примером противоречивых требований служат характеристики удобства использования и защищенности. Чем сильнее защищена система, тем больше проверок, процедур идентификации и пр. нужно проходить пользователям. Соответственно, тем менее удобна для них работа с такой системой. При разработке реальных систем приходится искать некоторый разумный компромисс, чтобы сделать систему достаточно защищенной и способной поставить ощутимую преграду для несанкционированного доступа к ее данным и, в то же время, не отпугнуть пользователей сложностью работы с ней.

Список стандартов, регламентирующих описание архитектуры, которое является основной составляющей проектной документации на ПО, выглядит так.

- **IEEE 1016-1998 Recommended Practice for Software Design Descriptions [2]** (рекомендуемые методы описаний проектных решений для ПО).
- **IEEE 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems [3]** (рекомендуемые методы описания архитектуры программных систем).
  Основное содержание этого стандарта сводится к определению набора понятий, связанных с архитектурой программной системы.
  Это, прежде всего, само понятие архитектуры как набора основополагающих принципов организации системы, воплощенных в наборе ее компонентов, связях их друг с другом и между ними и окружением системы, а также принципов проектирования и развития системы.
  Это определение, в отличие от данного в начале этой лекции, делает акцент не на наборе

структур в основе архитектуры, а на принципах ее построения.
Стандарт IEEE 1471 определяет также ***представление архитектуры (architectural description)*** как согласованный набор документов, описывающий архитектуру с точки зрения определенной группы заинтересованных лиц с помощью набора моделей. Архитектура может иметь несколько представлений, отражающих интересы различных групп заинтересованных лиц.
Стандарт рекомендует для каждого представления фиксировать отраженные в нем взгляды и интересы, роли лиц, которые заинтересованы в таком взгляде на систему, причины, обуславливающие необходимость такого рассмотрения системы, несоответствия между элементами одного представления или между различными представлениями, а также различную служебную информацию об источниках информации, датах создания документов и пр.
Стандарт IEEE 1471 отмечает необходимость использования архитектуры системы для решения таких задач, как следующие.

- Анализ альтернативных проектов системы.
- Планирование перепроектирования системы, внесения изменений в ее организацию.
- Общение по поводу системы между различными организациями, вовлеченными в ее разработку, эксплуатацию, сопровождение, приобретающими систему или продающими ее.
- Выработка критериев приемки системы при ее сдаче в эксплуатацию.
- Разработка документации по ее использованию и сопровождению, включая обучающие и маркетинговые материалы.
- Проектирование и разработка отдельных элементов системы.
- Сопровождение, эксплуатация, управление конфигурациями и внесение изменений и поправок.
- Планирование бюджета и использования других ресурсов в проектах, связанных с разработкой, сопровождением или эксплуатацией системы.
- Проведение обзоров, анализ и оценка качества системы.

Разработка и оценка архитектуры на основе сценариев

При проектировании архитектуры системы на основе требований, зафиксированных в виде вариантов использования, первые возможные шаги состоят в следующем.

1. Выделение компонентов
   a. Выбирается набор «основных» сценариев использования — наиболее существенных и выполняемых чаще других.
   b. Исходя из опыта проектировщиков, выбранного архитектурного стиля (см. следующую лекцию) и требований к переносимости и удобству сопровождения системы определяются компоненты, отвечающие за определенные действия в рамках этих сценариев, т.е. за решение определенных подзадач.
   c. Каждый сценарий использования системы представляется в виде последовательности обмена сообщениями между полученными компонентами.
   d. При возникновении дополнительных хорошо выделенных подзадач добавляются новые компоненты, и сценарии уточняются.
2. Определение интерфейсов компонентов
   a. Для каждого компонента в результате выделяется его интерфейс — набор сообщений, которые он принимает от других компонентов и посылает им.
   b. Рассматриваются «неосновные» сценарии, которые так же разбиваются на последовательности обмена сообщениями с использованием, по возможности, уже определенных интерфейсов.
   c. Если интерфейсы недостаточны, они расширяются.

d. Если интерфейс компонента слишком велик, или компонент отвечает за слишком многое, он разбивается на более мелкие.

3. Уточнение набора компонентов
   a. Там, где это необходимо в силу требований эффективности или удобства сопровождения, несколько компонентов могут быть объединены в один.
   b. Там, где это необходимо для удобства сопровождения или надежности, один компонент может быть разделен на несколько.
4. Достижение нужных свойств.
   Все это делается до тех пор, пока не выполнятся следующие условия:
   a. Все сценарии использования реализуются в виде последовательностей обмена сообщениями между компонентами в рамках их интерфейсов.
   b. Набор компонентов достаточен для обеспечения всей нужной функциональности, удобен для сопровождения или портирования на другие платформы и не вызывает заметных проблем производительности.
   c. Каждый компонент имеет небольшой и четко очерченный круг решаемых задач и строго определенный, сбалансированный по размеру интерфейс.

На основе возможных *сценариев использования или модификации* системы возможен также анализ характеристик архитектуры и оценка ее пригодности для поставленных задач или сравнительный анализ нескольких архитектур. Это так называемый *метод анализа архитектуры ПО* (Software Architecture Analysis Method, SAAM) [1,4]. Основные его шаги следующие.

1. Определить набор сценариев действий пользователей или внешних систем, использующих некоторые возможности, которые могут уже планироваться для реализации в системе или быть новыми. Сценарии должны быть значимы для конкретных заинтересованных лиц, будь то пользователь, разработчик, ответственный за сопровождение, представитель контролирующей организации и пр. Чем полнее набор сценариев, тем выше будет качество анализа. Можно также оценить частоту появления и важность сценариев, возможный ущерб от невозможности их выполнить.
2. Определить архитектуру (или несколько сравниваемых архитектур). Это должно быть сделано в форме, понятной всем участникам оценки.
3. Классифицировать сценарии. Для каждого сценария из набора должно быть определено, поддерживается ли он уже данной архитектурой или для его поддержки нужно вносить в нее изменения. Сценарий может поддерживаться, т.е. его выполнение не потребует внесения изменений ни в один из компонентов, или же не поддерживаться, если его выполнение требует изменений в описании поведения одного или нескольких компонентов или изменений в их интерфейсах. Поддержка сценария означает, что лицо, заинтересованное в его выполнении, оценивает степень поддержки как достаточную, а необходимые при этом действия — как достаточно удобные.
4. Оценить сценарии. Определить, какие из сценариев полностью поддерживаются рассматриваемыми архитектурами. Для каждого неподдерживаемого сценария надо определить необходимые изменения в архитектуре — внесение новых компонентов, изменения в существующих, изменения связей и способов взаимодействия. Если есть возможность, стоит оценить трудоемкость внесения таких изменений.
5. Выявить взаимодействие сценариев. Определить какие компоненты требуется изменять для неподдерживаемых сценариев; если требуется изменять один компонент для поддержки нескольких сценариев — такие сценарии называют взаимодействующими. Нужно оценить смысловые связи между взаимодействующими сценариями.
   Малая связанность по смыслу между взаимодействующими сценариями означает, что компоненты, в которых они взаимодействуют, выполняют слабо связанные между собой задачи и их стоит декомпозировать.
   Компоненты, в которых взаимодействуют много (> 2-х) сценариев, также являются возможными проблемными местами.

6. Оценить архитектуру в целом (или сравнить несколько заданных архитектур). Для этого надо использовать оценки важности сценариев и степень их поддержки архитектурой.

Рассмотрим сравнительный анализ двух архитектур на примере индексатора — программы для построения индекса некоторого текста, т.е. упорядоченного по алфавиту списка его слов без повторений.

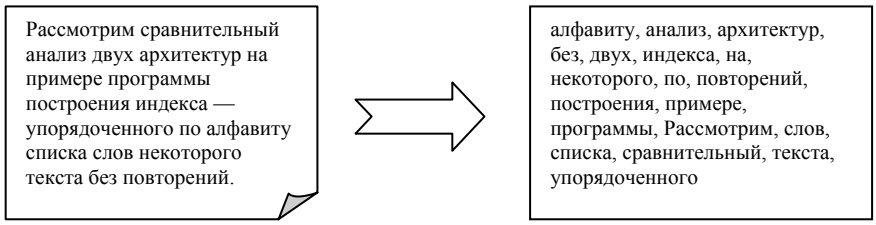


Рисунок 28. Пример работы индексатора текста.

1. Выделим следующие сценарии работы или модификации программы.
   a. Надо сделать так, чтобы индексатор мог работать в инкрементальном режиме, читая на входе одну фразу за другой и пополняя получаемый в процессе работы индекс.
   b. Надо сделать так, чтобы индексатор мог игнорировать предлоги, союзы, местоимения, междометия, частицы и другие служебные слова.
   c. Надо сделать так, чтобы индексатор мог обрабатывать тексты, подаваемые ему на вход в виде архивов.
   d. Надо сделать так, чтобы в индексе оставались только слова в основной грамматической форме — существительные в единственном числе и именительном падеже, глаголы в неопределенной форме и пр.
2. Определим две возможных архитектуры индексатора для сравнительного анализа.
   a. В качестве первой архитектуры рассмотрим разбиение индексатора на два компонента. Один компонент принимает на свой вход входной текст, полностью прочитывает его и выдает на выходе список слов, из которых он состоит. Второй компонент принимает на вход список слов, а на выходе выдает его упорядоченный вариант без повторений. Этот вариант архитектуры построен в стиле «каналы и фильтры» (см. следующую лекцию).

Рисунок 29. Архитектура индексатора в стиле каналов и фильтров.

   b. Другой вариант архитектуры индексатора устроен следующим образом. Имеется внутренняя структура данных, хранящая подготовленный на настоящий момент вариант индекса. Он представляет собой упорядоченный список без повторений всех слов, прочитанных до настоящего момента. Кроме того, имеются две переменные — строка, хранящая последнее (быть может, не до конца) прочитанное слово, и ссылка на то слово в подготовленном списке, которое лексикографически следует за последним словом (соответственно, предшествующее этому слово в списке лексикографически предшествует последнему прочитанному слову).
   В дополнение к этим данным имеются следующие компоненты.
      i. Первый читает очередной символ на входе и передает его на обработку одному из остальных.
      Если это разделитель слов (пробел, табуляция, перевод строки), управление получает второй компонент.
      Если это буква — третий.
      Если входной текст кончается — четвертый.

ii. Второй компонент закачивает ввод последнего слова — оно помещается в список перед тем местом, на которое указывает ссылка; после чего последнее слово становится пустым, а ссылка начинает указывать на первое слово в списке.

iii. Третий компонент добавляет прочитанную букву в конец последнего слова, после чего, быть может, перемещает ссылку на следующее за полученным слово в списке.

iv. Четвертый компонент выдает полученный индекс на выход.

Эта архитектура построена в стиле «репозиторий» (см. следующую лекцию).

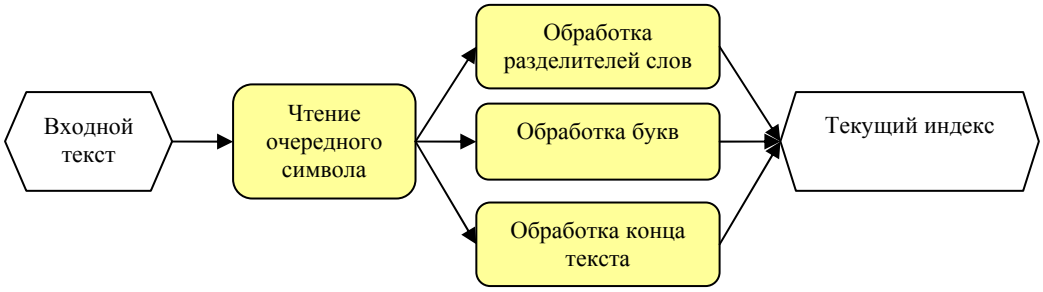


Рисунок 30. Архитектура индексатора в стиле репозитория.

3. Определим поддерживаемые сценарии из выделенного набора.
   a. Сценарий a.
   Этот сценарий прямо поддерживается второй архитектурой.
   Чтобы поддержать его в первой, необходимо внести изменения в оба компонента так, чтобы первый компонент мог бы пополнять промежуточный список, читая входной текст фраза за фразой, а второй — аналогичным способом пополнять результирующий упорядоченный список, вставляя туда поступающие ему на вход слова.
   b. Сценарий b.
   Обе архитектуры не поддерживают этот сценарий.
   Для его поддержки в первой архитектуре необходимо изменить первый компонент или, лучше, вставить после него дополнительный фильтр, отбрасывающий вспомогательные части речи.
   Для поддержки этого сценария второй архитектурой нужно ввести дополнительный компонент, который перехватывает буквы, выдаваемые модулем их обработки (соответственно, этот модуль больше не должен перемещать указатель по итоговому списку) и сигналы о конце слова от первого компонента, после чего он должен отсеивать служебные слова.
   c. Сценарий c.
   Этот сценарий также требует изменений в обеих архитектурах.
   Однако в обоих случаях эти изменения одинаковы — достаточно добавить дополнительный компонент, декодирующий архивы, если они подаются на вход.
   d. Сценарий d.
   Этот сценарий также не поддерживается обеими архитектурами.
   Требуемые им изменения аналогичны требованиям второго сценария, только в этом случае дополнительный компонент-фильтр должен еще и преобразовывать слова в их основную форму и только после этого пытаться добавить результат к итоговому индексу.
   Таким образом, требуется, как и во втором случае, изменить или добавить один компонент в первой архитектуре и изменить один и добавить новый во второй.
4. Мы уже выполнили оценку сценариев на предыдущем шаге. Итоги этой оценки приведены в Таблице 6.
5. Мы видели, что при использовании первого варианта архитектуры только для поддержки первого сценария пришлось бы вносить изменения в ее компоненты. В остальных случаях

достаточно было добавить новый компонент, что несколько проще.
При использовании второго варианта нам в двух разных сценариях, помимо добавления нового компонента, потребовалось изменить компонент, обрабатывающий буквы.

Архитектура	Сценарий a	Сценарий b	Сценарий c	Сценарий d
Каналы и фильтры	- -	+ + *	+ + *	+ + *
Репозиторий	+ + + +	+ + - + *	+ + + + *	+ + - + *

Таблица 6. Итоги оценки двух вариантов архитектуры индексатора.
+ обозначает возможность не изменять компонент, - — необходимость изменения компонента,
*** — необходимость добавления одного компонента**

6. В целом первая архитектура на предложенных сценариях выглядит лучше второй. Единственный ее недостаток — отсутствие возможности инкрементально поставлять данные на вход компонентам. Если его устранить, сделав компоненты способными потреблять данные постепенно, эта архитектура станет почти идеальным вариантом, поскольку она легко расширяется — для решения многих дополнительных задач потребуется только добавлять компоненты в общий конвейер.
   Вторая архитектура, несмотря на выигрыш в инкрементальности, проигрывает в целом. Основная ее проблема — слишком специфически построенный компонент-обработчик букв. Необходимость изменить его в нескольких сценариях показывает, что нужно объединить обработчик букв и обработчик конца слов в единый компонент, выдающий слова целиком, после чего полученная архитектура не будет ничем уступать исправленной первой.

UML. Виды диаграмм UML

Для представления архитектуры, а точнее — различных входящих в нее структур, удобно использовать графические языки. На настоящий момент наиболее проработанным и наиболее широко используемым из них является *унифицированный язык моделирования* (Unified Modeling Language, UML) [5-7], хотя достаточно часто архитектуру системы описывают просто набором именованных прямоугольников, соединенных линиями и стрелками, которые представляют возможные связи.

UML предлагает использовать для описания архитектуры 8 видов диаграмм. 9-й вид UML диаграмм, диаграммы вариантов использования (см. Лекцию 4), не относится к архитектурным представлениям. Кроме того, и другие виды диаграмм можно использовать для описания внутренней структуры компонентов или сценариев действий пользователей и прочих элементов, к архитектуре часто не относящихся. В этом курсе мы не будем разбирать диаграммы UML в деталях, а ограничимся обзором их основных элементов, необходимым для общего понимания смысла того, что изображено на таких диаграммах.

Диаграммы UML делятся на две группы — *статические* и *динамические диаграммы*.

Статические диаграммы

Статические диаграммы представляют либо постоянно присутствующие в системе сущности и связи между ними, либо суммарную информацию о сущностях и связях, либо сущности и связи, существующие в какой-то определенный момент времени. Они не показывают способов поведения этих сущностей. К этому типу относятся *диаграммы классов*, *объектов*, *компонентов* и *диаграммы развертывания*.

- ***Диаграммы классов (class diagrams)*** показывают *классы* или *типы* сущностей системы, характеристики классов (*поля* и *операции*) и возможные связи между ними. Пример диаграммы классов изображен на Рис. 31.

  Классы представляются прямоугольниками, поделенными на три части. В верхней части показывают имя класса, в средней — набор его полей, с именами, типами, модификаторами доступа (**`public`** ‘+’, **`protected`** ‘#’, **`private`** ‘-’) и начальными значениями, в нижней — набор операций класса. Для каждой операции показывается ее модификатор доступа и

сигнатура.
На Рис. 31 изображены классы `Account`, `Person`, `Organization`, `Address`, `CreditAccount` и абстрактный класс `Client`.
Класс `CreditAccount` имеет **`private`** поле `maximumCredit` типа **`double`**, а также **`public`** метод `getCredit()` и **`protected`** метод `setCredit()`.

Интерфейсы, т.е. типы, имеющие только набор операций и не определяющие способов их реализации, часто показываются в виде небольших кружков, хотя могут изображаться и как обычные классы. На Рис. 31 представлен интерфейс `AccountInterface`.

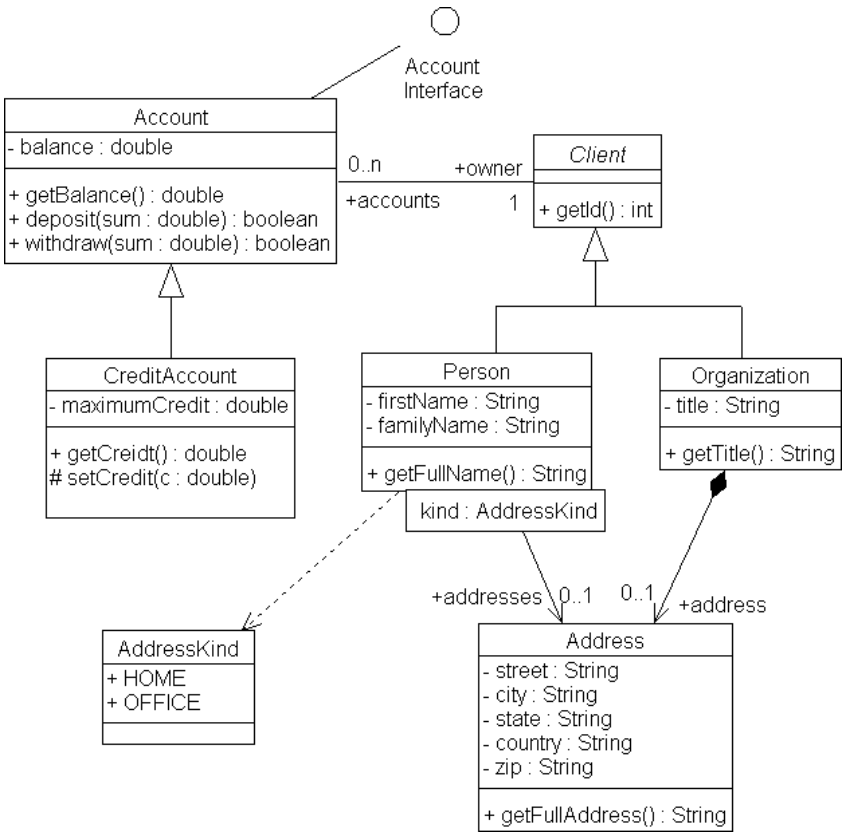


Рисунок 31. Диаграмма классов.

Наиболее часто используется три вида связей между классами — связи по композиции, ссылки, связи по наследованию и реализации.

Композиция описывает ситуацию, в которой объекты класса *A* включают в себя объекты класса *B*, причем последние не могут разделяться (объект класса *B*, являющийся частью объекта класса *A*, не может являться частью другого объекта класса *A*) и существуют только в рамках объемлющих объектов (уничтожаются при уничтожении объемлющего объекта).
Композицией на Рис. 31 является связь между классами `Organization` и `Address`.

Ссылочная связь (или *слабая агрегация*) обозначает, что объект некоторого класса *A* имеет в качестве поля ссылку на объект другого (или того же самого) класса *B*, причем ссылки на один и тот же объект класса *B* могут иметься в нескольких объектах класса *A*.

И композиция, и ссылочная связь изображаются стрелками, ведущими от класса *A* к классу *B*. Композиция дополнительно имеет закрашенный ромбик у начала этой стрелки. Двусторонние ссылочные связи, обозначающие, что объекты могут иметь ссылки друг на друга, показываются линиями без стрелок. Такая связь показана на Рис. 31 между классами `Account` и `Client`.

Эти связи могут иметь описание *множественности*, показывающее, сколько объектов класса *B* может быть связано с одним объектом класса *A*. Оно изображается в виде текстовой метки около конца стрелки, содержащей точное число или нижние и верхние границы, причем бесконечность изображается звездочкой или буквой n. Для двусторонних

связей множественности могут показываться с обеих сторон. На Рис. 31 множественности, изображенные для связи между классами `Account` и `Client`, обозначают, что один клиент может иметь много счетов, а может и не иметь ни одного, и счет всегда привязан ровно к одному клиенту.

Наследование классов изображается стрелкой с пустым наконечником, ведущей от наследника к предку. На Рис. 31 класс `CreditAccount` наследует классу `Account`, а классы `Person` и `Organization` — классу `Client`.

Реализация интерфейсов показывается в виде пунктирной стрелки с пустым наконечником, ведущей от класса к реализуемому им интерфейсу, если тот показан в виде прямоугольника. Если же интерфейс изображен в виде кружка, то связь по реализации показывается обычной сплошной линией (в этом случае неоднозначности в ее толковании не возникает). Такая связь изображена на Рис. 31 между классом `Account` и интерфейсом `AccountInterface`.

Один класс *использует* другой, если этот другой класс является типом параметра или результата операции первого класса. Иногда связи по использованию показываются в виде пунктирных стрелок. Пример такой связи между классом `Person` и перечислимым типом `AddressKind` можно видеть на Рис. 31.

Ссылочные связи, реализованные в виде ассоциативных массивов или отображений (map) — такая связь в зависимости от некоторого набора ключей определяет набор ссылок-значений — показываются при помощи стрелок, имеющих прямоугольник с перечислением типов и имен ключей, примыкающий к изображению класса, от которого идет стрелка. Множественность на конце стрелки при этом обозначает количество ссылок, соответствующее одному набору значений ключей.
На Рис. 31 такая связь ведет от класса `Person` к классу `Address`, показывая, что объект класса `Person` может иметь один адрес для каждого значения ключа `kind`, т.е. один домашний и один рабочий адреса.

Диаграммы классов используются чаще других видов диаграмм.

- ***Диаграммы объектов (object diagrams)*** показывают часть объектов системы и связи между ними в некотором конкретном состоянии или суммарно, за некоторый интервал времени. Объекты изображаются прямоугольниками с идентификаторами ролей объектов (в контексте тех состояний, которые изображены на диаграмме) и типами. Однородные коллекции объектов могут изображаться накладывающимися друг на друга прямоугольниками.
  Такие диаграммы используются довольно редко.

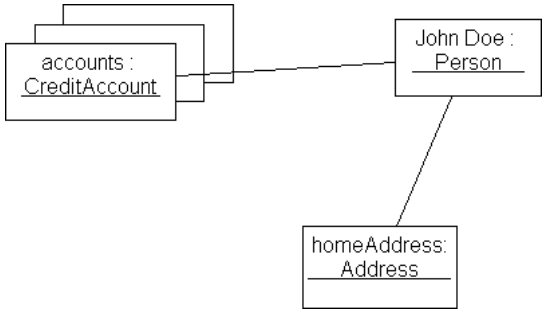


Рисунок 32. Диаграмма объектов.

- ***Диаграммы компонентов (component diagrams)*** представляют компоненты в нескольких смыслах — атомарные составляющие системы с точки зрения ее сборки, конфигурационного управления и развертывания. Компоненты сборки и конфигурационного управления обычно представляют собой файлы с исходным кодом, динамически подгружаемые библиотеки, HTML-странички и пр., компоненты развертывания — это компоненты JavaBeans, CORBA, COM и т.д. Подробнее о таких компонентах см. Лекцию 12.

Компонент изображается в виде прямоугольника с несколькими прямоугольными или другой формы «зубами» на левой стороне.

Связи, показывающие зависимости между компонентами, изображаются пунктирными стрелками. Один компонент зависит от другого, если он не может быть использован в отсутствии этого другого компонента в конфигурации системы. Компоненты могут также реализовывать интерфейсы.
Диаграммы этого вида используются редко.

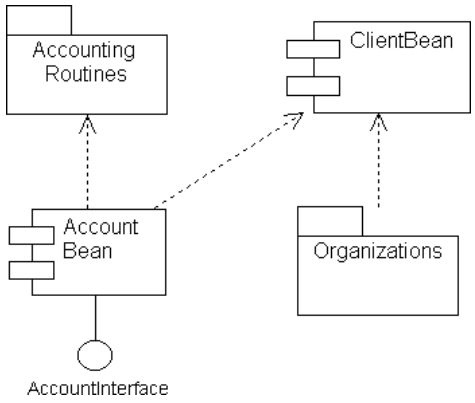


Рисунок 33. Диаграмма компонентов.

На диаграмме компонентов, изображенной на Рис. 33, можно также увидеть *пакеты*, изображаемые в виде «папок», точнее — прямоугольников с прямоугольными «наростами» над левым верхним углом. Пакеты являются пространствами имен и средством группировки диаграмм и других модельных элементов UML — классов, компонентов и пр. Они могут появляться на диаграммах классов и компонентов для указания зависимостей между ними и отдельными классами и компонентами. Иногда на такой диаграмме могут присутствовать только пакеты с зависимостями между ними.

- ***Диаграммы развертывания (deployment diagrams)*** показывают декомпозицию системы на физические устройства различных видов — серверы, рабочие станции, терминалы, принтеры, маршрутизаторы и пр. — и связи между ними, представленные различного рода сетевыми и индивидуальными соединениями.
Физические устройства, называемые ***узлами*** системы (***nodes***), изображаются в виде кубов или параллелепипедов, а физические соединения между ними — в виде линий.
На диаграммах развертывания может быть показана привязка (в некоторый момент времени или постоянная) компонентов развертывания системы к физическим устройствам — например, для указания того, что компонент EJB AccountEJB исполняется на сервере приложений, а аплет AccountInfoEditor — на рабочей станции оператора банка.

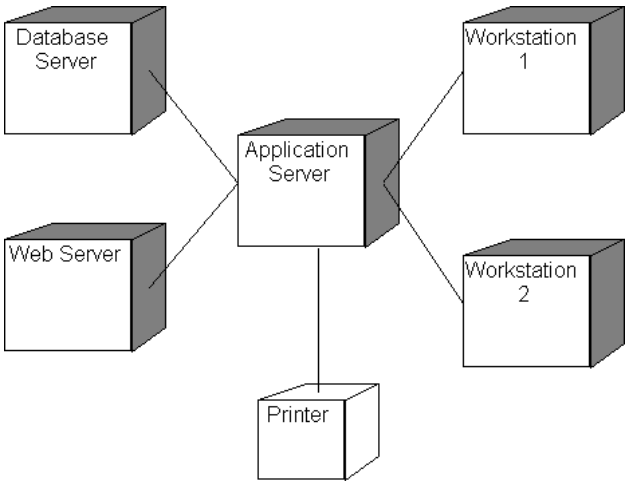


Рисунок 34. Диаграмма развертывания.

Эти диаграммы используются достаточно редко. Пример диаграммы развертывания изображен на Рис. 34.

Динамические диаграммы

Динамические диаграммы описывают происходящие в системе процессы. К ним относятся *диаграммы деятельности*, *сценариев*, *диаграммы взаимодействия* и *диаграммы состояний*.

- ***Диаграммы деятельности (activity diagrams)*** иллюстрируют набор процессов-деятельностей и потоки данных между ними, а также возможные их синхронизации друг с другом.

  Деятельность изображается в виде прямоугольника с закругленными сторонами, слева и справа, помеченного именем деятельности.

  Потоки данных показываются в виде стрелок. Синхронизации двух видов — *развилки (forks)* и *слияния (joins)* — показываются жирными короткими линиями (кто-то может посчитать их и тонкими закрашенными прямоугольниками), к которым сходятся или от которых расходятся потоки данных. Кроме синхронизаций, на диаграммах деятельности могут быть показаны разветвления потоков данных, связанных с выбором того или иного направления в зависимости от некоторого условия. Такие разветвления показываются в виде небольших ромбов.

  Диаграмма может быть поделена на несколько горизонтальных или вертикальных областей, называемых *дорожками (swimlanes)*. Дорожки служат для группировки деятельностей в соответствии с выполняющими их подразделением организации, ролью, приложением, подсистемой и пр.
  Диаграммы деятельности могут заменять часто используемые диаграммы потоков данных (см. Лекцию 4), поэтому применяются достаточно широко. Пример такой диаграммы показан на Рис. 35.

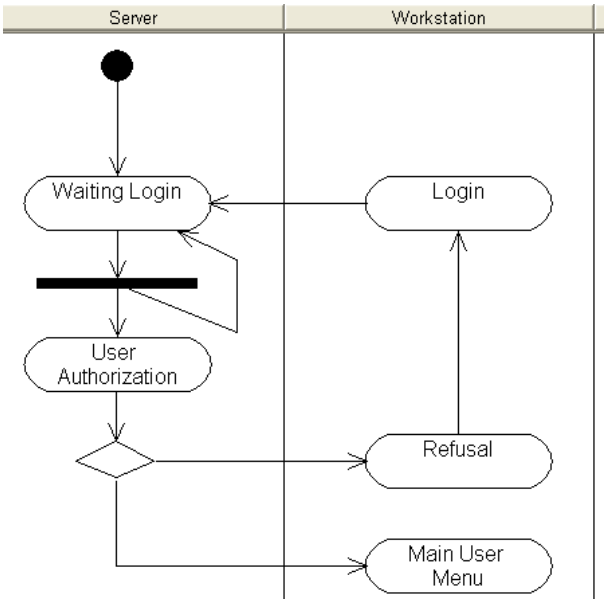


Рисунок 35. Диаграмма деятельности.

- ***Диаграммы сценариев*** (или ***диаграммы последовательности***, ***sequence diagrams***) показывают возможные сценарии обмена сообщениями или вызовами во времени между различными компонентами системы (здесь имеются в виду архитектурные компоненты, компоненты в широком смысле — это могут быть компоненты развертывания, обычные объекты, подсистемы и пр.). Эти диаграммы являются подмножеством специального графического языка — языка *диаграмм последовательностей сообщений (Message Sequence Charts, MSC)*, который был придуман раньше UML и достаточно долго развивается параллельно ему.

Компоненты, участвующие во взаимодействии, изображаются прямоугольниками вверху диаграммы. От каждого компонента вниз идет вертикальная линия, называемая его *линией жизни*. Считается, что ось времени направлена вертикально вниз. Интервалы времени, в которые компонент активен, т.е. управление находится в одной из его операций, представлены тонким прямоугольником, для которого линия жизни компонента является осью симметрии.

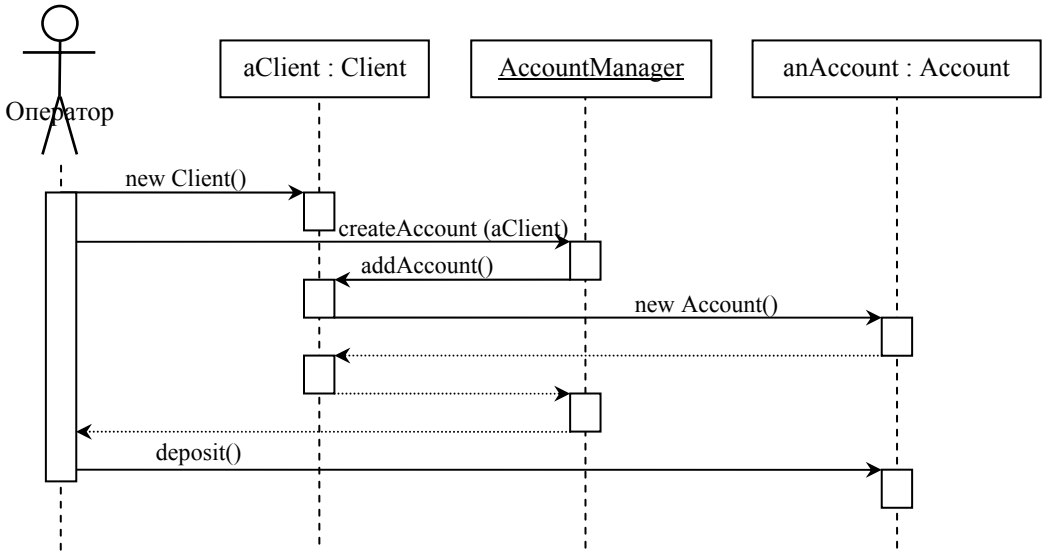


Рисунок 36. Пример диаграммы сценария открытия счета.

Передача сообщения или вызов изображаются стрелкой от компонента-источника к компоненту-приемнику. Возврат управления показан пунктирной стрелкой, обратной к соответствующему вызову.

Эти диаграммы используются достаточно часто, например, при детализации сценариев, входящих в варианты использования. Пример такой диаграммы изображен на Рис. 36.

- ***Диаграммы взаимодействия (collaboration diagrams)*** показывают ту же информацию, что и диаграммы сценариев, но привязывают обмен сообщениями/вызовами не к времени, а к связями между компонентами. Пример такой диаграммы представлен на Рис. 37.

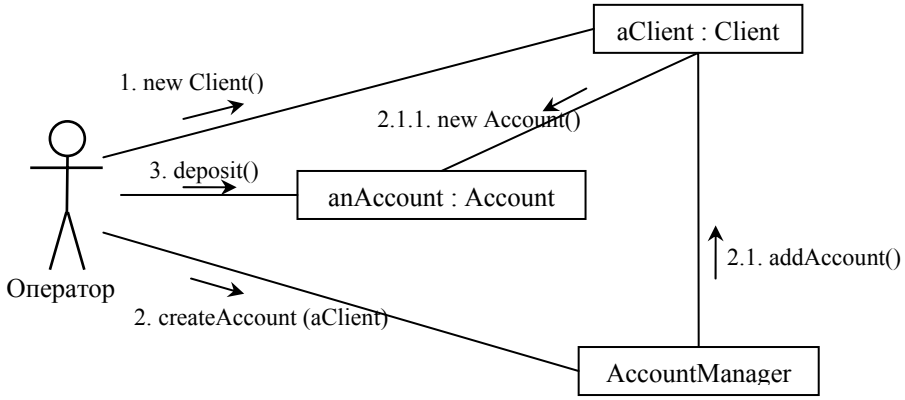


Рисунок 37. Диаграмма взаимодействия, соответствующая диаграмме сценария на Рис. 36.

На диаграмме изображаются компоненты в виде прямоугольников и связи между ними. Вдоль связей могут передаваться сообщения, показываемые в виде небольших стрелок, параллельных связи. Стрелки нумеруются в соответствии с порядком происходящих событий. Нумерация может быть иерархической, чтобы показать вложенность действий друг в друга (т.е. если вызов некоторой операции имеет номер 1, то вызовы, осуществляемые при выполнении этой операции, будут нумероваться как 1.1, 1.2, и т.д.). Диаграммы взаимодействия используются довольно редко.

- ***Диаграммы состояний (statechart diagrams)*** показывают возможные *состояния* отдельных компонентов или системы в целом, *переходы* между ними в ответ на какие-либо *события* и выполняемые при этом *действия*.

  Состояния показываются в виде прямоугольников с закругленными углами, переходы — в виде стрелок. Начальное состояние представляется как небольшой темный кружок, конечное — как пустой кружок с концентрически вложенным темным кружком. Вы могли обратить внимание на темный кружок на диаграмме деятельности на Рис. 35 — он тоже изображает начальное состояние: дело в том, что диаграммы деятельности являются диаграммами состояний специального рода, а деятельности — частный случай состояний. Пример диаграммы состояний приведен на Рис. 38.

  Состояния могут быть устроены иерархически: они могут включать в себя другие состояния, даже целые отдельные диаграммы вложенных состояний и переходов между ними. Пребывая в таком состоянии, система находится ровно в одном из его *подсостояний*. На Рис. 38 почти все изображенные состояния являются подсостояниями состояния `Site`. Кроме того, в нижней части диаграммы три состояния объединены, чтобы показать, что переход по действию `cancel` возможен в каждом из них и приводит в одно и то же состояние `Basket`.

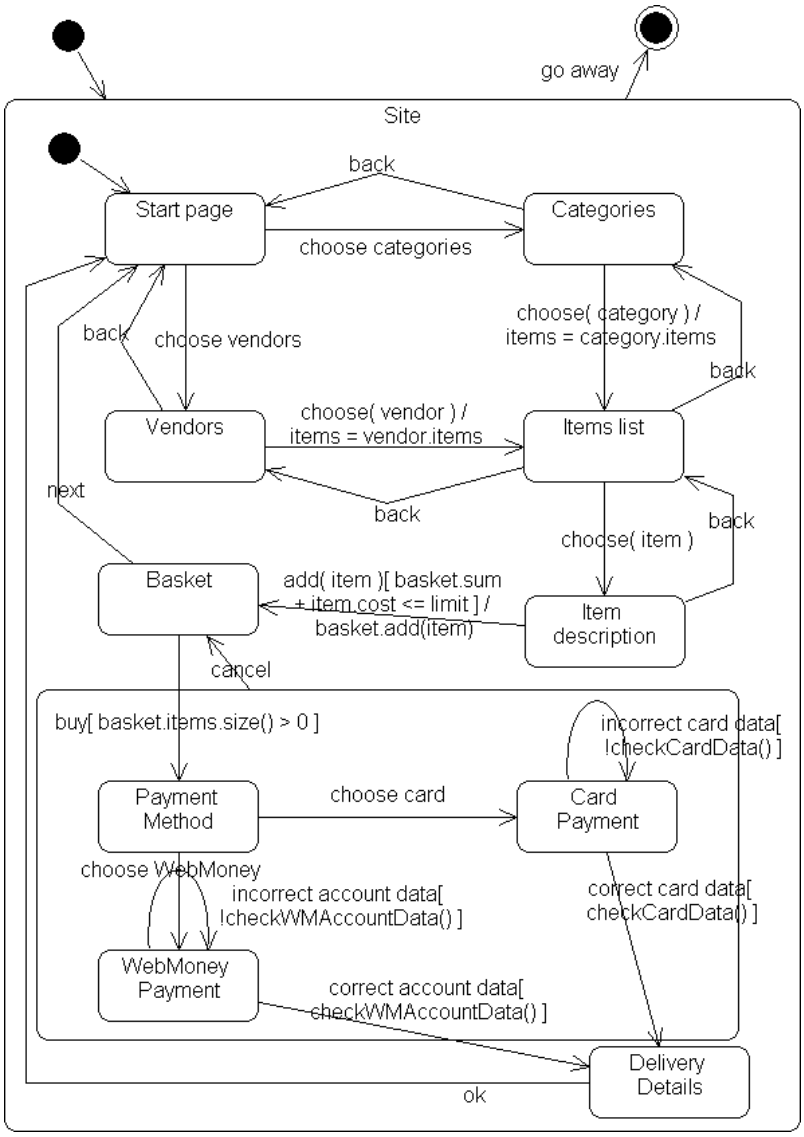


Рисунок 38. Пример диаграммы состояний, моделирующей сайт Интернет-магазина.

Состояние может декомпозироваться и на *параллельные подсостояния.* Они изображаются как области внутри объемлющего состояния, разделенные пунктирными линиями, их аналогом на диаграммах деятельности являются дорожки. Пребывая в объемлющем

состоянии, система должна находиться одновременно в каждом из его параллельных подсостояний.

Помимо показанных на диаграмме состояний изображаемая подсистема может иметь глобальные (в ее рамках) переменные, хранящие какие-то данные. Значения этих переменных являются общими частями всех изображаемых состояний.

На Рис. 38 примерами переменных являются список видимых пользователем товаров, `items`, и набор уже отобранных товаров с количеством для каждого, корзина, `basket`.

Переходы могут происходить между состояниями одного уровня, но могут также вести из некоторого состояния в подсостояние соседнего или, наоборот, из подсостояния в некоторое состояние, находящее на том же уровне, что и объемлющее состояние.

На переходе между состояниями указываются следующие данные:

- *Событие*, приводящее к выполнению этого перехода. Обычно событие — это вызов некоторой операции в одном из объектов или приход некоторого сообщения, хотя могут указываться и абстрактные события.
  Например, из состояния `Categories` на Рис. 38 можно выйти, выполнив команду браузера «Назад». Она соответствует событию `back`, инициирующему переход в состояние `Start page`. Другой переход из состояния `Categories` происходит при выборе категории товаров пользователем. Соответствующее событие имеет параметр — выбранную категорию. Оно изображено как `choose(category)`.
- *Условие выполнения* (*охранное условие, guardian*). Это условие, зависящее от параметров события и текущих значений глобальных переменных, выполнение которого необходимо для выполнения перехода. При наступлении нужного события переход выполняется, только если его условие тоже выполнено.
  Условие перехода показывается в его метке в квадратных скобках.
  На Рис. 38 примером условного перехода является переход из состояния `Basket` в состояние `Payment Method`. Он выполняется, только если пользователь выполняет команду «Оплатить» (событие `buy`) и при этом в его корзине есть хотя бы один товар.
- *Действие*, выполняемое в дополнение к переходу между состояниями. Обычно это вызовы каких-то операций и изменения значения глобальных переменных.
  Действие показывается в метке перехода после слеша (символа ‘/’). При этом изменения значений переменных перечисляются в начале, затем, после знака ‘^’, указывается вызов операции.
  Например, на Рис. 38 при выборе пользователем категории товаров происходит переход из состояния `Categories` в `Items list`. При этом список товаров, видимый пользователю, инициализируется списком товаров выбранной категории.

Диаграммы состояний используются часто, хотя требуется довольно много усилий, чтобы разработать их с достаточной степенью детальности.

Литература к Лекции 6

[1] Л. Басс, П. Клементс, Р. Кацман. Архитектура программного обеспечения на практике. СПб.: Питер, 2006.
[2] IEEE Std 1016-1998 Recommended Practice for Software Design Descriptions.
[3] IEEE 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems.
[4] R. Kazman et al. SAAM: A Method for Analyzing the Properties of Software Architectures. Proceedings of the 16-th International Conference on Software Engineering, 1994.
[5] Г. Буч, Дж. Рамбо, А. Джекобсон. Язык UML. Руководство пользователя. М.: ДМК, 2000.
[6] Дж. Рамбо, А. Якобсон, Г. Буч. UML: Специальный справочник. СПб.: Питер, 2002.
[7] М. Фаулер, К. Скотт. UML в кратком изложении. М.: Мир, 1999.
[8] И. Соммервилл. Инженерия программного обеспечения. М.: Вильямс, 2002.
[9] Э. Дж. Брауде. Технология разработки программного обеспечения. СПб.: Питер, 2004.

Лекция 7. Образцы проектирования

Аннотация

Рассматривается понятие образца проектирования, классификация образцов проектирования и некоторые широко используемые примеры образцов анализа и архитектурных стилей.

Ключевые слова

Образец проектирования, архитектурный стиль, идиома, образец анализа, образец организации, образец процесса, архитектурный стиль «каналы и фильтры», архитектурный стиль «многоуровневая система».

Текст лекции

Образцы человеческой деятельности

Чем отличается работа опытного проектировщика программного обеспечения от работы новичка? Имеющийся у эксперта опыт позволяет ему аккуратнее определять задачи, которые необходимо решить, точнее выделять среди них наиболее важные и менее значимые, четче представлять ограничения, в рамках которых должна работать будущая система. Но важнее всего то, что эксперт отличается накопленными знаниями о приемлемых или не приемлемых в определенных ситуациях решениях, о свойствах программных систем, обеспечиваемых ими, и способностью быстро подготовить качественное решение сложной проблемы, опираясь на эти знания.

Давней мечтой преподавателей всех дисциплин является выделение таких знаний «в чистом виде» и эффективная передача их следующим поколениям специалистов. В области проектирования сложных систем на роль такого представления накопленного опыта во второй половине XX века стали претендовать *образцы проектирования (design patterns* или просто *patterns)*, называемые также типовыми решениями или шаблонами. Наиболее широко образцы применяются при построении сложных систем, на которые накладывается множество разнообразных требований. Одной из первых работ, которая систематически излагает довольно большой набор образцов, относящихся именно к разработке программ, стала книга [1].

На основе имеющегося опыта исследователями и практиками разработки ПО выделено множество образцов — типовых архитектур, проектных решений для отдельных подсистем и модулей или просто программистских приемов, — позволяющих получить достаточно качественные решения типовых задач, а не изобретать каждый раз велосипед.

Более того, люди, наиболее активно вовлеченные в поиск образцов проектирования в середине 90-х годов прошлого века, пытались создать основанные на образцах языки, которые, хотя и были бы специфичными для определенных предметных областей, имели бы более высокий уровень абстракции, чем обычные языки программирования. Предполагалось, что человек, знакомый с таким языком, практически без усилий сможет создавать приложения в данной предметной области, компонуя подходящие образцы нужным способом. Эту программу реализовать так и не удалось, однако выявленные образцы, несомненно, являются одним из самых значимых средств передачи опыта проектирования сложных программных систем.

Образец (pattern) представляет собой шаблон решения типовой, достаточно часто встречающейся задачи в некотором контексте, т.е. при некоторых ограничениях на ожидаемые решения и определенном наборе требований к ним.

В качестве примера рассмотрим такую ситуацию. Мы разработали большую программу из многих модулей. Так сложилось, что почти все они опираются на некоторый выделенный модуль и часто используют его операции. В какой-то момент, однако, разработчик этого модуля решил поменять названия операций в его интерфейсе и порядок следования параметров (может быть и так, что его разработчиком является другая организация, у которой этот модуль был приобретен, и такие изменения в нем появились в очередной версии, в которой исправлены многие серьезные ошибки). Изменить код других модулей системы достаточно тяжело, так как вызовы операций

данного модуля используются во многих местах. А если придется работать с несколькими разными версиями — не менять же код каждый раз!

Другим примером такой ситуации является разработка набора тестов для некоторых операций. Хотелось бы, чтобы с помощью этого набора можно было бы тестировать любую возможную реализацию функций, выполняемых этими операциями. Если функции достаточно часто встречаются, например, совместно реализуют очередь, хранящую некоторые элементы, то такая возможность очень полезна. Но у каждого набора операций может быть свой интерфейс, переделывать все тесты под который слишком трудоемко.

Если можно представить набор требуемых операций как интерфейс некоторого класса в объектно-ориентированном языке программирования, достойно выйти из такой ситуации поможет образец проектирования *адаптер (adapter)*.

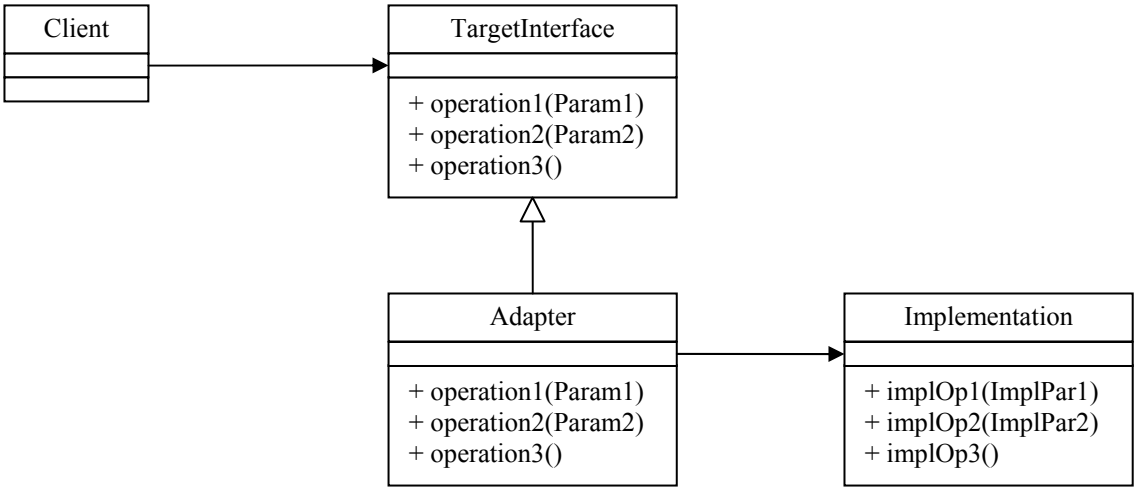


Рисунок 39. Структура классов-участников образца адаптер.

Предлагаемое решение состоит в следующем. Операции, которые необходимы для работы нашей системы, называемой *клиентом*, объединяются в некоторый класс или интерфейс (называемый *целевым*), и система пишется так, что она работает с объектом этого типа и его операциями. Получив *реализацию* тех же функций с отличающимися именами и типами параметров, мы определяем *адаптер* — класс-наследник целевого класса (или реализующий соответствующий интерфейс), в котором перегружаем нужные нам операции, выражая их через имеющуюся реализацию. При этом каждый раз объем дополнительной работы достаточно мал (если, конечно, полученная реализация действительно реализует нужные функции), а код клиента остается неизменным.

Образец проектирования нельзя выдумать или изобрести. Некоторый шаблон решения можно считать кандидатом в образцы проектирования, если он неоднократное применялся для решения одной и той же задачи на практике, если решения на его основе использовались в нескольких (как минимум, трех) случаях, в различных системах.

Образцы проектирования часто сильно связаны друг с другом в силу того, что они решают смежные задачи. Поэтому часто наборы связанных, поддерживающих друг друга образцов представляются вместе в виде систем *образцов (pattern system)* или *языка образцов (pattern language)*, в которых указаны возникающие между ними связи и описываются ситуации, в которых полезно совместное использование нескольких образцов.

По типу решаемых задач выделяют следующие разновидности образцов.

- ***Образцы анализа (analysis patterns)***.
  Они представляют собой типовые решения при моделировании сложных взаимоотношений между понятиями некоторой предметной области. Обычно они являются представлением этих понятий и отношений между ними с помощью набора классов и их связей, подходящего для любого объектно-ориентированного языка. Такие представления обладают важными атрибутами качественных модельных решений — способностью

отображать понятным образом большое многообразие ситуаций, возникающих в реальной жизни, отсутствием необходимости вносить изменения в модель при небольших изменениях в требованиях к основанному на ней программному обеспечению и удобством внесения изменений, вызванных естественными изменениями в понимании моделируемых понятий. В частности, небольшое расширение данных, связанных с некоторым понятием, приводит к небольшим изменениям в структуре, чаще всего, лишь одного класса модели. Образцы анализа могут относиться к определенной предметной области, как следует из их определения, но могут также и с успехом быть использованы для моделирования понятий в разных предметных областях.
В отличие от образцов проектирования и идиом (см. ниже), образцы анализа используются при концептуальном моделировании и не отражают прямо возможную реализацию такой модели в виде конкретного кода участвующих в ней классов. Например, поле `X` класса концептуальной модели в реализации может остаться полем, а может превратиться в пару методов `getX()` и `setX()` или в один метод `getX()` (т.е. в *свойство*, *property*, в терминах C# и JavaBeans).

- ***Архитектурные образцы*** или ***архитектурные стили (architectural styles, architectural patterns)***.
  Такие образцы представляют собой типовые способы организации системы в целом или крупных подсистем, задающие некоторые правила выделения компонентов и реализации взаимодействий между ними.
- ***Образцы проектирования (design patterns)*** в узком смысле.
  Они определяют типовые проектные решения для часто встречающихся задач среднего уровня, касающиеся структуры одной подсистемы или организации взаимодействия двух-трех компонентов.
- ***Идиомы (idioms, programming patterns)***.
  Идиомы являются специфическими для некоторого языка программирования способами организации элементов программного кода, позволяющими, опять же, решить некоторую часто встречающуюся задачу.
- ***Образцы организации (organizational patterns)*** и ***образцы процессов (process patterns)***.
  Образцы этого типа описывают успешные практики организации разработки ПО или другой сложной деятельности, позволяющие решать определенные задачи в рамках некоторого контекста, который включает ограничения на возможные решения.

Для описания образцов были выработаны определенные шаблоны. Далее мы будем использовать один из таких шаблонов для описания архитектурных стилей, образцов проектирования и идиом. Этот шаблон включает в себя следующие элементы.

- Название образца, а также другие имена, под которыми этот образец используется.
- Назначение — задачи, которые решаются с помощью данного образца. В этот же пункт включается описание контекста, в котором данный образец может быть использован.
- Действующие силы — ограничения, требования и идеи, под воздействием которых вырабатывается решение.
- Решение — основные идеи используемого решения. Включает следующие подпункты.
  - Структура — структура компонентов, принимающих участие в данном образце, и связей между ними. В рамках образца компоненты принято именовать исходя из ролей, которые они в нем играют.
  - Динамика — основные сценарии совместной работы компонентов образца.
  - Реализация — возможные проблемы при реализации и способы их преодоления, примеры кода на различных языках (в данном курсе мы будем использовать для примеров только язык Java). Варианты и способы уточнения данного образца.
  - Следствия применения образца — какими дополнительными свойствами, достоинствами и недостатками, обладают полученные на его основе решения.

- Известные примеры использования данного образца.
- Другие образцы, связанные с данным.

Далее в этой лекции рассматриваются некоторые из известных образцов в соответствии с приведенной классификацией. Другие образцы будут упоминаться в последующих лекциях при рассмотрении способов решения тех или иных задач, а также библиотек языков Java и C#.

Образцы анализа

Образец анализа является типовым решением по представлению набора понятий некоторой предметной области в виде набора классов и связей между ними. Основной источник описаний выделенных образцов анализа — это работы Мартина Фаулера (Martin Fowler) [2,3].

В качестве примера образцов анализа рассмотрим группу образцов, связанных с представлением в программной системе данных измерений и наблюдений.

Наиболее простым образцом этой группы является образец *величина (quantity)*. Результаты большинства измерений имеют количественное выражение, однако, если представлять их в виде атрибутов числовых типов (рост — 182, вес — 83), часть информации пропадает. Пока все пользователи системы и разработчики, вносящие в нее изменения, помнят, *в каких единицах* измеряются все хранимые величины, все в порядке, но стоит хоть одному ошибиться — и результаты могут быть весьма серьезны. Такого рода ошибка в 1998 году вывела из строя американский космический аппарат Mars Climate Orbiter, предназначавшийся для исследования климата Марса. Данные о текущих параметрах движения аппарата поступали на Землю, обрабатывались, и результирующие команды отправлялись обратно. При этом процедуры мониторинга и управления движением на самом аппарате воспринимали величину импульса как измеренную в Ньтонах на секунду, а программы обработки данных на Земле — как значение импульса в фунтах силы на секунду. В итоге это привело к выходу на гораздо более низкую, чем планировалось, орбиту, потере управления и гибели аппарата стоимостью около 130 миллионов долларов [4].

Поэтому более аккуратное решение — использовать для хранения данных числовых измерений объекты специального класса `Quantity`, в полях которого хранится не только значение величины, но и единица ее измерения. Кроме того, весьма полезно определить операции сложения, вычитания и сравнения таких величин.

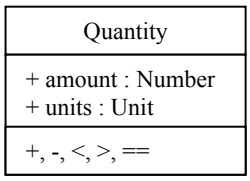


Рисунок 40. Класс для представления величин, имеющих разные единицы измерения.

Помимо измерений, использовать такое представление удобно и для сумм денег в финансовых системах. Аналогом единиц измерения в этом случае выступают различные валюты. От физических величин валюты отличаются изменяемым отношением, с помощью которого их можно переводить одну в другую. Это отношение может зависеть от времени. Кроме того, существуют единицы измерения физических величин, которые преобразуются друг в друга более сложным, чем умножение на некоторое число, способом — например, градусы по Фаренгейту и по Цельсию.

Эти примеры могут быть охвачены образцом *преобразование*, который позволяет представлять в системе правила преобразования различных единиц измерения друг в друга. Для большинства преобразований достаточно величины отношения между единицами, быть может, зависящего от времени, поэтому стоит выделить класс для хранения этого отношения.

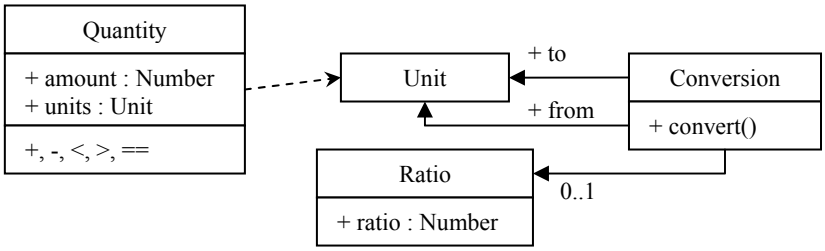


Рисунок 41. Представление возможных преобразований между единицами измерений.

Другой тип связи между различными единицами измерения — так называемые *составные единицы*, например Ньютон для измерения силы (1 Н = 1 кг*м/с^2). Разрешение подобного рода соотношений может быть реализовано, если определить два подкласса класса `Unit` — один для представления простых единиц, `PrimeUnit`, другой для представления составных, `CompoundUnit`, и определить две связи, сопоставляющие одной составной единице два мультимножества простых — те, что участвуют в ней в положительных степенях, и те, что участвуют в отрицательных.

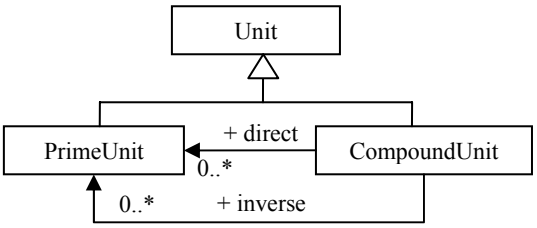


Рисунок 42. Представление составных единиц измерений.

В медицине, где хранение данных измерений имеет особое значение, измерения почти всегда связываются с пациентом, для которого они производились. К тому же, медицинских измерений, имеющих, например, значение длины, очень много. Для того, чтобы различать оба этих атрибута измерения — объект измерения и вид измерения (например, пациент Иванов Петр Сергеевич и окружность его талии), их нужно явно ввести в модель. Так возникает образец *измерение*. Этот образец становится полезным, если имеется очень много различных измерений для каждого объекта, группируемых в достаточно много видов измеряемых явлений.

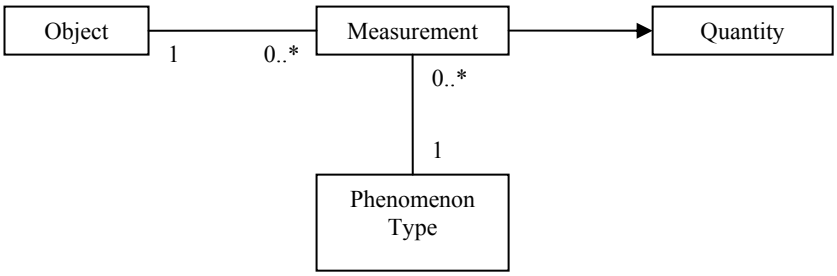


Рисунок 43. Набор классов для представления результатов измерений.

Бывает, однако, необходимо вести учет не только количественных измерений, но и качественных наблюдений, результат которых представляется не числом, а некоторым значением перечислимого типа (группа крови II, ожог 3-й степени и пр.). При этом наблюдения очень похожи на измерения: относятся к некоторому объекту и определяют некоторое значение для какого-то вида наблюдений.

Для совместного представления результатов наблюдений и измерений можно использовать образец *наблюдение*, структура классов которого показана на Рис. 44. Требуется некоторая привычка, чтобы быстро разложить по этим классам какой-нибудь реальный пример. Например, группа крови — вид явлений, II — явление этого вида, наблюдение заключается в том, что у Петра Сергеевича Иванова была обнаружена именно такая группа крови. Эти усилия, однако, с лихвой окупаются огромным количеством фактов, которые без изменений можно уложить в эту схему.

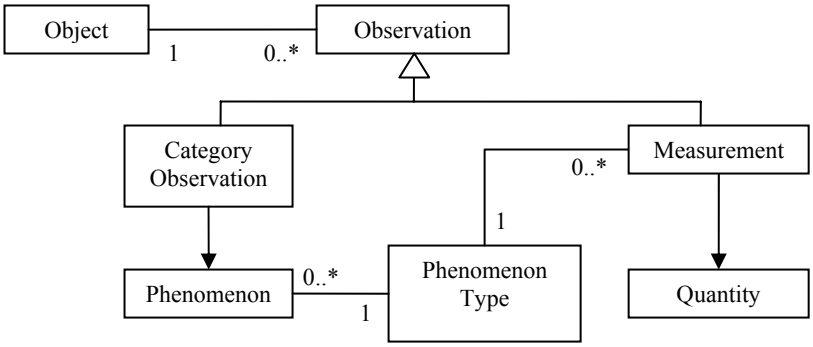


Рисунок 44. Набор классов для представления результатов как измерений, так и наблюдений.

Архитектурные стили

Архитектурный стиль определяет основные правила выделения компонентов и организации взаимодействия между ними в рамках системы или подсистемы в целом. Различные архитектурные стили подходят для решения различных задач в плане обеспечения нефункциональных требований — различных уровней производительности, удобства использования, переносимости и удобства сопровождения. Одну и ту же функциональность можно реализовать, используя разные стили.

Работа по выделению и классификации архитектурных стилей была проведена в середине 1990-х годов. Ее результаты представлены в работах [5,6]. Ниже приведена таблица некоторых архитектурных стилей, выделенных в этих работах.

Виды стилей и конкретные стили	**Контекст использования и основные решения**	**Примеры**
Конвейер обработки данных (data flow)	Система выдает четко определенные выходные данные в результате обработки четко определенных входных данных, при этом процесс обработки не зависит от времени, применяется многократно, одинаково к любым данным на входе. Обработка организуется в виде набора (не обязательно последовательности) отдельных компонентов-обработчиков, передающих свои результаты на вход другим обработчикам или на выход всей системы. Важными свойствами являются четко определенная структура данных и возможность интеграции с другими системами.	
Пакетная обработка (batch sequential)	Один-единственный вывод производится на основе чтения некоторого одного набора данных на входе, промежуточные преобразования организуются в виде последовательности.	Сборка программной системы: компиляция, сборка системы, сборка документации, выполнение тестов.
Каналы и фильтры (pipe-and-filter)	Нужно обеспечить преобразование непрерывных потоков данных. При этом преобразования инкрементальны и следующее может быть начато до окончания предыдущего. Имеется, возможно, несколько входов и несколько выходов. В дальнейшем возможно добавление дополнительных преобразований.	Утилиты UNIX

	Замкнутый цикл управления (closed-loop control)	Нужно обеспечить обработку постоянно поступающих событий в плохо предсказуемом окружении. Используется общий диспетчер событий, который классифицирует событие и отдает его на асинхронную обработку обработчику событий такого типа, после чего диспетчер снова готов воспринимать события.	Встроенные системы управления в автомобилях, авиации, спутниках. Обработка запросов на сильно загруженных Web-серверах. Обработка действий пользователя в GUI.
Вызов-возврат (call-return)		Порядок выполнения действий четко определен, отдельные компоненты не могут выполнять полезную работу, не получая обращения от других.	
	Процедурная декомпозиция	Данные неизменны, процедуры работы с ними могут немного меняться, могут возникать новые. Выделяется набор процедур, схема передачи управления между которыми представляет собой дерево с основной процедурой в его корне.	Основная схема построения программ для языков C, Pascal, Ada
	Абстрактные типы данных (abstract data types)	В системе много данных, структура которых может меняться. Важны возможности внесения изменений и интеграции с другими системами. Выделяется набор абстрактных типов данных, каждый из которых предоставляет набор операций для работы с данными такого типа. Внутреннее представление данных скрывается.	Библиотеки классов и компонентов
	Многоуровневая система (layers)	Имеется естественное расслоение задач системы на наборы задач, которые можно было бы решать последовательно — сначала задачи первого уровня, затем, используя полученные решения, — второго, и т.д. Важны переносимость и возможность многократного использования отдельных компонентов. Компоненты разделяются на несколько уровней таким образом, что компоненты данного уровня могут использовать для своей работы только соседей или компоненты предыдущего уровня. Могут быть более слабые ограничения, например, компонентам верхних уровней разрешено использовать компоненты всех нижележащих уровней.	Телекоммуникационные протоколы в модели OSI (7 уровней), реальные протоколы сетей передачи данных (обычно 5 уровней или меньше). Системы автоматизации предприятий (уровни интерфейса пользователя-обработки запросов-хранения данных).
	Клиент-сервер	Решаемые задачи естественно распределяются между инициаторами и обработчиками запросов, возможно изменение внешнего представления данных и способов их обработки.	Основная модель бизнес-приложений: клиентские приложения, воспринимающие запросы пользователей и сервера, выполняющие эти запросы.

Интерактивные системы		Необходимость достаточно быстро реагировать на действия пользователя, изменчивость пользовательского интерфейса.	
	Данные–представление–обработка (model-view-controller, MVC)	Изменения во внешнем представлении достаточно вероятны, одна и та же информация представляется по-разному в нескольких местах, система должна быстро реагировать на изменения данных. Выделяется набор компонентов, ответственных за хранение данных, компоненты, ответственные за их представления для пользователей, и компоненты, воспринимающие команды, преобразующие данные и обновляющие их представления.	Наиболее часто используется при построении приложений с GUI. Document-View в MFC (Microsoft Foundation Classes) — документ в этой схеме объединяет роли данных и обработчика.
	Представление–абстракция–управление (presentation-abstraction-control)	Интерактивная система на основе агентов, имеющих собственные состояния и пользовательский интерфейс, возможно добавление новых агентов. Отличие от предыдущей схемы в том, что для каждого отдельного набора данных его модель, представление и управляющий компонент объединяются в агента, ответственного за всю работу именно с этим набором данных. Агенты взаимодействуют друг с другом только через четко определенную часть интерфейса управляющих компонентов.	
Системы на основе хранилища данных		Основные функции системы связаны с хранением, обработкой и представлением больших количеств данных.	
	Репозиторий (repository)	Порядок работы определяется только потоком внешних событий. Выделяется общее хранилище данных — репозиторий. Каждый обработчик запускается в ответ на соответствующее ему событие и как-то преобразует часть данных в репозитории.	Среды разработки и CASE-системы
	Классная доска (blackboard)	Способ решения задачи в целом неизвестен или слишком трудоемок, но известны методы, частично решающие задачу, композиция которых способна выдавать приемлемые результаты, возможно добавление новых потребителей данных или обработчиков. Отдельные обработчики запускаются, только если данные репозитория для их работы подготовлены. Подготовленность данных определяется с помощью некоторой системы шаблонов. Если можно запустить несколько обработчиков, используется система их приоритетов.	Системы распознавания текста

Таблица 7. Некоторые архитектурные стили.

Многие из представленных стилей носят достаточно общий характер и часто встречаются в разных системах. Кроме того, часто можно обнаружить, что в одной системе используются несколько архитектурных стилей — в одной части преобладает один, в другой — другой, или же один стиль используется для выделения крупных подсистем, а другой — для организации более мелких компонентов в подсистемах.

Более подробного рассмотрения заслуживают стили «Каналы и фильтры», «Многоуровневая система». Далее следуют их описания согласно [7].

Каналы и фильтры

Название. Каналы и фильтры (pipes and filters).

Назначение. Организация обработки потоков данных в том случае, когда процесс обработки распадается на несколько шагов. Эти шаги могут выполняться отдельными обработчиками, возможно, реализуемыми разными разработчиками или даже организациями. При этом нужно принимать во внимание следующие факторы.

Действующие силы.

- Должны быть возможны изменения в системе за счет добавления новых способов обработки и перекомбинации имеющихся обработчиков, иногда самими конечными пользователями.
- Небольшие шаги обработки проще переиспользовать в различных задачах.
- Не являющиеся соседними обработчики не имеют общих данных.
- Имеются различные источники входных данных — сетевые соединения, текстовые файлы, сообщения аппаратных датчиков, базы данных.
- Выходные данные могут быть востребованы в различных представлениях.
- Явное хранение промежуточных результатов может быть неэффективным, создаст множество временных файлов, может привести к ошибкам, если в его организацию сможет вмешаться пользователь.
- Возможно использование параллелизма для более эффективной обработки данных.

Решение. Каждая отдельная задача по обработке данных разбивается на несколько мелких шагов. Выходные данные одного шага являются входными для других. Каждый шаг реализуется специальным компонентом — *фильтром (filter)*. Фильтр потребляет и выдает данные инкрементально, небольшими порциями. Передача данных между фильтрами осуществляется по *каналам (pipes)*.

Структура. Основными ролями компонентов в рамках данного стиля являются фильтр и канал. Иногда выделяют специальные виды фильтров — *источник данных (data source)* и *потребитель данных (data sink)*, которые, соответственно, только выдают данные или только их потребляют. Каждый поток обработки данных состоит из чередующихся фильтров и каналов, начинается источником данных и заканчивается их потребителем.

Фильтр получает на свой вход данные и обрабатывает их, дополняя их результатами обработки, удаляя какие-то части и трансформируя их в некоторое другое представление. Иногда фильтр сам требует входные данные и выдает выходные по их получении, иногда он, наоборот, может реагировать на события прихода данных на вход и требования данных на выходе. Фильтр обычно потребляет и выдает данные некоторыми порциями.

Канал обеспечивает передачу данных, их буферизацию и синхронизацию обработки их соседними фильтрами (например, если оба соседних фильтра активны, работают в параллельных процессах). Если никакой дополнительный буферизации и синхронизации не требуется, канал может представлять собой простую передачу данных в виде параметра или результата вызова операции.

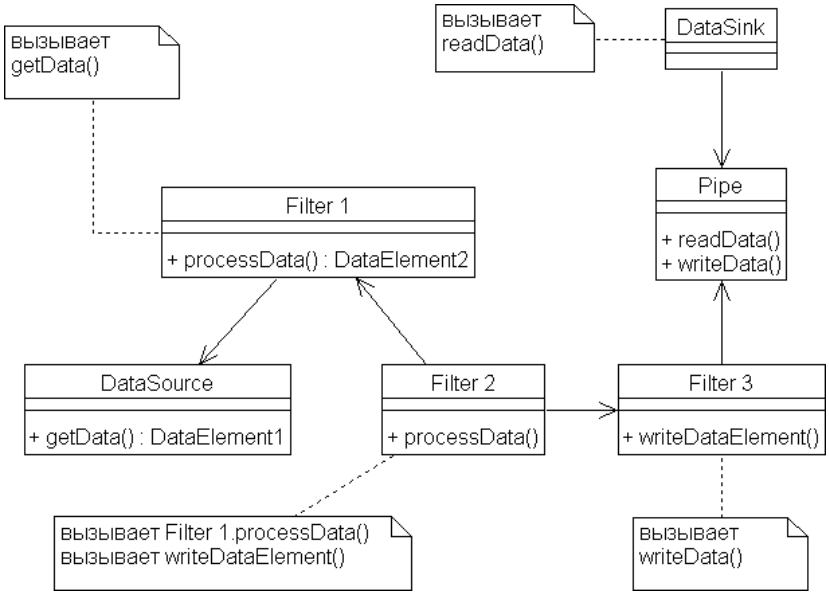


Рисунок 45. Пример структуры классов для образца каналы и фильтры.

На Рис. 45 показан пример диаграммы классов для данного образца, в котором 3 канала реализованы неявно — через вызовы операций и возвращение результатов, а один — явно. Из участвующих в этом примере фильтров источник и потребитель данных, а также Filter 1 запрашивают входные данные, Filter 3 сам передает их дальше, а Filter 2 и запрашивает, и передает данные самостоятельно.

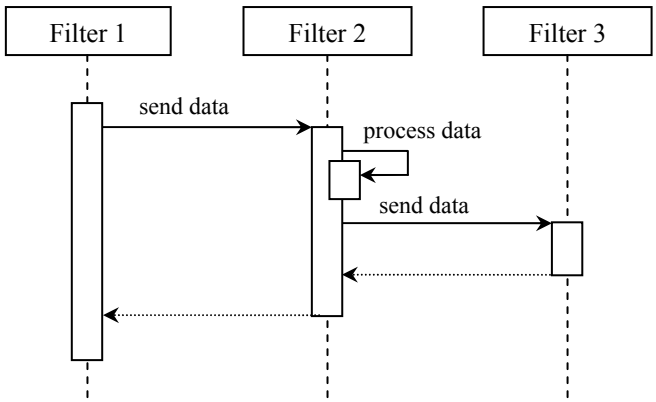


Рисунок 46. Сценарий работы проталкивающего фильтра.

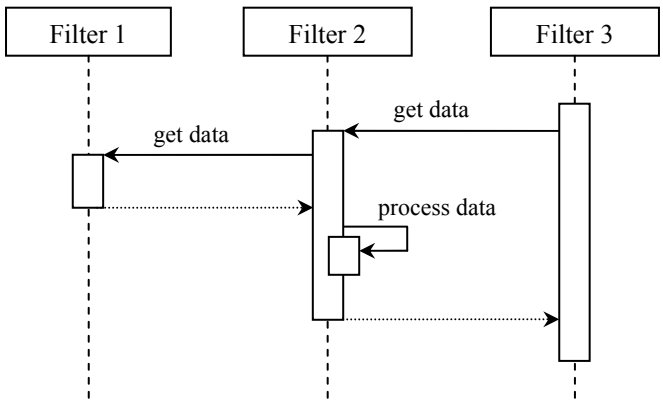


Рисунок 47. Сценарий работы вытягивающего фильтра.

Динамика. Возможны три различных сценария работы одного фильтра — проталкивание данных (push model, фильтр сам передает данные следующему компоненту, а получает их только в результате передачи предыдущего), вытягивание данных (pull model, фильтр требует данные у предыдущего компонента, следующий сам должен затребовать данные у

него) и смешанный вариант. Часто реализуется только один вид передачи данных для всех фильтров в рамках системы. Кроме того, канал может буферизовать данные и синхронизовать взаимодействующие с ним фильтры. Сценарии работы системы в целом строятся в виде различных комбинаций вариантов работы отдельных фильтров.

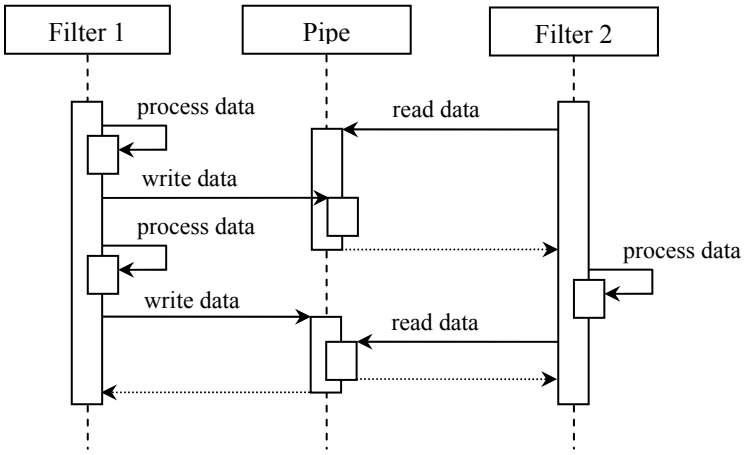


Рисунок 48. Сценарий работы буферизующего и синхронизующего канала.

Реализация. Основные шаги реализации следующие.

- Определить шаги обработки данных, необходимые для решения задач системы. Очередной шаг должен зависеть только от выходных данных предшествующего шага.
- Определить форматы данных при их передаче по каждому каналу.
- Определить способ реализации каждого канала, проталкивание или вытягивание данных, необходимость дополнительной буферизации и синхронизации.
- Спроектировать и реализовать необходимый набор фильтров. Реализовать каналы, если для их представления нужны отдельные компоненты.
- Спроектировать и реализовать обработку ошибок. Обработку ошибок при применении этого стиля достаточно тяжело организовать, поэтому ею часто пренебрегают. Однако, требуется, как минимум, адекватная диагностика случающихся на разных этапах ошибок.
  Могут быть выделены специальные каналы для передачи сообщений об ошибках. При возникновении ошибок ввода соответствующий фильтр может игнорировать дальнейшие входные данные до получения определенного разделителя, гарантирующего, что после него идут данные, не связанные с предыдущими.
- Сконфигурировать необходимый конвейер обработки данных, собрав вместе нужные фильтры и соединяющие их каналы.

Следствия применения образца.

Достоинства.

- Промежуточные данные могут не храниться в файлах, но могут и храниться, если это необходимо для каких-то дополнительных целей.
- Фильтры можно легко заменять, переиспользовать, менять местами, переставлять и комбинировать, реализуя множество функций на основе одних и тех же компонентов.
- Конвейерные системы обработки данных могут быть разработаны очень быстро, если имеется богатый набор фильтров.
- Активные фильтры могут работать параллельно, давая в результате более эффективное решение на многопроцессорных системах.

Недостатки.

- Управление обработкой с помощью большого общего состояния, которое иногда необходимо, не может быть эффективно реализовано с помощью этого стиля.
- Часто параллельная обработка не приносит никакого повышения производительности, поскольку передача данных между фильтрами может быть достаточно дорогой, фильтры могут требовать всех входных данных, прежде чем выдадут хоть что-то, и их синхронизация с помощью каналов может приводить к значительным простоям.
- Часто фильтры большее время тратят на преобразование формата поступающих входных данных, чем на их обработку. Использование одного формата, например, текстового, также зачастую снижает эффективность их использования.
- Обработка ошибок в рамках данного стиля очень сложна. В том случае, если разрабатываемая система должна быть очень надежной, а возвращение к самому началу работы в случае обнаружения ошибки, так же как ее игнорирование не являются допустимыми сценариями, использовать этот стиль не стоит.

Примеры. Наиболее известный пример использования данного образца — система утилит UNIX [8], пополненная возможностями оболочки (shell) по организации каналов между процессами. Большинство утилит могут играть роль фильтров при обработке текстовых данных, а каналы строятся при помощи соединения стандартного ввода одной программы со стандартным выводом другой.

Другим примером может служить часто используемая архитектура компилятора как последовательности фильтров, обрабатывающих входную программу — лексического анализатора (лексера), синтаксического анализатора (парсера), семантического анализатора, набора оптимизаторов и генератора результирующего кода. Таким способом можно достаточно быстро построить прототипный компилятор для несложного языка. Более производительные компиляторы, нацеленные на промышленное использование, строятся по более сложной схеме, в частности, используя элементы стиля «Репозиторий».

Многоуровневая система

Название. Многоуровневая система (layers).

Назначение. Реализация больших систем, которые имеют большое количество разноплановых элементов, использующих друг друга. Некоторые аспекты работы таких систем могут включать в себя много операций, выполняемых разными компонентами на разных уровнях (т.е. одна задача решается за счет последовательных обращений между элементами разных уровней, другая — тоже, но участвующие в решении этих задач элементы могут быть различны). При этом нужно принимать во внимание следующие факторы.

Действующие силы.

- Изменения в требованиях к решению одной из задач не должны приводит к изменениям в коде многочисленных компонентов, желательно, чтобы они сводились к изменениям внутри одного компонента. То же касается и изменений платформы, на которой работает система.
- Интерфейсы между компонентами должны быть стабильными или даже соответствовать имеющимся стандартам.
- Части системы должны быть заменяемы. Компоненты должны быть заменяемы другими, если те реализуют такие же интерфейсы. В идеале может даже потребоваться в ходе работы переключиться на другую реализацию, даже если при начале работы системы она не была доступна.
- Низкоуровневые компоненты должны позволять разрабатывать другие системы быстрее.

- Компоненты с похожими областями ответственности должны быть сгруппированы для повышения понятности системы и удобства внесения в нее изменений.
- Нет возможности выделить компоненты некоторого стандартного размера: одни из них решают достаточно сложные задачи, другие — совсем простые.
- Сложные компоненты нуждаются в дальнейшей декомпозиции.
- Использование большого числа компонентов может отрицательно сказаться на производительности, поскольку данным придется часто преодолевать границы между компонентами.
- Разработка системы должна быть эффективно поделена между отдельными разработчиками. При этом интерфейсы и зоны ответственности компонентов, передаваемых разным разработчикам, должны быть очень четко определены.

Решение. Выделяется некоторый набор уровней, каждый из которых отвечает за решение своих собственных подзадач. Для этого он использует интерфейс, предоставляемый предыдущим уровнем, предоставляя, в свою очередь, некоторый интерфейс для следующего уровня.

Каждый отдельный уровень может быть в дальнейшем декомпозирован на более мелкие компоненты.

Структура. Основными компонентами являются *уровни*. Иногда выделяют *клиентов*, использующих интерфейс самого верхнего уровня. Каждый уровень предоставляет интерфейс для решения определенного множества задач. Сам он решает их, опираясь на интерфейс предшествующего уровня.

На каждом уровне может находиться много более мелких компонентов.

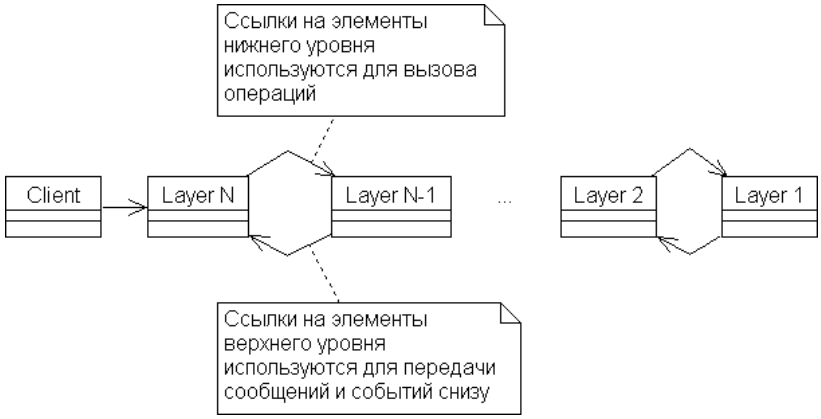


Рисунок 49. Пример структуры многоуровневой системы. Классы для уровней условные, они обычно декомпозируются на наборы более мелких компонентов.

Динамика. Сценарии работы системы могут быть получены компоновкой следующих четырех. Часто в виде многих уровней реализуются коммуникационные системы, две такие системы могут взаимодействовать через самый нижний уровень — при этом пара симметричных сценариев (по подъему-спуску обращений) выполняется в рамках одного общего сценария на разных машинах.

- Обращение клиента к верхнему уровню инициирует цепочку обращений с верхнего уровня до самого нижнего.
- Событие на нижнем уровне (например, приход сообщения по сети или нажатие на кнопку мыши) инициирует цепочку обращений, идущую снизу вверх, вплоть до некоторого события самого верхнего уровня, видимого клиентам.
- Обращение клиента к верхнему уровню приводит к цепочке вызовов, которая, однако, не доходит до самого низа. Такая ситуация реализуется, если, например, один из уровней кэширует ответы на запросы и может выдать ответ на ранее уже подававшийся запрос без обращения к более низким уровням.

- То же самое может произойти и с событием, которое передается с самого нижнего уровня. Дойдя до некоторого уровня, оно может поглотиться им (с изменением состояния каких-то компонентов), потому что не соответствует никакому событию на более высоких уровнях. Например, нажатие клавиши Capslock не приводит само по себе ни к каким реакциям программы, но изменяет значение нажимаемых после этого клавиш, меняя их регистр на противоположный.

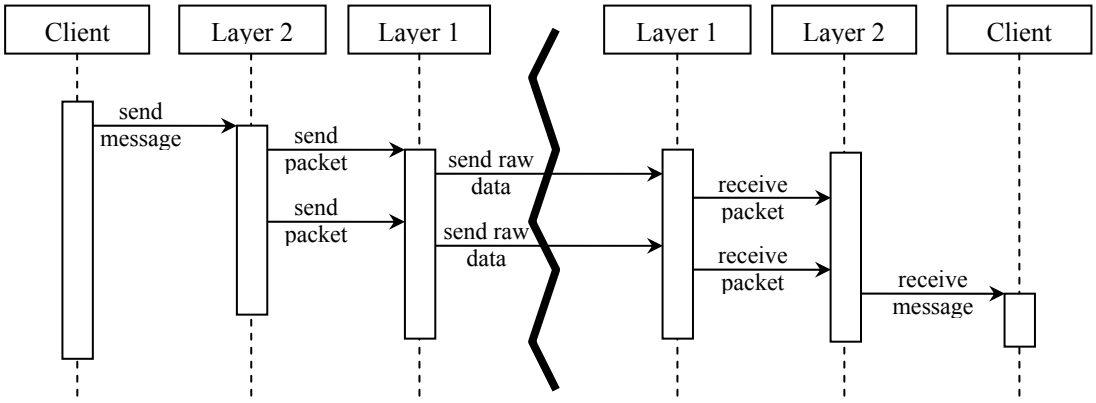


Рисунок 50. Составной сценарий пересылки сообщения по сети.

Реализация. Основные шаги реализации следующие.

- Определить критерии группировки задач по уровням. Это критически важный шаг. Неправильное распределение задач, скорее всего, приведет к необходимости перепроектировать систему.
- Определить количество уровней, которые будут реализованы, и их имена. Часто приходится объединять концептуально различные задачи, чтобы добиться большей эффективности системы. С другой стороны, произвольное смешение задач на уровне ведет к непонятной архитектуре и к системе, очень неудобной для сопровождения. При возможности поместить некоторую задачу на несколько уровней, стоит размещать ее на самом высоком уровне из тех, где она может быть решена с достаточной производительностью.
- Определить интерфейсы, предоставляемые нижними уровнями верхним. Здесь нужно помнить, что с помощью несколько б***о***льшего, чем минимально необходимый, набора интерфейсных операций нижнего уровня можно добиться значительного повышения производительности системы в целом.
- Определить компоненты и их взаимодействие в рамках каждого отдельного уровня.
- Определить способы взаимодействия соседних уровней. Можно использовать проталкивание, вытягивание данных или комбинацию этих подходов.
- Отделить соседние уровни. В идеале нижние уровни не должны знать ничего о верхних, каждый уровень должен знать только о непосредственно предшествующем ему. Для этого передачу данных с нижнего уровня можно организовать в виде обратных вызовов (callbacks) — указатель на функцию, которую нужно вызвать для передачи сообщения наверх, верхний уровень может передавать в качестве параметра при предшествующих запросах.
- Спроектировать и реализовать обработку ошибок. Ошибки лучше обрабатывать на самом нижнем уровне, который в состоянии их заметить.

Следствия применения образца.

Достоинства.

- Возможность легко заменять и переиспользовать компоненты одного уровня, не оказывая влияния на остальные уровни. Возможность отлаживать и тестировать уровни по отдельности.

- Поддержка стандартов. Многоуровневость системы делает возможной поддержку стандартных интерфейсов, таких как POSIX.

Недостатки.

- Изменение функциональности одного уровня может привести к каскадному изменению всех уровней. Существенный рост производительности нижнего уровня и требование обеспечить соответствующий рост производительности на более высоких уровнях также могут привести к переопределению всех интерфейсов.
- Падение производительности из-за необходимости все вызовы и данные проводить через все уровни.
- Часто уровни дублируют работу друг друга, например, при обработке ошибок, поскольку они разрабатываются независимо и не имеют информации о деталях реализации друг друга.
- Большое количество уровней может привести к существенному повышению сложности системы и падению ее производительности. С другой стороны, слишком малое число уровней (например, два) часто не позволяет обеспечить необходимую гибкость и переносимость.

Примеры. Наиболее известный пример использования данного образца — стандартная модель протоколов связи открытых систем (Open System Interconnection, OSI) [9]. Она состоит из 7-ми уровней.

- Самый нижний уровень — *физический*. Он отвечает за передачу отдельных битов по каналам связи. Основные его задачи — гарантировать правильное определение нуля и единицы разными системами, определить временные характеристики передачи (за какое время передается один бит), обеспечить передачу в одном или двух направлениях, и т.п.
- Второй уровень — *канальный* или *уровень передачи данных*. Его задача — предоставить верхним уровням такие сервисы, чтобы для них передача данных выглядела бы как посылка и прием потока байт без потерь и без перегрузок.
- Третий уровень — *сетевой*. Его задача — обеспечить прозрачную связь между компьютерами, не соединенными непосредственно, а также обеспечивать нормальную работу больших сетей, по которым одновременно путешествует очень много пакетов данных.
- Четвертый уровень — *транспортный*. Он обеспечивает надежную передачу данных верхних уровней словно по некоторой трубе — пакеты приходят обязательно в той же последовательности, в которой они были отправлены. Заметим, что канальный уровень решает такую же задачу, но только для непосредственно связывающихся друг с другом машин.
- Пятый, *сеансовый* уровень предоставляет возможность устанавливать сеансы связи (или сессии), содержащие некоторый набор передаваемых туда и обратно сообщений, и управлять ими.
- Шестой, *уровень представления*, определяет форматы передаваемых данных. Например, именно здесь определяется, что целое число будет представляться 4-мя байтами, причем старшие биты числа идут раньше младших, первый бит интерпретируется как знак, а отрицательные числа представляются в дополнительной системе (т.е. 0x0000000f обозначает 15, а 0x80000000f — $-2147483633 = -(2^{31}-15)$).
- Наконец, седьмой уровень — *прикладной* — содержит набор протоколов, которыми непосредственно пользуются программы и с которыми работают пользователи — HTTP, FTP, SMTP, POP3 и пр.

Модель OSI оказалась все же слишком сложна для использования на практике. Сейчас наиболее широко применяемые наборы протоколов строятся по урезанной схеме OSI — в

ней отсутствуют пятый и шестой уровни, прикладные протоколы пользуются непосредственно службами протоколов транспортного уровня.

Другой пример многоуровневой архитектуры — архитектура современных информационных систем или систем автоматизации бизнеса. Она включает следующие уровни [3].

- Интерфейс взаимодействия с внешней средой.
  Чаще всего этот уровень рассматривается как интерфейс пользователя. В его рамках определяется представление данных для передачи другим системам или пользователям, набор экранов, форм и отчетов, с которыми имеют дело пользователи.
- Бизнес-логика. На этом уровне реализуются основные правила функционирования данного бизнеса, данной организации.
- Предметная область. Данный уровень содержит концептуальную схему данных, с которыми имеет дело организация. Эти же данные могут использоваться и другими организациями в своей работе.
- Уровень управления ресурсами.
  На нем находятся все ресурсы, которыми пользуется система, в том числе другие системы. Очень часто используемые ресурсы сводятся к набору баз данных, необходимых для работы организации. На этом уровне определяется структура используемых ресурсов и способы управления ими, в частности, конкретное размещение данных по таблицам реляционной базы данных или классам объектной базы данных и соответствующий набор индексов. Чаще всего схемы баз данных оптимизируются под конкретный набор запросов, и поэтому их структура несколько отличается от концептуальной схемы данных, находящейся на предыдущем уровне.

Часто два средних уровня объединяются в один — уровень функционирования приложений, что дает в результате широко используемую *трехзвенную архитектуру* информационных систем.

Литература к Лекции 7

[1] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер-ДМК, 2001.
[2] M. Fowler. Analysis Patterns: Reusable Object Models. Addison-Wesley, 1997.
[3] М. Фаулер и др. Архитектура корпоративных программных приложений. М.: Вильямс, 2004.
[4] Mars Climate Orbiter Mishap Investigation Board Phase I Report.
Доступен по адресу ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf
[5] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Englewood Cliffs, NJ, 1996.
[6] M. Shaw and P. Clementz. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. Proceeding of COMPSAC, Washington, D.C., August 1997.
[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture. A System of Patterns. Wiley, 2002.
[8] Э. Таненбаум. Современные операционные системы. 2-е издание. СПб.: Питер, 2002.
[9] Э. Таненбаум. Компьютерные сети. 4-е издание. СПб.: Питер, 2003.
[10] Л. Басс, П. Клементс, Р. Кацман. Архитектура программного обеспечения на практике. СПб.: Питер, 2006.
[11] Э. Дж. Брауде. Технология разработки программного обеспечения. СПб.: Питер, 2004.

Лекция 8. Образцы проектирования (продолжение)

Аннотация

Рассматриваются дополнительные примеры образцов: архитектурный стиль «данные–представление–обработка», ряд образцов проектирования, идиом и образцов организации работ.

Ключевые слова

Образец проектирования, архитектурный стиль, идиома, образец организации, образец процесса, архитектурный стиль «данные-представление-обработка», образец проектирования «подписчик», идиома «шаблонный метод», инспекция программ по Фагану.

Текст лекции

В этой лекции мы продолжим разбирать примеры образцов — рассмотрим детально архитектурный стиль «Данные–представление–обработка», а также примеры тех видов образцов, которые не уместились в предыдущую лекцию: образцов проектирования в узком смысле, идиом и образцов организации.

Данные–представление–обработка

Название. Данные–представление–обработка (model–view–controller, MVC).

Назначение. Интерактивные приложения с гибким интерфейсом пользователя. Требования к пользовательскому интерфейсу в интерактивных приложениях меняются чаще всего. Разные пользователи имеют разные наборы требований. В несколько меньшей степени это касается методов обработки данных, лежащих в основе таких приложений, — визуальное представление управляющих элементов может меняться вместе с интерфейсом, а сами выполняемые действия зависят от бизнес-логики и предметной области, и поэтому более устойчивы. Наименее подвержена изменениям модель данных, с которыми работает приложение.

Поэтому для увеличения гибкости и удобства изменений в таких приложениях необходимо соответствующим образом разделить их компоненты. При этом нужно принимать во внимание следующие факторы.

Действующие силы.

- Одна и та же информация может быть представлена по-разному и в нескольких местах для удобства доступа к ней многих различных пользователей, имеющих разные привычки и разные навыки работы с информацией.
- Изменения в данных должны немедленно отображаться в различных представлениях этих данных.
- Внесение изменений в пользовательский интерфейс должно быть максимально простым, иногда оно даже должно быть возможно прямо во время работы приложения.
- Поддержка различных стандартов пользовательского интерфейса и его перенос между платформами не должны влиять на код, связанный с методами работы с данными и структурой данных приложения.

Решение. Выделяется три набора компонентов. Первый набор — *данные*, *модель данных* или просто *модель* (*model*) — соответствует структуре данных предметной области, в которой работает приложение. Обязанности этих компонентов: представлять в системе данные и базовые операции над ними. Компоненты второго набора — *представления* (*view*) — соответствуют различным способам представления данных в пользовательском интерфейсе. Для одних и тех же данных может иметься несколько представлений. Каждому компоненту представления соответствует один компонент из третьего набора, *обработчик* (*controller*) — компонент, осуществляющий обработку действий пользователей. Такой компонент получает команды, чаще всего нажатия клавиш и нажатия кнопок мыши в

областях, соответствующих визуальным элементам управления — кнопкам, элементам меню и пр. Эти команды он преобразует в действия над данными. В результате каждого действия требуется обновить все представления всех данных, которые подверглись изменениям.

Структура. Основными ролями компонентов в данном стиле являются *модель*, *представление* и *обработчик*.

Компонент-модель моделирует данные приложения, реализует основные операции над ними и возможность регистрировать зависимые от него обработчики и представления. При изменениях в данных модель оповещает о них все зарегистрированные компоненты.

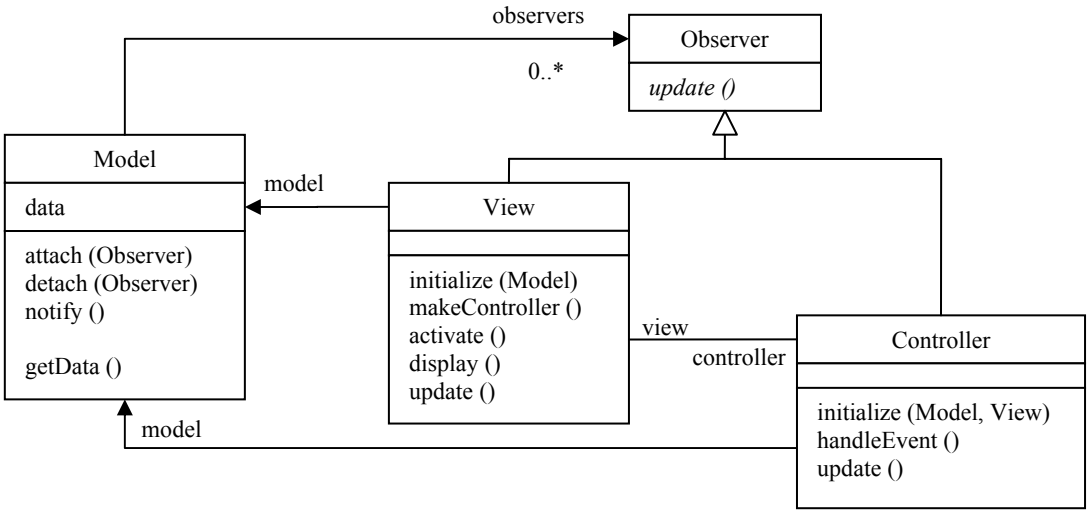


Рисунок 51. Структура классов модели, представления и обработчика.

Компонент-представление представляет данные в некотором виде для пользователей, читая их из модели при необходимости, т.е. при инициализации и после сообщений о произошедших изменениях. Кроме того, он инициализирует связанный с ним обработчик.

Компонент-обработчик обрабатывает действия пользователя, транслируя их в операции над моделью или запросы на показ некоторых элементов представлений. При оповещении об изменениях в модели он соответствующим образом изменяет собственное состояние, например, делая активными или отключая какие-нибудь кнопки и пункты меню.

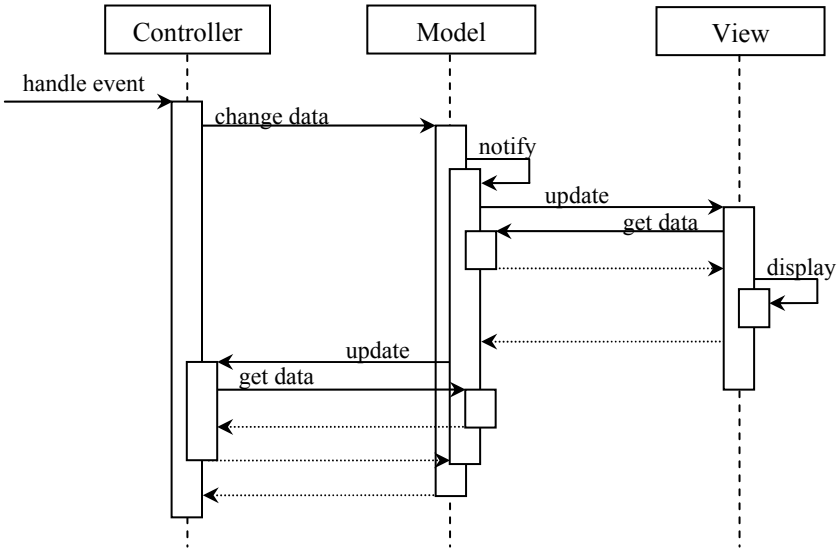


Рисунок 52. Сценарий обработки действия пользователя.

Динамика. У системы два базовых сценария работы — инициализация всех компонентов и обработка некоторого действия пользователя с изменением данных и обновлением соответствующих им представлений и обработчиков.

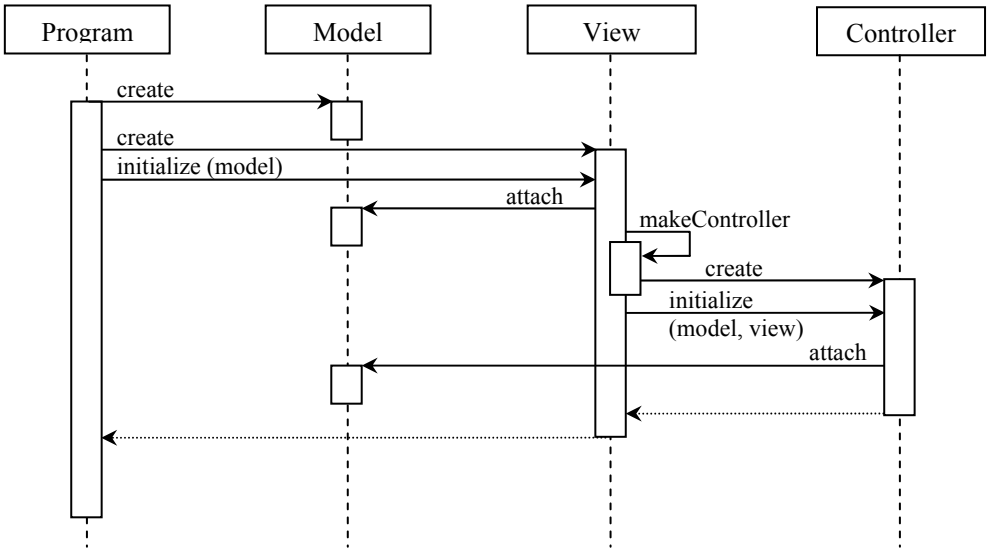


Рисунок 53. Сценарий инициализации системы.

Реализация. Основные шаги реализации следующие.

- Отделить взаимодействие человека с системой от базовых функций самой системы. Для этого необходимо выделить структуру данных, с которыми система работает, и набор необходимых для функционирования системы операций над ними.
- Реализовать механизм передачи изменений. Для этого можно воспользоваться образцом проектирования Подписчик (иначе называемым Наблюдатель, Observer).
- Спроектировать и реализовать необходимые представления.
- Спроектировать и реализовать необходимые обработчики действий пользователя.
- Спроектировать и реализовать связь между обработчиком и представлением. Обычно представление должно инициализировать соответствующий обработчик. Для этого можно, например, использовать образец проектирования Метод порождения (Factory Method).
- Реализовать построение системы из компонентов и инициализацию компонентов.

В качестве дополнительных аспектов реализации необходимо рассмотреть следующие.

- Динамические представления, создаваемые во время работы приложения.
- Подключаемые элементы управления, которые могут быть включены во время работы приложения. Например, переключение из режима «новичок» в режим «эксперт».
- Инфраструктура и иерархия представлений и обработчиков.
  Часто имеется готовая библиотека таких компонентов, на основе которых нужно строить собственные представления и обработчики. Эту задачу нужно решать с учетом семантики и возможностей библиотечных компонентов и связей между ними.
  Кроме того, одни представления могут визуально включать другие, а также элементы управления, с которыми связаны обработчики. Эта визуальная связь часто должна быть поддержана, например, возможностью переключения фокуса пользовательского ввода между отдельными элементами.
  Необходимо внимательно спроектировать (насколько это возможно, с учетом ограничений платформы и библиотек визуальных компонентов) стратегии обработки событий, особенно таких, в которых могут быть одновременно заинтересованы несколько компонентов, присутствующих на экране.
- Возможно, потребуется сделать систему еще более переносимой за счет отделения ее компонентов от конкретных библиотек и платформ. Для этого нужно разработать собственный набор абстрактных визуальных компонентов.

Следствия применения образца.

Достоинства.

- Возможность иметь несколько представлений одних данных, обновляемых по результатам воздействий пользователя на одно из них. Все такие представления синхронизированы, их показания соответствуют друг другу.
- Поддержка подключаемых и динамически изменяемых представлений и обработчиков.
- Возможность изменения стилей пользовательского интерфейса во время работы.
- Возможность построения каркаса (библиотек визуальных компонентов) для разработки многих интерактивных приложений.

Недостатки.

- Возрастание сложности разработки.
- Потери в производительности из-за необходимости обработки запросов пользователей сначала в обработчиках, затем в моделях, а затем во всех обновляемых компонентах.
- Если не оптимизировать производимые обновления аккуратно, чаще всего в ходе работы происходит много ненужных вызовов операций, обновляющих представления и обработчики.
- Представления и обработчики связаны очень тесно, из-за чего эти компоненты почти никогда нельзя переиспользовать по отдельности.
- И представления, и обработчики достаточно тяжело использовать без соответствующей им модели.
- Представления и обработчики наверняка потребуют изменений при их переносе на другую платформу или в другую библиотеку элементов графического интерфейса пользователя.
- Данный образец тяжело использовать в большинстве средств разработки GUI, поскольку они чаще всего определяют собственные стратегии обработки событий и стандартные обработчики для многих событий, например, для нажатия правой кнопки мыши.

Примеры. Впервые этот архитектурный стиль был использован при проектировании библиотеки разработки пользовательского интерфейса для языка Smalltalk [1]. С тех пор создатели множества подобных каркасов и библиотек используют те же принципы.

Еще один пример библиотеки для разработки пользовательского интерфейса, построенного на основе данного стиля — библиотека MFC (Microsoft Foundation Classes) от Microsoft. В ней используется более простой вариант стиля — с объединенными представлениями и обработчиками. Такая схема получила название документ-представление (Document-View): документы соответствуют моделям, представления объединяют функции представлений и обработчиков.

Такое объединение часто имеет смысл, поскольку представления и обработчики тесно связаны и практически не могут использоваться друг без друга. Их разделение на отдельные компоненты должно обосновываться серьезными причинами.

Последний пример использования этого стиля — архитектура современных Web-приложений, т.е. бизнес-приложений с пользовательским интерфейсом на основе HTML и связью между отдельными элементами, построенной на базе основных протоколов Интернет. В ней роль модели играют компоненты, реализующие бизнес-логику и хранение данных, а роль представлений и обработчиков исполняется HTML-страничками и HTML-формами, статичными или динамически генерируемыми. Далее в этом курсе построение таких приложений будет рассматриваться более детально, поэтому образец «данные-представление-обработка» имеет большое значение для дальнейшего изложения.

Образцы проектирования

Образцы проектирования в узком смысле являются типовыми проектными решениями, позволяющими удовлетворить часто встречающиеся требования к гибкости приложений и реализовать возможности их расширения за счет специальных форм организации классов и их связей.

Далее мы в деталях рассмотрим образец проектирования «подписчик». О другом примере, «адаптере», было рассказано в начале предыдущей лекции.

Подписчик

Название. Подписчик (subscriber) или подписчик-издатель (publisher-subscriber). Известен также под названиями «наблюдатель» (observer), «слушатель» (listener) или «подчиненные» (dependents).

Назначение. Реализация системы, в которой нужно поддерживать согласованными состояния большого числа объектов. Чаще всего при этом достаточно много компонентов зависит от небольшого набора данных. Можно было бы связать их явно, введя обращения ко всем компонентам, которые должны знать об изменениях, при каждом внесении изменений, но полученная система станет очень жесткой. Добавление нового компонента потребует сделать изменения во всех местах, от которых он зависит. Предлагаемое в рамках данного образца решение основано на гибкой связи между субъектом (от которого зависят другие компоненты) и этими зависимыми компонентами, называемыми *подписчиками*. Здесь нужно принимать во внимание следующие факторы.

Действующие силы.

- Об изменениях в состоянии некоторого компонента должны узнавать один или несколько других компонентов.
- Количество и конкретный вид компонентов, которые должны оповещаться об этих изменениях, заранее не известны или могут быть изменены во время работы приложения.
- Проведение зависимыми компонентами время от времени явных запросов о произошедших изменениях неэффективно.
- Источник информации (субъект или издатель) не должен быть тесно связан со своими подписчиками или зависим от них.

Решение. Компонент, от которого зависят другие, берет на себя функции *издателя*. Он предоставляет интерфейс для регистрации компонентов-*подписчиков*, заинтересованных в информации о его изменениях, и хранит множество зарегистрированных подписчиков. При изменениях он оповещает их с помощью вызова предназначенного для этого метода `update()`.

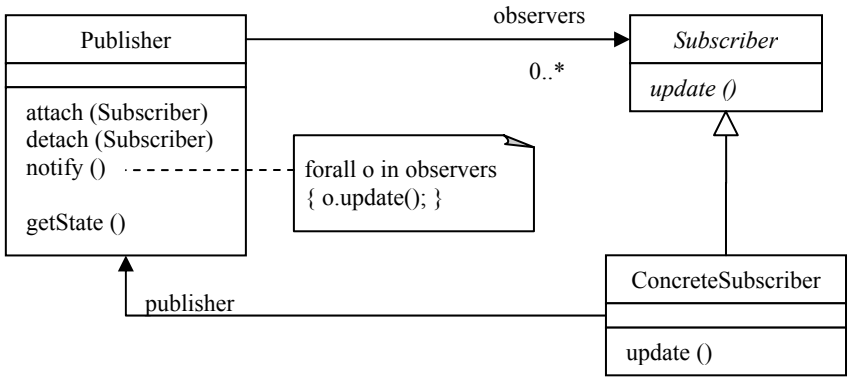


Рисунок 54. Структура классов подписчиков и издателя.

Структура. Основными компонентами являются *издатель* (или *субъект*) и *подписчики* (или *наблюдатели*). Подписчики реализуют общий интерфейс, у которого имеется метод

`update()` для оповещения подписчика о том, что в состоянии издателя произошли изменения. Издатель хранит множество подписчиков, позволяет регистрировать или удалять их из этого набора. При возникновении изменений он оповещает все элементы этого множества при помощи метода `update()`.

Динамика. Можно использовать два вида обновления подписчиков: издатель сам может сообщать им о том, какие именно изменения произошли (схема проталкивания, push model), или после получения уведомления подписчик сам обращается к издателю, чтобы узнать, что именно изменилось (схема вытягивания, pull model). Вторая схема значительно более гибкая, она позволяет подписчикам получать только необходимую им информацию, в то время как согласно первой схеме каждый подписчик получает всю информацию о произошедших изменениях.

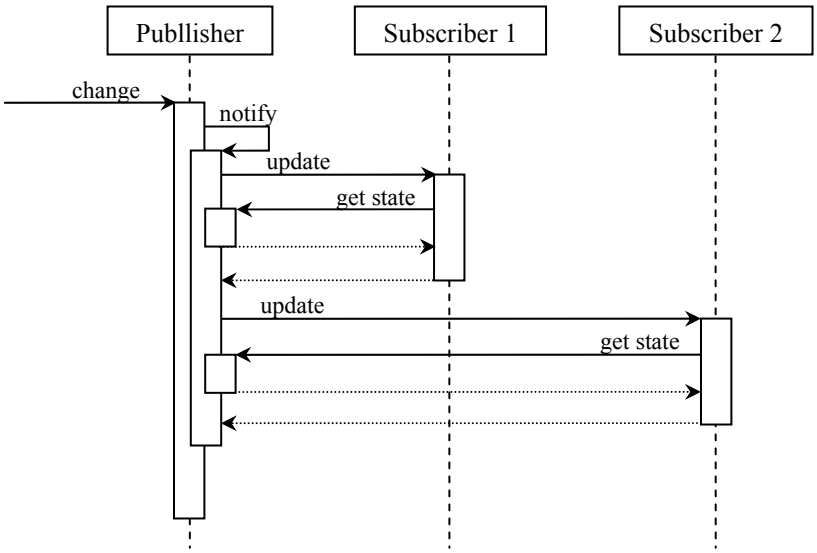


Рисунок 55. Сценарий оповещения об изменениях по схеме вытягивания.

Реализация. Основные шаги реализации следующие.

- Определить протокол обновления — будет ли использоваться простой метод `update()` или в качестве его параметров нужно будет передавать изменившийся объект-издатель и данные произошедшего изменения.
- Определить схему обновления одного подписчика: на основе проталкивания или на основе вытягивания информации.
- Определить отображение издателей на подписчиков. Если издателей много, а подписчиков мало, то хранение множеств подписчиков для каждого издателя может быть неэффективным — для экономии памяти за счет времени поиска подписчиков можно использовать отдельную таблицу, отображающую издателей на подписчиков.
- Обеспечить гарантии целостности состояния издателя перед оповещением подписчиков.
- Обеспечить гарантии аккуратного удаления объекта-подписчика из системы — нужно, чтобы он был удален из всех списков оповещений.
- Если семантика обновлений сложна, например, если подписчик зависит от нескольких издателей, которые могут изменяться в рамках одной операции, то, возможно, потребуется выделить такие сложные связи в отдельный компонент, называемый *менеджером изменений (change manager)*. Такой компонент должен сам хранить отображение между издателями и подписчиками, определять стратегию проведения обновления и обновлять всех зависимых подписчиков по запросу от издателя. В качестве стратегии обновления может выступать механизм, гарантирующий подписчику получение только одного уведомления, если изменяются несколько издателей, от которых он зависит.

Следствия применения образца.

Достоинства

- Слабая связанность между издателем и подписчиками — издатель знает только, что у него есть несколько подписчиков с одинаковым интерфейсом.
- Удобным образом, не зависящим от числа участников, поддерживаются широковещательные оповещения.

Недостатки

- Низкая производительность в случае большого количества подписчиков — даже небольшое изменение требует оповестить их всех, и каждый из них будет пытаться провести обновление своего состояния.
- Неэффективное использование информации из-за необходимости оповещать всех подписчиков, даже тех, для которых выполненные изменения несущественны. Кроме того, подписчик может зависеть от нескольких издателей и не знать, какой именно издатель оповещает его об изменении. Чтобы исправить эту ситуацию, необходимо внесение изменений в протокол оповещения — предоставление дополнительной информации, как об изменившемся издателе, так и о виде изменения.

Примеры. Один из примеров мы уже видели — в рамках более широкого стиля «данные-представление-обработка» представления и обработчики являются подписчиками по отношению к модели-издателю.

Вариант этого образца с введением менеджеров изменений описан под названием «канал событий» (Event Channel) в спецификации службы оповещения о событиях в стандарте CORBA [2].

Идиомы

Идиома представляет собой типовое решение, определяющее специфическую структуризацию элементов кода на некотором языке программирования. Чаще всего это некоторый «трюк», с помощью которого можно придать программе на данном языке нужные свойства. При этом идиома может оказаться специфичной для языка и не иметь аналога в других языках. Кроме того, очень часто удачные идиомы при развитии языков программирования превращаются в новые синтаксические конструкции, делающие такую идиому ненужной.

Далее мы увидим примеры таких идиом в виде соглашений о построении кода компонентов JavaBeans, превратившихся в C# в элементы языка.

В этой лекции мы рассмотрим в качестве примера идиому «шаблонный метод» [3].

Шаблонный метод

Название. Шаблонный метод (template method).

Назначение. Фиксация общей схемы некоторого алгоритма, имеющего много вариантов, с предоставлением возможности реализовать эти варианты за счет переопределения отдельных его шагов, без изменения схемы в целом. При использовании этой идиомы нужно принимать во внимание следующие факторы.

Действующие силы.

- Инвариантные части алгоритма нужно записать один раз и переиспользовать, изменяя только варьирующиеся шаги и элементы.
- Иногда такие инвариантные части еще нужно выделить, чтобы сделать более понятным и более удобным для сопровождения код нескольких классов, реализующих близкие по назначению методы.
- Многие алгоритмы при их практическом использовании могут зависеть от большого числа факторов, изменяющихся гораздо чаще, чем общая схема такого алгоритма (вспомните принцип разделения политик и алгоритмов).

Решение. Общая схема алгоритма, которую нужно зафиксировать, помещается в абстрактный класс в виде метода, код которого реализует эту схему. В тех местах, где необходимо использовать какой-то варьирующийся элемент алгоритма, этот метод обращается к другому методу данного класса, который можно переопределить в классах-потомках.

Структура. Метод, реализующий основную схему алгоритма, называется *шаблонным методом*. Он пишется один раз в абстрактном базовом классе и не переопределяется в потомках. Методы, вызываемые им, делятся на следующие группы.

- *Конкретные операции*, реализация которых известна на момент написания метода и не должна изменяться.
- *Абстрактные операции*, которые представляют собой изменяемые части алгоритма. Они не имеют реализации и должны определяться в каждом классе-потомке в соответствии с теми вариациями, которые он вносит в базовый алгоритм.
- *Операции-перехватчики,* или *зацепки (hook operations),* которые также представляют собой изменяемые элементы алгоритма, но имеют некоторую реализацию по умолчанию, записанную в базовом классе. Эти операции могут переопределяться в классах-потомках, если представляемые ими элементы алгоритма нужно изменить по сравнению с имеющейся реализацией по умолчанию.
- *Фабричные методы (factory methods),* предназначенные для создания объектов, которые связаны с работой конкретного варианта алгоритма. Они реализуются по образцу, также называемому «фабричный метод». Суть такого метода в том, что при его вызове мы точно не знаем, объект какого конкретного класса будет создан — это зависит от текущей конфигурации системы.

Динамика. Типичный сценарий работы шаблонного метода показан на Рис. 56. Для наглядности на этой диаграмме операции, выполняемые в одном объекте, разделены между двумя виртуальными объектами: первый представляет все действия, выполняемые в рамках абстрактного класса, определяющего шаблонный метод, а второй — те действия, которые (пере)определяются в конкретном подклассе.

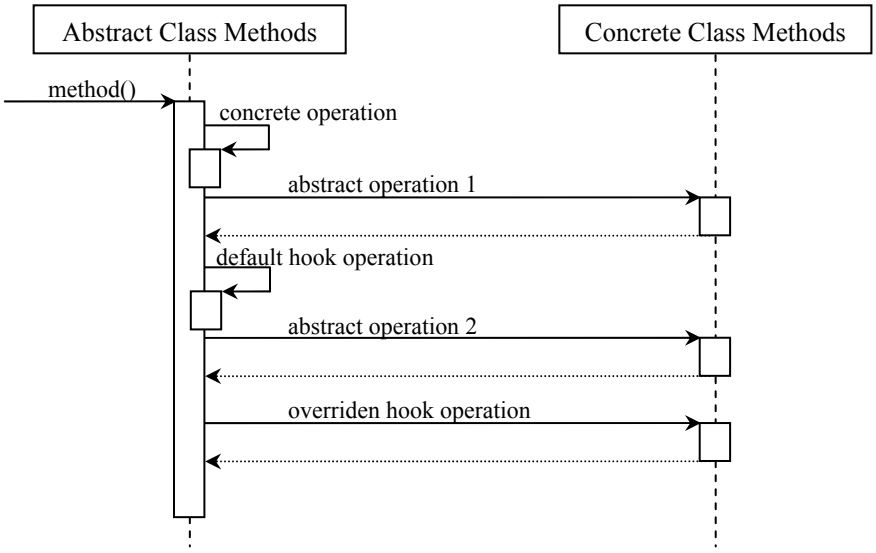


Рисунок 56. Сценарий работы шаблонного метода.

Реализация. Основные шаги реализации следующие.

- Определить основные шаги алгоритма. Определить набор данных, которыми он пользуется.
- Выделить среди шагов алгоритма те, которые зависят от конкретных политик. Определить для каждого такого шага абстрактный метод.

- Выделить среди данных, которыми пользуется алгоритм, те, чья структура зависит от политик или конкретного варианта алгоритма. Определить для порождения таких данных фабричные методы, а для операций над ними — абстрактные методы. Для их представления могут понадобиться дополнительные классы, но изменяемая часть данных должна, по возможности, находиться в абстрактном классе, определяющем шаблонный метод.
- Реализовать общую схему алгоритма в теле шаблонного метода. Выделить его элементы, используемые несколько раз или представляющие собой отдельные операции, в отдельные методы абстрактного класса.
- Определить, какие дополнительные возможности по вариации поведения алгоритма могут понадобиться. Определить для таких возможностей методы-перехватчики.
- Определить несколько наиболее часто используемых вариантов алгоритма. Реализовать их в виде подклассов определяющего шаблонный метод класса, определив в них методы, которые представляют абстрактные операции, и, если это необходимо, переопределив методы-перехватчики.

Следствия применения образца.

Достоинства

- Общая часть алгоритма реализуется явно и может быть легко переиспользована.
- Изменяемые части алгоритма могут варьироваться удобным образом, не влияя друг на друга и давая в результате различные его модификации.
- Алгоритм может быть параметризован большим набором политик, для каждой из которых возможна реализация по умолчанию.

Недостатки

- Снижение понятности кода за счет сложного потока управления.
- Снижение производительности в случае большого числа параметров, из которых в каждом конкретном варианте алгоритма используется лишь несколько.

Примеры. Шаблонные методы очень часто используются при построении библиотечных классов и каркасов приложений.

Жизненный цикл компонентов EJB реализован в виде шаблонного метода, в котором абстрактной операцией служит создание объектов данного компонента. Имеется также несколько операций-перехватчиков, позволяющих разработчику компонента специфическим образом обрабатывать переход компонента из одного состояния в другое.

Другой пример — реализация метода `start()`, запускающего отдельный поток в Java. Инструкции, выполняемые в рамках потока, помещаются в метод `run()` объекта класса `Thread` или класса, реализующего интерфейс `Runnable`. Этот метод служит операцией-перехватчиком для метода `start()` — реализация метода `run()` по умолчанию ничего не делает.

Образцы организации и образцы процессов

Образцы организации работ и образцы процессов существенно отличаются от остальных видов образцов, рассматриваемых здесь. Они фиксируют успешные практики по организации деятельности, связанной с разработкой ПО (или другими сложными видами деятельности). Такие образцы только поддерживают проектирование и разработку ПО, не давая вариантов самих проектных решений.

Образцы этого вида чаще всего извлекаются из форм организации работ и процессов, принятых в успешных компаниях-производителях ПО. Плодотворность их использования оценивается управленцами достаточно субъективно. При этом, однако, для признания некоторого вида организации работ образцом, необходимо успешное ее использование для решения одних и тех же задач в нескольких организациях.

Шаблон для описания таких образцов выглядит следующим образом.

- Название образца.
- Контекст использования, включающий основную решаемую задачу и начальные условия.
- Действующие силы — проблемы, ограничения, требования, рассуждения и идеи, под воздействием которых вырабатывается решение.
- Решение — описание используемой формы организации работ, выделяемых подзадач, выполняемых действий, используемых техник.
- Итоговый контекст — описание ожидаемых результатов использования образца, обоснование того, что его применение даст нужный эффект.

В качестве примера образца организации работ приведем процесс *инспекции программ (Fagan inspection process)*, определенный Майклом Фаганом (Michael Fagan) [4,5] (похожий процесс, называемый технической экспертизой, technical review, может быть найден в [6]).

Инспекция программ по Фагану

Название. Инспекция программ по Фагану (Fagan inspection process).

Контекст использования. Поиск ошибок на ранних этапах разработки программного обеспечения — при подготовке требований, проектировании, начальных этапах кодирования, планировании тестов.

Действующие силы.

- Усилия, необходимые для исправления ошибки, и, соответственно, ее стоимость возрастают в зависимости от этапа проекта, на котором она обнаружена. Из эмпирических данных известно, что каждый раз при переходе через границу между фазами (при использовании водопадной модели разработки) подготовка требований – проектирование – кодирование – тестирование – эксплуатация трудозатраты на исправление найденных на данном этапе ошибок возрастают в 3-5 раз. При использовании итеративных моделей затраты возрастают меньше, но не намного. Поэтому, чем раньше ошибки будут обнаруживаться, тем эффективней будет разработка в целом.
- Членам команды разработчиков надо понимать, над чем работает каждый из них и какие решения он использует. Это помогает значительно повысить эффективность собственной работы.
- Каждый артефакт — требования, проектные документы, код, тестовые планы — должен быть подготовлен на нужном уровне качества, прежде чем он будет использован для дальнейшей работы.
- Знания о найденных ошибках позволяют членам команды избегать их повторения, а также обращать больше внимания на компоненты, которые оказались наиболее подвержены ошибкам на предыдущих этапах.

Решение. Несколько членов команды разработчиков проводят тщательную инспекцию результатов работы одного из них. Такие инспекции основываются на *первичных документах*, чтобы проверить соответствие им *вторичных документов*. Первичные и вторичные документы для каждого вида деятельности в ходе разработки, для которых проведение инспекций эффективно, представлены в Таблице 8.

Выделяются следующие роли участвующих в процессе инспекции лиц.

- *Ведущий (moderator).* Он руководит проведением инспекции, руководит собраниями, фиксирует обнаруженные ошибки, назначает время проведения собраний, сроки подготовки отчетов, следит за исправлением найденных ошибок.
  В качестве ведущего должен использоваться компетентный разработчик или архитектор, не вовлеченный в проект, материалы которого инспектируются.
- *Автор (author).* Это автор первичного документа или человек, имеющий достаточно полное представление о нем. Его обязанности — подготовить рассказ об основных

положениях первичного документа и отвечать на вопросы, возникающие у членов инспектирующей команды по его поводу.

- *Интерпретатор (reader).* Это автор вторичного документа, который разработан в соответствии с первичным. Его обязанности — объяснить участникам инспекции основные идеи, лежащие в основе его интерпретации первичного документа, и отвечать на их вопросы по поводу вторичного документа.
- *Инспектор (tester).* В ходе всей инспекции он анализирует вторичный документ, проверяя его на соответствие первичному.

Вид деятельности	**Первичные документы**	**Вторичные документы**
Анализ требований	Модели предметной области, составленные заказчиками и пользователями требования	Требования к ПО
Проектирование	Требования к ПО	Описание архитектуры, проектная документация
Кодирование	Проектная документация	Код, проектная документация на отдельные компоненты
Тестирование	Требования к ПО, проектная документация, код	Тестовые планы и наборы тестовых вариантов

Таблица 8. Первичные и вторичные документы на разных этапах разработки.

Обычно рекомендуется использовать не более 4-х человек в команде, проводящей инспекцию. Расширение ее возможно в особых случаях и только за счет разработчиков, которым непосредственно придется иметь дело с инспектируемыми вторичными документами.

Сам процесс инспекции состоит из следующих шагов.

1. *Планирование (planning).*
   На этом шаге ведущий должен убедиться в том, что первичный и вторичный документы готовы к проведению инспекции — они существуют, написаны достаточно понятно, с достаточной степенью детализации.
   Кроме того, на этом шаге проводиться планирование всего хода инспекции — определяются участники, их роли, назначаются сроки проведения собраний и время, выделяемое на выполнение каждого шага.
2. *Обзор (review).*
   Проводится собрание, на котором автор представляет наиболее существенные положения первичного документа и отвечает на вопросы участников о нем.
   Первичный и вторичный документы выдаются на руки участникам инспекции для дальнейшей работы.
   Ведущий объясняет задачи данной инспекции, вопросы и моменты, на которые стоит обратить особое внимание, а также сообщает, какие ошибки были уже обнаружены в рассматриваемых документах, чтобы участники группы имели представление об их проблемных местах.
3. *Подготовка (preparation).*
   Каждый из участников тщательно изучает оба документа самостоятельно, пытаясь понять заложенные в них решения и проследить их реализацию.
   Часто на этом этапе обнаруживаются ошибки, но гораздо меньше, чем на следующем.
4. *Совместная инспекция (inspection meeting).*
   Проводится совместное собрание, на котором интерпретатор рассказывает об основных идеях и техниках, использованных во вторичном документе, а также объясняет, почему были приняты те или иные решения и почему они соответствуют первичному документу.

Участники задают вопросы и акцентируют внимание на проблемных местах. Как только ведущий по ходу собрания замечает ошибку (или кто-то обращает его внимание на нее), он сообщает о ней и убеждается, что все участники согласны с тем, что это именно ошибка, т.е. несоответствие между первичным и вторичным документами. Каждая ошибка фиксируется, описывается ее положение, она классифицируется по некоторой схеме, например, критическая (приводящая к ошибке в работе системы) или некритическая (связанная с опечатками, излишней сложностью или неудобством интерфейса и пр.).

5. *Доработка (rework).*
   В ходе доработки интерпретатор исправляет обнаруженные ошибки.
6. *Контроль результатов (follow-up).*
   Результаты доработки проверяются ведущим. Он проверяет, что все найденные ошибки были исправлены и что не было внесено новых ошибок. Если по результатам инспекции было переработано более 5% вторичного документа, следует провести полную инспекцию вновь. Иначе ведущий сам определяет, насколько документ подготовлен к дальнейшему использованию.
   Кроме того, ведущий подготавливает отчет обо всех обнаруженных ошибках для последующего использования в других инспекциях и при оценке качества результатов разработки.

Итоговый контекст. В результате проведения инспекций повышается качество проектных документов и кода, разработчики знакомятся ближе с работой друг друга и с задачами проекта в целом, углубляют понимание проблем проекта и используемых в нем решений. Кроме того, руководитель проекта получает надежные данные о качестве результатов разработки.

Руководитель должен понимать, что *результаты инспекций не должны использоваться как показатель качества работы разработчиков*, иначе все положительные эффекты от их проведения пропадают — разработчики начинают неохотно участвовать в инспекциях, скрывают детали своей работы, снисходительнее относятся к ошибкам других в расчете на взаимность и пр.

При выполнении этого условия инспекции являются эффективным средством обнаружения ошибок на ранних этапах. Статистика показывает, что они находят до 80% ошибок, обнаруживаемых за весь период разработки ПО.

Литература к Лекции 8

[1] G. E. Krasner, S. T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming, 1(3), pp. 26–49, August-September 1988. SIGS Publications, NY, USA, 1988.

[2] CORBA Event Service Specification, version 1.2. Object Management Group, October 2004. Доступен как http://www.omg.org/cgi-bin/apps/doc?formal/04-10-02.pdf.

[3] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер-ДМК, 2001.

[4] M. E. Fagan. Design and Code inspections to reduce errors in program development. IBM Systems Journal, vol. 15, No. 3, pp. 258–287, 1976.

[5] M. E. Fagan. Advances in Software Inspections. IEEE Transactions on Software Engineering, vol. 12, No. 7, pp. 744–751, July 1986.

[6] S. Ambler. Process Patterns: Building Large-Scale Systems using Object Technology. Cambridge University Press, Cambridge, MA, 1998.

[7] М. Фаулер и др. Архитектура корпоративных программных приложений. М.: Вильямс, 2004.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture. A System of Patterns. Wiley, 2002.

[9] Э. Дж. Брауде. Технология разработки программного обеспечения. СПб.: Питер, 2004.

Лекция 9. Принципы создания удобного пользовательского интерфейса

Аннотация

Рассматриваются основные факторы удобства использования ПО, а также психо-физиологические особенности человека, делающие предметы удобными и неудобными для него. Рассказывается о методике проектирования, ориентированного на удобство использования.

Ключевые слова

Удобство использования, удобство обучения, ментальная модель, метафора, наглядность, эвристики удобства использования, проектирование, ориентированное на удобство использования, модель ролей пользователей, модель задач, модель содержимого интерфейса, эвристическое инспектирование интерфейса, GOMS, тестирование удобства использования.

Текст лекции

Удобство использования программного обеспечения

Одним из важных показателей качества программного обеспечения является ***удобство его использования***. Оно описывается с помощью таких характеристик, как понятность пользовательского интерфейса, легкость обучения работе с ним, трудоемкость решения определенных задач с его помощью, производительность работы пользователя с ПО, частота появления ошибок и жалоб на неудобства. Для построения действительно удобных программ нужен учет контекста их использования, психологии пользователей, необходимости помогать начинающим пользователям и предоставлять все нужное для работы опытных. Однако самым значимым фактором является то, помогает ли данная программа решать действительно значимые для пользователей задачи.

Многие программисты имеют технический или математический склад ума. Для таких людей «понятность», «легкость обучения» представляются весьма субъективными факторами. Сами они достаточно легко воспринимают сложные вещи, если те представлены в рамках непротиворечивой системы понятий, как бы дико эта система и входящие в нее понятия ни выглядели для постороннего наблюдателя. Такие люди чаще всего изучают новое ПО при помощи документации и искренне убеждены в том, что пользователи будут разбираться с написанной ими программой тем же способом. Типичный подход программиста при разработке пользовательского интерфейса — предоставить пользователю те же рычаги и кнопки, с помощью которых программист сам хотел бы управлять своей программой.

К пользователям, у которых возникли проблемы с программой, многие программисты достаточно суровы. Любимый их ответ в такой ситуации — «RTFM!» (read this fucking manual, прочти эту чертову инструкцию). Они любят рассказывать друг другу анекдоты о «ламерах», которые не в силах понять, что файлы нужно сохранять.

Посмотрим теперь, что будет с «обычным человеком», впервые попытавшимся воспользоваться компьютером вообще и текстовым редактором в частности (как ни тяжело представить такое сейчас).

Пользователь открывает редактор, скажем, Microsoft Word, как-то набирает текст, затем печатает его на принтере и выключает компьютер. Когда он включает компьютер в следующий раз и не находит важный документ, который он набрал (вы же сами помните!), он страшно раздосадован. Что вы говорите? Надо было сохранить документ? Что значит «сохранить»? Куда? Он же набрал документ и своими глазами видел, что «тот в компьютере есть». Зачем его еще как-то «сохранять»? Ну ладно, ну хорошо, раз вы так уверяете, что нужно нажать эту кнопочку, он будет ее нажимать. Да-да, каждые 10 минут, специально для вас, он будет нажимать эту кнопочку (зачем она нужна?...). Конечно же, спустя некоторое время он забудет это сделать.

Человек «понимает» смысл и назначение вещей и действий с ними, только если они в его сознании находятся в рамках некоторой *системы связанных друг с другом понятий*. Набор текста

на компьютере больше всего напоминает набор текста на печатной машинке (и специально сделан выглядящим так в редакторах WYSIWYG), чуть менее он близок к письму. В обоих этих случаях, написав или напечатав некоторый текст на листе бумаги, мы получим достаточно долго хранящийся документ. Чтобы избавиться от него, нужно предпринимать специальные действия — смять, порвать, пролить на него кофе, выкинуть в мусорную корзину. Если такой документ пропадает без наших действий, значит кто-то (сотрудник, начальник, жена, ребенок или уборщица) взял его. Человек, только что столкнувшийся с электронными документами, воспринимает их как аналоги бумажных и ожидает от них тех же свойств.

Документы «в компьютере» не такие. У компьютера есть два вида памяти — оперативная, или временная, и постоянная. В большинстве случаев набранный в редакторе текст находится в оперативной памяти, содержимое которой пропадает при отключении питания. Чтобы текст смог «пережить» это отключение, он должен быть перемещен в постоянную память. Именно для этого служит операция «сохранить документ».

В предыдущем абзаце описана некоторая система понятий, непривычная для новичка и не доступная с помощью непосредственного созерцания компьютера и размышлений. Ее необходимо как-то передать новому пользователю, иначе он не сможет понять, зачем же сохранять уже написанные документы, ведь они и так есть. Иначе, максимум, что он сможет сделать — выучить *ритуал*, согласно которому нужно иногда нажимать на кнопку «Сохранить». Очень многие люди работают с компьютерами и другой сложной техникой с помощью ритуалов, поскольку не всегда в силах разобраться в новой для них системе понятий, в рамках которой действует эта техника. Но гораздо чаще — потому, что ее производитель и разработчики не тратят столько усилий, сколько нужно, чтобы научить этой системе каждого пользователя.

Если же подпускать пользователей к компьютеру только после прочтения необходимой документации и усвоения ее информации, редко кто из них вообще заинтересуется использованием компьютера. Они используют компьютер и программное обеспечение только как инструменты для решения своих задач (единственный вид программ, где можно хоть как-то рассчитывать на чтение документации, — игры и развлечения), и им не хочется тратить время на чтение инструкций и осмысление правил, не относящихся напрямую к их области деятельности. Почему бы этим программам не быть столь же наглядными, как молоток и отвертка — никто не станет же всерьез читать инструкцию к отвертке?

Можно поспорить с этим на том основании, что компьютер и ПО намного сложнее отвертки, с их помощью можно выполнять гораздо больше действий и сами действия гораздо сложнее. Но, с другой стороны, умеют же сейчас делать автомобили, для вождения которых нужно знать только правила движения и основные элементы управления, а если что-то идет не так — пусть разбираются автослесари. Автомобиль сравним по сложности с самыми сложными программами, а то и превосходит их. И многие водители (по крайней мере, на Западе), используют автомобили, ничего не понимая в их устройстве. Пользователи изначально не хотят читать инструкции и не будут этого делать, пока эти инструкции занимают сотни страниц, написаны непонятным и сухим языком, требуют внимательности и обдумывания, не отвечают сразу на вопросы, которые интересуют пользователя в данный момент, а также пока начальство не скажет, что инструкцию все-таки прочитать надо. Но ведь есть еще и естественная человеческая забывчивость...

Удобство обычной, «не компьютерной» модели работы с документами подтверждается тем, что Palm Pilot, первый компьютер без разделения памяти на временную и постоянную, разошелся небывалым для такого устройства тиражом — за первые два года было продано около 2-х миллионов экземпляров.

Все сказанное выше служит иллюстрацией того факта, что «простота» и «легкость обучения» все-таки не совсем субъективны, а имеют объективные составляющие, которые необходимо учитывать при разработке части программного обеспечения, предназначенной для непосредственного взаимодействия с человеком — пользовательского интерфейса. Если посмотреть внимательнее, непонимание программистами пользователей в большой степени вызвано их, программистов, собственной ленью и нежеланием задумываться над непривычными вещами.

Рисунок 57. Что это? Лень или ошибка программиста, использовавшего не то стандартное окно Windows? А что делать пользователю, получившему такое сообщение?

Психологические и физиологические факторы

Вопросы удобства использования программного обеспечения тесно связаны с аналогичными вопросами для других видов инструментов и оборудования, а также предметов быта. И решаются они на примерно той же основе, что и вопросы типа «удобна ли эта дверная ручка», «удобно ли такое табло спидометра в автомобиле», «удобен ли данный способ управления станком» и пр.

На применяемые в этой области решения огромное влияние оказывают общие законы психологии и физиологии человека, ведь вещи удобны или неудобны большей частью не из-за субъективных предпочтений, а из-за того, что строение человеческого тела и законы работы сознания помогают или мешают использовать их эффективно.

Фундаментальной основой для определения удобств и неудобств понимания человеком функционирования и способов использования различных предметов является *когнитивная психология*, которая изучает любые познавательные процессы человеческого сознания. Психология использования машин, инструментов, оборудования и предметов обихода в ходе практической деятельности человека обычно называется *инженерной психологией* [1,2]. За рубежом выделена особая наука, изучающая психологические, физиологические и анатомические аспекты взаимодействия человека и компьютера, которая так и называется — *взаимодействие человека и компьютера (Human-Computer Interaction, HCI)*.

При рассмотрении задач построения удобного ПО используют много информации из перечисленных дисциплин. Наиболее важные для разработки пользовательского интерфейса результаты этих дисциплин можно сформулировать следующим образом.

Человеку свойственно ошибаться

Обычный человек в нормальном состоянии постоянно делает ошибки разного рода. Можно сказать, что человек, в отличие от компьютера, является адаптивной аналоговой системой и успешность его «функционирования» в гораздо большей степени определяется не точностью выполнения действий и формулировки мыслей, а способностью быстро выдать хорошее приближение к нужному результату и достаточно быстро поправиться, если это необходимо.

Понаблюдайте за разговором двух хорошо знакомых людей — в нем очень часто встречаются обрывки фраз, восклицания и междометия. Далеко не каждая фраза досказывается и дослушивается до конца, и практически ни одна мысль не высказывается достаточно точно, чтобы быть однозначно понятной постороннему человеку, не включенному в контекст этого разговора. Очень часто высказываемая фраза и вовсе далека от выражаемой ею мысли, если на нее смотреть с точки зрения внешнего наблюдателя — просто большинство людей часто говорят не то, что хотели бы сказать, а то, что у них быстрее получается. Тем не менее, они понимают друг друга, а послушав их некоторое время, и третий человек начинает понимать, о чем идет речь.

Поэтому один из принципов построения удобных систем — терпимость к человеческим ошибкам, умение не замечать их, «понимая» пользователя правильно, несмотря на его не вполне корректные действия, а также наличие возможностей полного устранения последствий совсем уж неверных действий. Многими специалистами по удобству использования достаточно серьезно воспринимается радикальный тезис, состоящий в том, что пользователи не ошибаются, а лишь выполняют «действия, не направленные на достижение своих собственных целей».

К тому же сообщения об ошибках, которыми программы пугают неопытных пользователей, являются чаще всего отвлекающим от работы фактором, приводят к раздражению, а иногда — к отказу от работы с программой, которая «слишком умничает». Именно так пользователь

воспринимает указания программы, которая, явно не понимая, что же хочет человек, пытается заявлять о неверности его действий. Сообщений об ошибках в удобной программе практически не должно быть, а те, которые все-таки не удается убрать, ни в коем случае не должны формулироваться недостаточно информативно и категоричным тоном: «Неправильно! Некорректное значение!», как будто пользователь сдает экзамен. Человеку будет гораздо комфортнее, если система признается, что не может понять, что он хочет, объяснит, какие именно из введенных им данных вызывают проблемы и почему, а также предложит возможные варианты выхода из создавшейся ситуации. Но еще лучше — если ПО все понимает «правильно», даже если пользователь ошибся, и не обращает внимания на подобные мелочи.

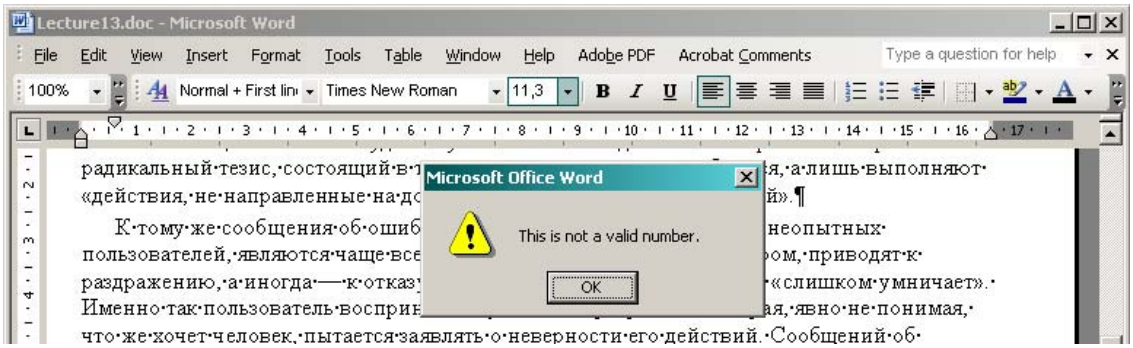


Рисунок 58. Почему 11,3 — неправильное число?

С другой стороны, действия, связанные с большим риском и невозможностью отмены, не должны быть легко доступны — иначе простота доступа в сочетании с человеческими ошибками будет часто приводить к крайне нежелательным последствиям. В самих программах таких действий немного — лишь потеря большого количества ценных данных может претендовать на такой статус, но ПО может управлять и кораблем, и самолетом, и атомной электростанцией. В подобных случаях необходимо особо выделять действия, выполнение которых может привести к тяжким последствиям. Не стоит делать запуск ракеты сравнимым по простоте с удалением файла.

Скоростные показатели деятельности человека

Время, которое человек затрачивает на различные действия, связанные с работой на компьютере, таково [3,4].

- Нажатие на клавишу клавиатуры: 0.2–1.25 с.
- Нажатие на кнопку мыши: 0.1 с.
- Перемещение курсора мыши (см. ниже о законе Фиттса): 1.0-1.5 с.

Значительное время затрачивается и на действия, которые тоже постоянно необходимо выполнять, хотя они не имеют такого прямого физического выражения, как предыдущие.

- Распознавание визуального образа: 0.1 с.
- Перевод взгляда и переключение внимания с одного объекта на другой: 0.25 с.
- Выбор из двух альтернатив (принятие простейшего решения): 1.25 с.
- Переключение внимания с мыши на клавиатуру и обратно: 0.36 с.

На данных такого рода основан GOMS — метод оценки производительности работы с интерфейсом (см. далее).

Наблюдения показывают [5], что б***о***льшую часть времени при работе человек тратит на интеллектуальную деятельность, связанную с определением целей своих действий, определением цепочки конкретных действий, которую нужно совершить, чтобы достичь поставленных целей, обнаружением всех необходимых для этого элементов, распознаванием очередного состояния системы и его оценкой с точки зрения достижения выбранных целей. Изменяя пользовательский интерфейс, можно лишь помочь пользователю быстрее найти нужные ему инструменты, выполнить нужные действия и быстрее понять, какие же результаты они дали.

Из приведенных данных можно сделать несколько важных выводов.

- Человек воспринимает и осознает информацию, а также производит действия достаточно медленно по сравнению с компьютером.
  Сама «медленность» действий человека и его восприятия, а также соотношения затрат времени на различные действия должны учитываться при проектировании интерфейсов, рассчитанных на взаимодействие с человеком.
- Глаз быстрее руки — человек гораздо быстрее узнает что-то, чем производит соответствующие действия.
  Поэтому, в частности, человеку часто удобнее работать с системами контекстной подсказки, предлагающими ему возможные варианты его дальнейшего ввода, чем набивать весь текст целиком самостоятельно.

В качестве отдельного наблюдения можно заметить, что человек гораздо быстрее узнает что-то, чем вспоминает, как оно называется. Значительно проще указать малознакомого человека на фотографии, чем вспомнить его имя и фамилию. Точно так же проще выбрать какой-то элемент из предоставленного списка, чем набрать его идентификатор или имя.

Приведенные значения для времени перемещения указателя мыши можно уточнить с помощью закона Фиттса (Fitts) [6] — $T = a + b \cdot log(D/W)$ (иногда используется формула $a + b \cdot (D/W)^{1/2}$). Здесь D обозначает расстояние, на которое переносится указатель, W — линейный размер объекта, a и b — некоторые константы, зависящие от человека и устройства мыши. Этот закон говорит, что чем ближе и больше элемент управления, тем быстрее можно попасть в него с помощью мыши.

Из этого следует, что удобнее располагать нужные пользователю элементы управления ближе к указателю мыши и делать их крупнее. Кроме того, поскольку мышь нельзя вывести за край экрана, элемент, расположенный на краю, воспринимается как «бесконечно большой» — попасть в него гораздо легче, чем в элемент аналогичного размера и на том же расстоянии от указателя, но со всех сторон окруженный «пустым» пространством.

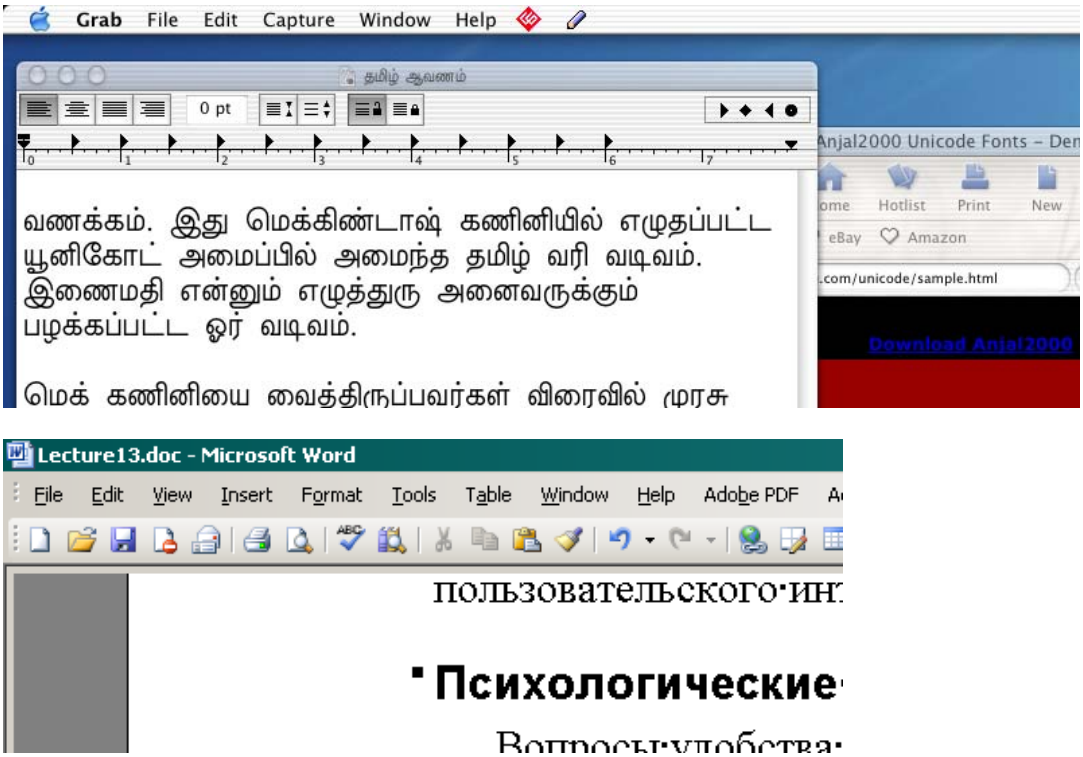


**Рисунок 59. Добираться до меню в MacOS (сверху) несколько удобнее, чем в Windows (снизу), поскольку в первом случае оно расположено вдоль края экрана.
Правда, надо привыкнуть к тому, что меню отделено от окна программы.**

Люди чаще всего промахиваются при первой попытке попасть указателем в нужное место, но большинство быстро поправляется, даже не осознавая произошедшего промаха. Если система снисходительна к промахам указателя, которые быстро исправляются, она удобнее той, которая немедленно реагирует на такие события. Например, панель задач в Windows выпадает сразу по

наведении на нее указателя мыши, поэтому при попытках нажать кнопку, расположенную где-то внизу экрана, она часто появляется «неожиданно», и пользователю приходится терять время на ожидание того, чтобы она скрылась обратно.

Внимание человека

Человеку очень тяжело долго сохранять сосредоточенность и не отвлекаться от какой-то деятельности. Чаще всего, помимо воли человека, через некоторое время ему в голову приходят разные мысли, и взгляд сам собой уползает куда-то в сторону.

Это означает, что нельзя требовать от человека длительного внимания к работе. Само понимание того, что необходимо сосредоточиться, создает у него стресс и вызывает негативные эмоции. Только в самых рискованных ситуациях, когда от его действий зависит очень многое — пике самолета, превышающий все нормы разогрев химического реактора — пользователь может сосредоточиться на значительное время, но все равно мы все предпочитаем избегать подобных обстоятельств. Такие ситуации должны явно отмечаться какими-то характерными и хорошо ощущаемыми сигналами (сирена, красный мигающий свет и пр.), и ни в коем случае эти же или похожие сигналы нельзя использовать для гораздо менее серьезных случаев, иначе пользователи будут сбиты с толку и могут их перепутать, что приводит к печальным последствиям.

В остальных же ситуациях ПО должно быть готово к постоянным переключениям внимания пользователя и ненавязчиво помогать ему восстановить фокус внимания на последних действиях, которые он выполнял, а также вспомнить их контекст — что он вообще делал, на каком шаге он сейчас находится, что еще нужно сделать и пр.

Поэтому, например, экранные формы, заполнение которых входит в одну процедуру, хорошо бы помечать так, чтобы пользователь сразу видел, сколько шагов он уже выполнил и сколько еще осталось. А текущее поле ввода стоит как-то выделять среди всех полей ввода на форме.

Для привлечения внимания пользователей к каким-то сообщениям или элементам управления нужно помнить правило: *сначала движение, затем яркий цвет, затем все остальное.* Внимание человека прежде всего акцентируется на движущихся объектах, потом — на выделенных цветом, и только потом на остальных формах выделения. Лучшим способом привлечения внимания является появление анимации, чуть менее действенным — яркие цветные окна и сообщения. Для мягкого, не режущего глаз привлечения внимания можно использовать выделение при помощи цветов мягких оттенков или изменения шрифта сообщений.

Наоборот, появление на экране или страничке ненужных цветных анимированных картинок является лучшим способом отвлечь человека от выполняемой им работы: глаза поневоле переводятся на такую картинку, и требуется психологическое усилие и некоторое время, чтобы от нее оторваться. Некоторые люди даже закрывают анимацию рукой, чтобы она не мешала сосредоточиться на полезной информации на экране!

Ярко выделенных элементов на одном экране не должно быть много, не больше, чем один-два, иначе человек перестает обращать внимание на такое выделение.

Другой фактор, связанный с вниманием пользователей, — их оценка времени выполнения системой заданных действий. Если человек ожидает, что система будет печатать документ достаточно долго, то, послав его на печать, он пойдет налить себе чаю и вернется к тому времени, когда, по его оценке, система должна закончить работу. Поэтому, прежде чем выполнять долгие операции, нужно убедиться, что все необходимые данные от пользователя получены. Иначе и пользователь, и система будут терять время, первый — ожидая на кухне, когда же система выполнит работу (которую он вроде бы запустил), а вторая — ожидая ввода дополнительных данных.

По той же причине стоит аккуратнее относиться к индикаторам степени выполнения, поскольку по их показаниям пользователи часто оценивают оставшееся время до конца выполнения задачи.

Понятность

Человек «понимает» что-либо, укладывая это в сознании в некоторую систему ассоциативных связей. Примерами механизмов, которые помогают в этом, являются следующие [4].

1. *Ментальная модель.* Этот механизм иллюстрируется примером с сохранением файлов, о котором рассказывалось выше. Пользователь понимает, как работать с системой, если ему объясняют набор сущностей, в терминах которых она функционирует, и правила работы с ними.
   Человек, один раз понявший модель действий какой-то системы, может помнить ее достаточно долго и воспринимает большинство происходящих в системе событий как вполне естественные, даже если управляющие ею правила достаточно сложны и неочевидны.
   Проблема в том, что модели работы многих систем сейчас довольно сложны, а большинство людей с трудом воспринимают сложные модели, и, начиная с некоторого уровня сложности, — не воспринимают вообще. К тому же для обучения людей такой модели нужно тратить дополнительные усилия, или же нужны дополнительные стимулы, чтобы люди прочитали документацию, в которой она объясняется.
2. *Метафора.* Сейчас довольно часто объяснения того, как работает ПО, строятся вокруг той или иной метафоры, т.е. приводится какой-то механизм или предмет из реальной жизни (или литературы), знакомый большинству пользователей, и говорится, что данное ПО работает так же. Это помогает понять основные правила его работы очень быстро, даже если их достаточно много, в отличие от долгих объяснений модели.
   Примером метафоры служит рабочий стол Windows. Для описания его работы привлекается аналогия с рабочим столом офисного служащего, на котором лежат различные документы и инструменты. Этот же пример достаточно наглядно показывает границы применимости метафоры для повышения понятности интерфейса — вряд ли с помощью этой метафоры можно распространить понимание работы Windows за пределы работы с файлами и папками, лежащими на рабочем столе.
   К сожалению, очень редко имеется или может быть придумана достаточно адекватная метафора для значительной части системы. Еще более редко такая метафора может охватить систему целиком. Иначе правила работы ПО покрываются набором слабо связанных метафор, и нужно постоянно помнить границы их применимости, а пользователи могут запутаться в этом нагромождении. Но даже в таком случае метафоры помогают, если четко ограничить зоны их действия.
3. *Наглядность (affordance)*. Свойство наглядности означает, что само представление интерфейса подсказывает, как с ним надо работать, — оно использует широко распространенные стереотипы и наглядные (не обязательно видимые глазами!) связи между элементами управления и управляемыми объектами.
   Примеры наглядности в ПО и других областях:
   a. Псевдотрехмерные выступающие кнопки как бы предлагают «Нажми меня!». При нажатии они на время «утапливаются», закрепляя таким образом наглядную ассоциативную связь с обычными кнопками.
   b. Если расположить вентили для управления конфорками газовой плиты в виде той же геометрической фигуры, как и сами конфорки (обычно по углам квадрата), соответствие между вентилями и управляемыми ими конфорками становится гораздо яснее, чем при традиционном расположении вентилей на одной прямой.

   Придумать интерфейс, имеющий свойство наглядности, очень непросто, однако если это удается, такая система почти всегда становится крупным событием на рынке.

   Гораздо чаще, к сожалению, можно найти примеры *антинаглядности* — интерфейс всем своим видом «говорит» одно, а использовать его надо совсем иначе, скажем, нарисована кнопка, а нажать ее нельзя. Антинаглядности необходимо избегать всеми силами. Например, если дверная ручка, «предлагающая» повернуть себя (ручка, за которую удобно

хвататься, с одним свободным концом, а другим закрепленная на круглой ножке), не поворачивается, чаще всего ее нечаянно отламывают.

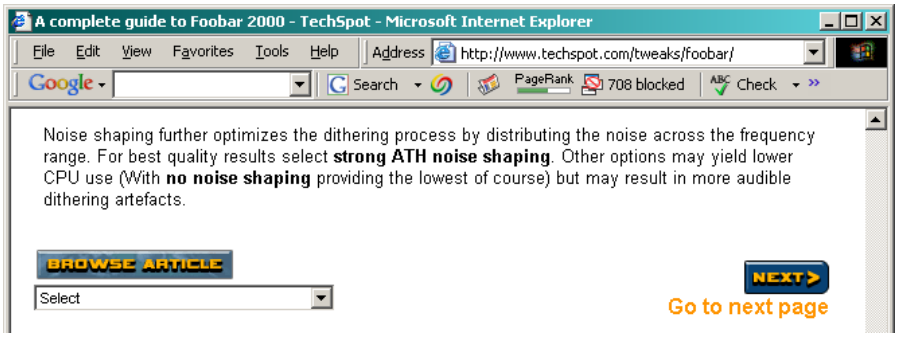


Рисунок 60. Антинаглядность. "Кнопка" Next не нажимается.

4. *Стандарт.* Человек хорошо воспринимает то, что привычно. Одним из способов «внедрения» нужных привычек является стандартизация и широкое использование стандартных интерфейсов. В последнее время стандартам внешнего вида интерфейса в целом и его отдельных элементов придается большое значение в плане обеспечения удобства использования. Этот подход действительно уменьшает затраты на обучение работе с новой программой, поскольку пользователям становится легче ориентироваться в ней. Иногда удается выработать хороший стандарт, который значительно облегчает жизнь и пользователям, и разработчикам.
   Однако значительно повысить удобство использования, особенно для опытных пользователей, только за счет применения стандартных элементов интерфейса чаще всего не удается. Это происходит потому, что стандарт, во-первых, закрепляет не все необходимые детали, во-вторых, предоставляет далеко не самые удобные, а иногда и совсем неудобные элементы. Например, стандартным средством для работы с документами, которые не вмещаются в рамки окна программы, на некоторое время стали полосы прокрутки, реализующие неудобный для человека способ перемещения по большому пространству. Сопоставьте свои действия и движения при разглядывании большой карты или схемы с тем, что приходится предпринимать при использовании для этого полос прокрутки.

Память человека

Обычно человеческую память разделяют на кратковременную и долговременную. Широко известно, что в кратковременную память помещается не более 5-9 (7±2) объектов. Долговременная же память хранит информацию в виде некоторой структуры с большим количеством связей между элементами; чтобы поместить туда нечто, нужно связать эту вещь с несколькими другими, лучше уже знакомыми.

Отсюда часто делают вывод, что 7 элементов на окне диалога — это нормально, а 10 — уже плохо. Это не совсем так. С одной стороны, довольно много людей имеет «объем» кратковременной памяти, равный 5 или 6 — для них 7 элементов уже много. С другой стороны, границы между двумя видами памяти менее резкие, чем это обычно представляется. Ограничения на число элементов в кратковременной памяти касаются несвязанных по смыслу вещей — последовательность букв ОДТЧПШСВДН запомнить с первого раза тяжело, но если знать, что это — первые буквы русских названий цифр, заканчивающихся нулем, буквы очень легко восстанавливаются. При этом используются связи между вещами (буквами и словами, самими словами и их смыслом), что более характерно для долговременной памяти.

Из сказанного можно заключить, что на формах лучше делать все же не 7, а 5, максимум 6 элементов, не связанных друг с другом, а для расширения возможностей кратковременной памяти нужно привлекать смысловые (или ассоциативные) связи между ними, например, разбить элементы, которых больше 5-ти, на меньшее число смысловых групп. Использование таких связей оказывается работоспособным механизмом и при обучении, во время которого пользователь должен запомнить довольно много.

В то же время, чем меньше нагружается память пользователей, тем меньше усилий им необходимо для работы с ПО. Предоставление нужной информации где-то на экране способно значительно облегчить им жизнь. И уж точно плохи программы, при работе с которыми пользователю приходится иногда записывать что-то на бумажке, чтобы затем использовать это в другой экранной форме или другом окне — посторонний наблюдатель сразу выразит недоумение таким способом работы, хотя сами пользователи часто привыкают к подобным вещам.

Стоит иметь в виду и постепенность запоминания — на изучение всего нового требуется время. Пользователи, не имеющие опыта работы с программой, поначалу совершают довольно много ошибок, даже пытаясь выполнять уже знакомые операции. Постепенно они вырабатывают некоторые шаблоны действий и уже не делают столько ошибок. Тем не менее, на начальном этапе им может быть необходима дополнительная помощь, чтобы не потерять интерес к изучению программы.

Разные категории пользователей

Необходимо обращать внимание на возрастные и другие особенности людей. Использование в интерфейсе программы мелких шрифтов, маленьких кнопок и пиктограмм способно сделать ее недоступной для тех, кому за 40. Использование только цветов для различения каких-то элементов интерфейса сделает его неудобным для людей с нарушениями цветового восприятия (а это около 10% всех возможных пользователей).

Конечно, если ваша программа предназначена для использования только молодежью или в организации, где могут работать только люди с нормальным цветовым зрением, можно не обращать на это внимание. Если же контекст использования программы позволяет людям с ограниченными возможностями пользоваться ею, стоит аккуратно оценивать ее удобство с их точки зрения и не пренебрегать теми элементами, которые могут им помочь.

Например, при настройках по умолчанию различные графические элементы Windows окрашены в небольшое число цветов, используются только оттенки синего и серого цветов. Это сделано для того, чтобы люди с наиболее часто встречающимися нарушениями цветового восприятия могли уверенно работать с системой. Нарушения цветового зрения в синей части спектра встречаются реже всех остальных.

Если же программа предназначена, наоборот, только для использования особой категорией людей, например, для обучения умственно отсталых детей, необходимо тщательно исследовать особенности их восприятия и разрабатывать интерфейс целиком для их удобства.

Факторы удобства использования и принципы создания удобного ПО

Основные факторы, с помощью которых можно оценить или даже измерить удобство использования программы, следующие.

- ***Адекватность интерфейса.***
  Адекватность пользовательского интерфейса программы — это его соответствие тем задачам, которые пользователи должны и хотели бы решать с ее помощью.
  Это соответствие имеет два аспекта: во-первых, все нужные пользователям задачи должны быть разрешимы (если это не так — у программы большие проблемы), во-вторых, те действия, которые пользователи выполняют чаще, должны требовать меньше усилий.
- ***Производительность работы пользователей.***
  Это количество однотипных реальных задач, которые пользователь может решить с помощью ПО за единицу времени.
- ***Скорость обучения новых пользователей.***
  Это количество задач, выполнению которых новый пользователь самостоятельно обучается за единицу времени.
  Кроме того, важным показателем является соответствие обучения частоте возникновения задач — чем чаще на практике возникает необходимость решить определенную задачу, тем быстрее пользователь должен научиться делать это.

- ***Эффективность предотвращения и преодоления ошибок пользователей.***
  Этот показатель тем лучше, чем реже пользователи ошибаются при работе с данным интерфейсом и чем меньше времени и усилий требуется для преодоления последствий уже сделанных ошибок.
  Стоит особо отметить, что предотвращение ошибок, в том числе «правильное» понимание программой действий пользователя, не вполне совпадающих с теми, которые в данном контексте хотели бы видеть ее разработчики, гораздо важнее. Предотвращенная ошибка — это ошибка не сделанная, пользователь не считает ее ошибкой и не испытывает никакого стресса по ее поводу, она не снижает его производительность.
  Большое значение имеет также риск, связанный с возникновением ошибки. Удобное ПО не должно требовать от пользователя серьезных усилий на совершение действий, ошибки в которых не слишком накладны; но если последствия ошибки катастрофичны, необходимо всячески препятствовать ее совершению. Именно поэтому запуск боевой ракеты или открытие хранилища банка не выполняются нажатием одной кнопки, которая всегда под рукой, а обставлены трудностями, вроде необходимости сделать два-три существенно разных действия, одновременно повернуть два ключа, и пр.
- ***Субъективное удовлетворение пользователей.***
  Этот фактор наиболее тяжело проанализировать, хотя измерить его довольно просто — нужно попросить группу пользователей оценить, понравилась им система или нет, по какой-то шкале (обычно используются 5-ти или 7-ми-бальные шкалы). Средний результат полученных оценок можно использовать как числовое значение этого показателя.
  Тяжелее *добиться*, чтобы интерфейс хорошо воспринимался пользователями в среднем — простое внесение исправлений, подсказанных одними пользователями, может отрицательно сказаться на оценках других. Считается [4], что субъективное удовлетворение пользователей почти всегда повышается в следующих случаях.
  - Если интерфейс программы эстетичен и элегантен, т.е. построен на немногих и близких к гармоничному (1:1.618) соотношениях между размерами отдельных элементов и расстояниями между ними, на мягких, неброских цветах и не режущих глаз их сочетаниях, небольшом количестве контрастов, слегка сглаженных углах, на аккуратном выравнивании отдельных элементов.
  - Если в работе системы не возникает долгих пауз, во время которых пользователи не знают, чем заняться. Даже если системе требуется много времени для выполнения каких-то действий, показываемые в это время картинки и предоставляемая дополнительная информация могут снизить субъективную длительность ожидания.
  - Если у пользователей, даже достаточно неопытных, не возникает стресса при работе с системой. Стресс может возникнуть по многим причинам.
    - От осознания ответственности за производимые действия и невозможности отменить их, если они вдруг окажутся неправильными.
    - От необходимости поддерживать высокий уровень внимания, запоминать какие-то вещи, чтобы использовать их позднее, или серьезно обдумывать выполнение каждого очередного действия.
    - От отсутствия контроля за поведением системы, когда она выполняет непонятные действия или сообщает о событиях, которые сам пользователь не инициировал (или не может связать их со своими предыдущими действиями).
      Другой источник потери ощущения контроля — изменчивость самого интерфейса. Если кнопки, элементы меню и пр. то появляются в определенных местах, то исчезают или становятся неактивными по неизвестным (ненаблюдаемым непосредственно) причинам, пользователь теряется. Связанный с таким поведением системы стресс объясняется также невозможностью быстро построить в сознании модель ее поведения — система становится чем-то зыбким, неустойчивым.
    - От частых, категоричных и неинформативных сообщениях об ошибках.

- Удовлетворение пользователей выше, если дается возможность настроить интерфейс программы так, как им нравится, но при этом предоставленных вариантов не чересчур много, чтобы пользователь не мог в них «заблудиться».

Специалисты по удобству использования обычно формулируют некоторый набор принципов и правил, позволяющих как оценивать удобство интерфейса, так и предлагать решения, повышающие его удобство. Ниже приведены такие правила из [3].

- **Правило доступности.**
  Система должна быть настолько понятной, чтобы пользователь, никогда раньше не видевший ее, но хорошо разбирающийся в предметной области, мог без всякого обучения начать ее использовать.
  Это правило служит некоторым идеалом, к которому надо стремиться, поскольку на практике достичь такой степени понятности почти никогда не удается. Тем не менее, все, что можно сделать для достижения этого идеала, делать нужно.
- **Правило эффективности.**
  Система не должна препятствовать эффективной работе опытных пользователей, работающих с ней долгое время.
  Очевидным примером нарушения этого правила является нацеленность системы только на новичков, например, выполнение почти всех операций с помощью мастеров (wizards), которые хорошо подходят для неопытного пользователя, ограничивая его в возможности сделать что-то не так, но неэффективны для эксперта, который и так знает, что и где ему нужно сделать.
- **Правило непрерывного развития.**
  Система должна способствовать непрерывному росту знаний, умений и навыков пользователя и приспосабливаться к его меняющемуся опыту.
  Плохие результаты приносит предоставление только базовых возможностей или оставление начинающего пользователя наедине со сложным интерфейсом, которым уверенно пользуются эксперты. Нарушение непрерывности при переходе от одного набора возможностей к другому также приносит неудобства, поскольку пользователь вынужден разбираться с добавленными возможностями в новом контексте.
  Большинство пользователей можно поместить в три группы: новичков, опытных и средних, которые уже знают больше, чем новички, и не делают столько ошибок, но еще не приобрели автоматизма при выполнении большинства операций и иногда путаются в интерфейсе. Новичкам необходима помощь в освоении новой для них системы и контроль их действий, опытным пользователям — высокая эффективность выполнения часто требующихся действий и возможность гибкого управления системой в таких ситуациях, которые встречаются реже, но способны вызвать проблемы при их неадекватной поддержке. Про средних же пользователей часто забывают, хотя подавляющее большинство пользователей ПО относится к этой категории. Им нужны достаточно высокие эффективность и гибкость вместе с возможностью быстро получать адекватную помощь по возникающим время от времени разнообразным вопросам.
- **Правило поддержки.**
  Система должна способствовать более простому и быстрому решению задач пользователя. Это означает, прежде всего, что система должна *действительно решать* задачи пользователя. Кроме того, она должна решать их *лучше, проще и быстрее*, чем имевшиеся до ее появления инструменты и методы.
- **Правило соблюдения контекста.**
  Система должна быть согласована с контекстом, в котором ей предстоит работать.
  Это правило требует от системы быть работоспособной не «вообще», а именно в том окружении, в котором ею будут пользоваться. В контекст могут входить специфика и объемы входных и выходных данных, тип и цели организаций, в которых система должна работать, уровень пользователей, зашумленность помещений и пр.

Представленные выше правила определяют общие требования, которым должен удовлетворять удобный интерфейс. Кроме них при разработке ПО необходимы некоторые подсказки, позволяющие сделать его более удобным.

Следующие *принципы* позволяют находить решения, повышающие удобство пользовательского интерфейса.

- **Принцип структуризации.**
  Пользовательский интерфейс должен быть целесообразно структурирован. Близкие по смыслу, родственные его части должны быть связаны видимым образом, а независимые — разделены; похожие элементы должны выглядеть похоже, а непохожие — различаться.
- **Принцип простоты.**
  Наиболее распространенные операции должны выполняться максимально просто. При этом должны быть видимые ссылки на более сложные процедуры.
- **Принцип видимости.**
  Все функции и данные, необходимые для решения определенной задачи, должны быть видны, когда пользователь пытается ее решить.
- **Принцип обратной связи.**
  Пользователь должен получать сообщения о действиях системы и о важных событиях внутри нее. Сообщения должны быть информативными, краткими, однозначными и написанными на языке, понятном пользователю.
- **Принцип толерантности.**
  Интерфейс должен быть гибким и терпимым к ошибкам пользователя. Ущерб от ошибок должен снижаться за счет возможности отмены и повтора действий и за счет разумной интерпретации любых разумных действий пользователя и введенных им данных. По возможности, следует избегать обязывающего взаимодействия (модальных диалогов), основанного на ограничении свободы пользователя.
- **Принцип повторного использования.**
  Следует стараться использовать многократно внутренние и внешние компоненты, обеспечивая тем самым унифицированность интерфейса и сходство между похожими его элементами.

Методы разработки удобного программного обеспечения

Одним из наиболее технологичных подходов к разработке удобного пользовательского интерфейса является *проектирование, ориентированное на использование* (usage-centered design), предложенное Л. Константайном и Л. Локвуд (L. Constantine, L. Lockwood) [3].

Основная идея этого метода — использование специальных моделей, способствующих адекватному определению набора задач, которые необходимо решать пользователям, и способов организации информации, позволяющих упростить их решение.

Список моделей, которые используются в рамках проектирования, ориентированного на использование, следующий.

- **Модель ролей.**
  Эта модель представляет собой список ролей пользователей системы. Каждая *роль* — это группа связанных задач и потребностей некоторого множества пользователей. Модель ролей может определять связи между ролями (роли могут уточнять друг друга, включать друг друга или просто быть похожими) и набор из одной-трех *центральных ролей*, на которые, в основном, и будет нацелено проектирование.
  Кроме того, каждая роль может быть снабжена профилями, указывающими различные ее характеристики по отношению к контексту использования системы. Профили могут включать следующую информацию.
  - Обязанности — требования к знаниям (о предметной области, о самой системе и пр.), которым пользователь в данной роли, скорее всего, удовлетворяет.

- Умения — уровень мастерства в работе с системой.
- Взаимодействия — типичные варианты взаимодействия пользователя в этой роли с системой, включая их частоту, регулярность, непрерывность, концентрацию, интенсивность, сложность, предсказуемость, а также управление взаимодействием (направляется ли оно пользователем, или он только реагирует на действия системы).
- Информация — источники, объем, направление передачи и сложность информации при взаимодействии с системой
- Критерии удобства — специфические критерии удобства работы для данной роли (быстрота реакции, точность указаний, удобство навигации и пр.).
- Функции — специфические функции, возможности и свойства системы, необходимые или полезные для данной роли.
- Возможные убытки от ошибок, которые может совершить человек в данной роли, риски использования различных функций.

Пример модели ролей для пользователей банкомата приведен на Рис. 61.

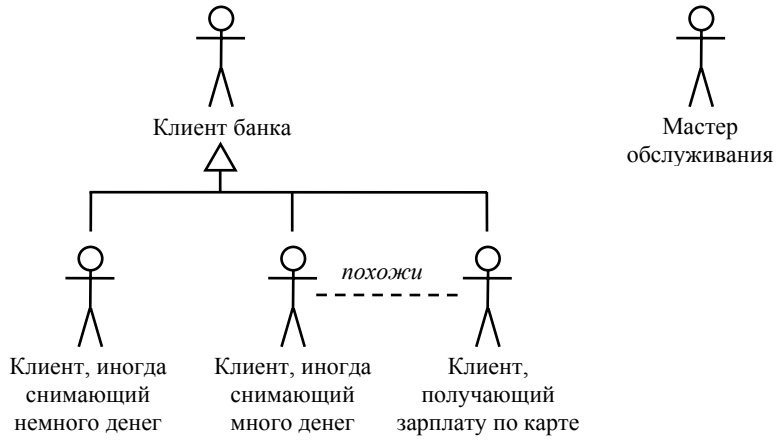


Рисунок 61. Модель ролей пользователей банкомата.

- **Модель задач.**
  Модель задач при проектировании пользовательского интерфейса строится на основе *сущностных вариантов использования (essential use cases)*. Описание сущностного варианта использования отличается от обычного тем, что в рамках его сценариев выделяются только цели и задачи пользователя, а не конкретные его действия.

Действия	**Реакции**
Вставить карту	
	Считать данные
	Запросить PIN
Ввести PIN	
	Проверить PIN
	Вывести меню
Нажать клавишу выдачи денег	
	Запросить сумму
Ввести сумму	
	Вывести сумму
	Запросить подтверждение

Задачи	**Обязательства**
Регистрация в системе	
	Проверка личности
	Предложение набора операций
Выбор операции выдачи денег	

Нажать клавишу подтверждения	
	Вернуть карту
	Выдать деньги
	Напечатать чек
	Выдать чек

	Выдача денег
	Выдача чека

Таблица 9. Описание обычного (слева) и сущностного (справа) вариантов использования.

Целью такого выделения является освобождение от неявных предположений о наличии определенных элементов интерфейсов, что помогает разрабатывать их именно для решаемых задач. Удобно описывать такие сценарии в виде двух последовательностей — устремлений пользователя (не действий, а задач которые он хочет решить) и обязательств системы в ответ на эти устремления.
Пример описания обычного и сущностного варианта использования при работе приведен в Таблице 9.

В результате модель задач представляет собой набор переработанных вариантов использования со связями между ними по обобщению, расширению и использованию. Некоторые из вариантов использования объявляются основными — без них программа потеряет значительное количество пользователей.

Всякая пользовательская роль при этом должна быть связана с одним или несколькими вариантами использования.

- **Модель содержимого.**
  Модель содержимого пользовательского интерфейса описывает набор взаимосвязанных *контекстов взаимодействия* или *рабочих пространств* (представляемых экранами, формами, окнами, диалогами, страницами и пр.) с содержащимися в них данными и возможными в их рамках действиями.
  При построении этой модели нужно определить, *что* войдет в состав интерфейса (какие данные и функции), и не решать вопрос о том, *как именно* оно будет выглядеть.
  На начальном этапе один контекст взаимодействия ставится в соответствие одному (не вспомогательному!) варианту использования или группе очень похожих вариантов, для выполнения которых понадобится один и тот же набор инструментов.
  Средства для поддержки вспомогательных расширяющих вариантов использования обычно удобно помещать в контексты расширяемых ими основных вариантов.
  Сначала устанавливается, какая информация должна находиться в заданном контексте для успешного решения задач соответствующего варианта использования, затем определяется список необходимых операций для работы с этой информацией.

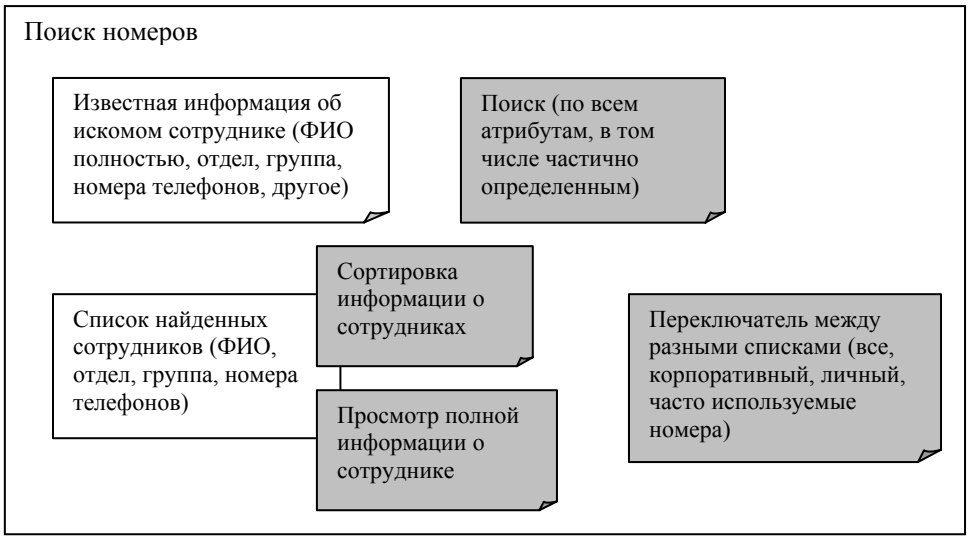


Рисунок 62. Пример модели содержимого контекста взаимодействия.

Часто при обсуждении содержимого контекста взаимодействия для его представления используют лист бумаги с наклеенными на него подписанными стикерами разных цветов (для различения информационных элементов и элементов управления). Такое представление удобно для быстрого внесения изменений по ходу обсуждения. Оно также наглядно показывает, что рассматривается лишь прототип окна или странички, а его элементы абстрактны, для них пока не предлагается конкретная форма. Его трудно принять за «почти готовый» проект окна, формы или странички, описывающий итоговые форму, расположение и цвета элементов интерфейса.
На Рис. 62 приведен пример модели содержимого окна поиска номеров телефонов программы, реализующей корпоративный телефонный справочник.

После определения набора контекстов и их информационного и функционального содержимого рисуется *карта навигации* между контекстами, показывающая возможные переходы между ними. Карта навигации объединяет различные контексты взаимодействия в рамках модели содержимого интерфейса.

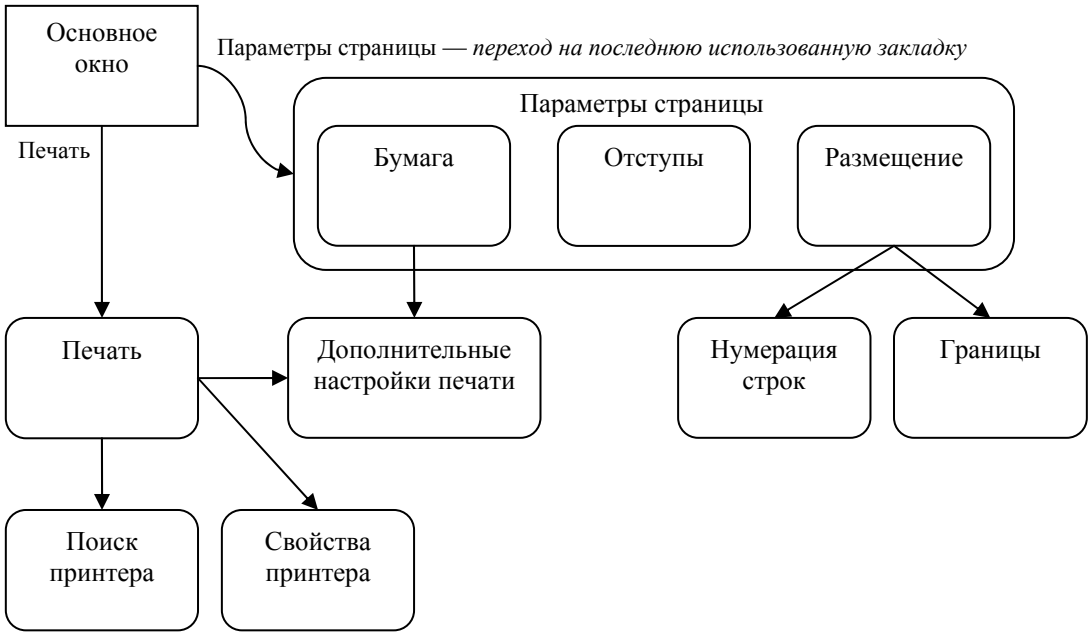


Рисунок 63. Часть карты навигации редактора Microsoft Word.

После разработки модели содержимого всякий основной вариант использования должен быть поддержан при помощи одного или нескольких контекстов взаимодействия. Чем меньше контекстов нужно использовать для выполнения одного варианта использования, тем лучше.

Перечисленные три вида моделей — ролей, задач и содержимого — являются основными. Оставшиеся два вида моделей используются только при необходимости.

- Операционная модель описывает контекст использования системы и состоит из профилей пользовательских ролей.
- Модель реализации представляет собой визуальный проект интерфейса и описание его работы.

Основные виды деятельности в рамках проектирования, ориентированного на использование, следующие.

- Совместное с пользователями определение требований к ПО, с учетом пожеланий и требований к его интерфейсу.
- Разработка модели предметной области с помощью пользователей.
- Разработка моделей ролей и задач с помощью пользователей.
- Разработка модели содержимого.

- Разработка визуального проекта интерфейса (модели реализации).
- Контроль удобства использования проекта интерфейса с участием пользователей.
- Проектирование объектной структуры ПО.
- Определение стандартов и стиля интерфейса с привлечением пользователей.
- Проектирование и разработка справочной системы и документации.
- Привязка интерфейса к контексту использования.
- Итеративная разработка архитектуры ПО.
- Итеративное конструирование ПО с постепенным введением запланированных функций.
- Контроль удобства использования готового ПО.

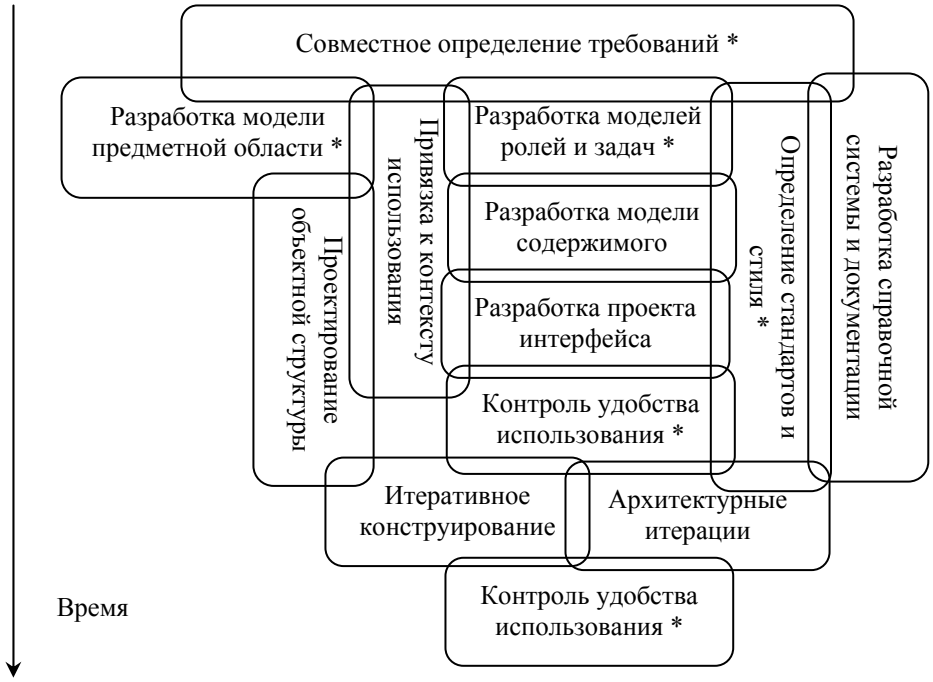


Рисунок 64. Взаимосвязи и распределение деятельностей во времени. Деятельности, в которых вовлечены пользователи, помечены звездочкой.

Эти деятельности не выполняются строго друг за другом в виде отдельных шагов. Для описания их распределения во времени используется диаграмма, изображенная на Рис. 64.

Контроль удобства программного обеспечения

Наиболее широко для контроля удобства использования ПО применяются различные виды *инспектирования*.

Эвристическое инспектирование [7] организуется как систематическая оценка различных элементов и аспектов интерфейса с точки зрения определенных эвристик. В качестве таких эвристик можно использовать изложенные выше правила и принципы построения удобных в использовании интерфейсов, или любую другую достаточно полную систему эвристик, приводимых в руководствах по удобству использования ПО.

В рамках одного сеанса инспектирования оценка интерфейса проводится несколькими специалистами, имеющими опыт в деятельности такого рода. Число оценщиков варьируется от 3-х до 5-ти. Их результаты объединяются в общую картину. В процессе инспектирования разработчики должны отвечать на вопросы, касающиеся различных аспектов как предметной области, так и работы проверяемой программы.

Оценка проводится в два этапа. На первом исследуется архитектура интерфейса в целом, на втором — отдельные контексты взаимодействия и элементы интерфейса. В целом оценка занимает 1-3 часа, после чего проводится анализ полученных результатов, во время которого оценщики описывают обнаруженные ими проблемы и предлагают способы их устранения.

При других видах инспектирования могут использоваться различные роли в группе оценщиков (оценщики, ведущий, летописец, пользователи, разработчики, эксперты в предметной области), различные шкалы серьезности обнаруженных дефектов (не более 3-4-х уровней).

В качестве метрик удобства использования применяются, например, следующие показатели (выбраны одни из самых простых и применимые в рамках описанного выше проектирования, ориентированного на удобство использования).

- ***Сущностная эффективность*** показывает степень приближения производительности пользователей при работе с данным интерфейсом к некоторой идеальной. Определяется она как процентное отношение количества действий, выполняемых пользователем в идеале — в рамках сущностного варианта использования, — к количеству действий пользователя в соответствующем сценарии работы с данным ПО.

  $$EE = \frac{Количество\ действий\ пользователя\ в\ сущностном\ варианте\ использования}{Количество\ действий\ пользователя\ в\ соответсвующем\ реальном\ сценарии} \cdot 100\%$$

  В качестве элементарных действий пользователя учитываются концептуально целостные единицы взаимодействия, такие как перечисленные ниже.
  - Ввод данных в одно поле вместе с нажатием перевода строки или табуляции.
  - Выбор поля, объекта или группы объектов (двойным или обычным) щелчком мыши или с помощью клавиатуры.
  - Переход от мыши к клавиатуре или обратно.
  - Выполнение действия (двойным или обычным) щелчком мыши на каком-либо объекте, с помощью меню или клавиатуры.
  - Перетаскивание объекта.

  Сущностная эффективность интерфейса, предназначенного для решения многих задач, определяется как сумма произведений сущностных эффективностей выполнения отдельных задач на частоты их выполнения.

  $$EE = \sum_i p_i * EE_i$$

- ***Согласованность задач*** показывает соответствие между частотами выполнения задач и скоростью их выполнения в данном интерфейсе. Вычисляется она следующим образом.
  - Задачи располагаются в порядке убывания частоты их возникновения на практике.
  - Оценивается количество действий пользователя, необходимое для выполнения каждой задачи.
  - Вычисляется индекс согласованности: для каждой пары задач, если порядок в этой паре в соответствии с трудоемкостью их выполнения совпадает с их порядком по частоте использования, к индексу прибавляется 1, иначе из индекса вычитается 1.
  - Итоговая согласованность задач оценивается как процентное отношение индекса согласованности к общему количеству различных пар задач, т.е. к n(n-1)/2, где n — число различных задач.

  Значение этой метрики колеблется от -100% (полная несогласованность) через 0% (отсутствие корреляции) до 100% (полная согласованность).

Для контроля эффективности работы пользователя с данным интерфейсом часто используется метод количественной оценки, основанный на выделении целей пользователя, операторов, методов и правил их выбора, в качестве названия которого используется аббревиатура ***GOMS*** (Goals, Operators, Methods, and Selection Rules) [8,9].

Этот метод применим для оценки эффективности работы только достаточно опытных пользователей и не учитывает возникающих по ходу работы ошибок. Кроме того, он опирается на оценки времени реакции системы, которые очень сложно получить для некоторых видов ПО, например, Web-приложений.

Основан GOMS на правилах разбиения задач на отдельные действия пользователя и на таблице заранее определенных длительностей выполнения этих действий. В качестве таких действий рассматриваются следующие.

- Нажатие на любую клавишу на клавиатуре, оценивается в 0.28 с.
- Нажатие на кнопку мыши, оценивается в 0.1 с.
- Перемещение указателя мыши, оценивается в 1.1 с.
- Переход от использования клавиатуры к мыши или обратно, оценивается в 0.4 с.
- Выбор очередного действия, оценивается в 1.2 с.
  Обычно считается, что выбор совершается при каждом переходе фокуса действий пользователя от одного элемента интерфейса к другому.
- Время реакции системы, оценивается в зависимости от имеющихся данных, как минимум в 0.1 с.
  Время реакции системы при выборе пункта меню или элемента раскрывающегося списка обычно не учитывается, но учитывается время открытия окон.

Помимо перечисленных оценочных методов используется и *тестирование удобства использования*, но, по сравнению с ними, оно может применяться только на поздних этапах разработки и, обнаруживая отдельные проблемы, не дает указаний на возможные исправления или более важные недостатки проекта в целом.

Тестирование проводится обычно в отдельном помещении, в котором пользователь может целиком сосредоточиться на работе с программой. Кроме того, все действия пользователя, ответные реакции системы и реакции пользователя на то, что он видит, протоколируются. Для этого удобно использовать съемку на видеокамеру со спины пользователя, так, чтобы были видны его действия и происходящее на экране. Для фиксации реакций пользователя можно установить зеркало, с помощью которого та же камера может снимать выражение его лица. Это помогает пользователю впоследствии объяснить, чем именно были вызваны его затруднения в данном месте. Кроме того, для протоколирования событий, которые видеокамера может и не зафиксировать, необходимо присутствие наблюдателей-людей, которые, однако, никак не должны влиять на действия пользователя (даже похмыкиванием, вздохами или ерзаньем на стуле, что пользователь может истолковать, часто обосновано, как какие-то намеки на «неправильность» его действий, или наоборот, одобрение).

Пользователь, участвующий в тестировании, должен чувствовать себя раскованно и понимать, что проводимое мероприятие никак не связано с оценкой его профессионализма, знаний или навыков. Это необходимо объяснять, поскольку большинство людей в такой ситуации так или иначе увязывают свои действия с их возможной оценкой окружающими, что вредит адекватности тестирования.

Об удобстве использования можно говорить еще очень долго, например, рассказать о проектировании и применении отдельных элементов интерфейса, а также об особенностях проектирования интерфейса различных видов ПО. Из-за ограниченности объема лекции мы остановимся здесь, а читателям, интересующимся данными вопросами, рекомендуем обратиться к специальной литературе по удобству использования ПО [3,4,7,10-13].

Литература к Лекции 9

[1] У. Вудсон, Д. Коновер. Справочник по инженерной психологии для инженеров и художников-конструкторов. М.: Мир, 1968.
[2] Ю. К. Стрелков. Инженерная и профессиональная психология. Доступно по ссылке http://psy.msu.ru/science/public/strelkov/index.html.
[3] Л. Константайн, Л. Локвуд. Разработка программного обеспечения. СПб.: Питер, 2004.
[4] В. В. Головач. Дизайн пользовательского интерфейса. Доступна на сайте http://www.uibook1.ru.
[5] D. O. Norman. The Design of Everyday Things. Basic Books, NY, 1988.

[6] P. M. Fitts. The Information Capacity of the Human Motor System in Controlling Amplitude of Movement. Experimental Psychology, 47, pp. 381–391, 1954.
[7] J. Nielsen. Usability Engineering. Academic Press, Boston, 1993.
[8] S. K. Card, T. P. Moran, and A. Newell. The Psychology of Human-Computer Interaction. Hillside, NJ: Lawrence Erlbaum, 1983.
[9] B. E. John, D. E. Kieras. The GOMS Family of Analysis Techniques: Tools for Design and Evaluation. CMU Technical Report, 1994. Доступно по ссылке ftp://www.eecs.umich.edu/people/kieras/GOMS/John-Kieras-TR94.pdf.
[10] Р. Дж. Торрес. Практическое руководство по проектированию и разработке пользовательского интерфейса. М.: Вильямс, 2002.
[11] А. Купер. Психбольница в руках пациентов. СПб.: Символ-Плюс, 2004.
[12] Я. Нильсен. Веб-дизайн: книга Якоба Нильсена. СПб.: Символ-Плюс, 2001.
[13] Я. Нильсен, М. Тахир. Дизайн Web-страниц. Анализ удобства и простоты использования 50 узлов. М.: Вильямс, 2002.

Лекция 10. Основные конструкции языков Java и C#

Аннотация

Рассматриваются базовые элементы технологий Java и .NET и основные конструкции языков Java и C#. Рассказывается о лексике, базовых типах, выражениях и инструкциях обоих языков, а также о правилах описания пользовательских типов.

Ключевые слова

Java, .NET, J2SE, J2EE, C#, Unicode, пакет, пространство имен, примитивный тип, выражение, инструкция, исключение, ссылочный тип, тип значений, класс, интерфейс, массив, перечислимый тип, делегатный тип.

Текст лекции

Платформы Java и .NET

На данный момент наиболее активно развиваются две конкурирующие линии технологий создания ПО на основе компонентов — технологии Java и .NET. В этой и следующих лекциях мы рассмотрим несколько элементов этих технологий, являющихся ключевыми в создании широко востребованного в настоящее время и достаточно сложного вида приложений. Это Web-приложения, т.е. распределенное программное обеспечение, использующее базовую инфраструктуру Интернет для связи между различными своими компонентами, а стандартные инструменты для навигации по Web — браузеры — как основу для своего пользовательского интерфейса.

Технологии Java представляют собой набор стандартов, инструментов и библиотек, предназначенных для разработки приложений разных типов и связанных друг с другом использованием языка программирования Java. Торговая марка Java принадлежит компании Sun Microsystems, и эта компания во многом определяет развитие технологий Java, но в нем активно участвуют и другие игроки — IBM, Intel, Oracle, Hewlett-Packard, SAP, Bea и пр.

В этот набор входят следующие основные элементы.

- Платформа *Java Platform Standard Edition (J2SE)* [1].
  Предназначена для разработки обычных, в основном, однопользовательских приложений.
- Платформа *Java Platform Enterprise Edition (J2EE)* [2].
  Предназначена для разработки распределенных Web-приложений уровня предприятия.
- Платформа *Java Platform Micro Edition (J2ME)* [3].
  Предназначена для разработки встроенных приложений, работающих на ограниченных ресурсах, в основном, в мобильных телефонах и компьютеризированных бытовых устройствах.
- Платформа *Java Card* [4].
  Предназначена для разработки ПО, управляющего функционированием цифровых карт. Ресурсы, имеющиеся в распоряжении такого ПО, ограничены в наибольшей степени.

С некоторыми оговорками можно считать, что J2ME является подмножеством J2SE, а та, в свою очередь, подмножеством J2EE. Java Card представляет собой, по существу, особый набор средств разработки, связанный с остальными платформами только поддержкой (в сильно урезанном виде) языка Java.

Язык Java — это объектно-ориентированный язык программирования, который транслируется не непосредственно в машинно-зависимый код, а в так называемый *байт-код*, исполняемый специальным интерпретатором, *виртуальной Java машиной (Java Virtual Machine, JVM)*. Такая организация работы Java-программ позволяет им быть переносимыми без изменений и одинаково работать на разных платформах, если на этих платформах есть реализация JVM, соответствующая опубликованным спецификациям виртуальной машины.

Кроме того, интерпретация кода позволяет реализовывать различные политики безопасности для одних и тех же приложений, выполняемых в разных средах, — к каким ресурсам (файлам, устройствам и пр.) приложение может иметь доступ, а к каким нет, можно определять при запуске виртуальной машины. Таким способом можно обеспечить запускаемое пользователем вручную приложение (за вред, причиненный которым, будет отвечать этот пользователь) б***о***льшими правами, чем аплет, загруженный автоматически с какого-то сайта в Интернет.

Режим интерпретации приводит обычно к более низкой производительности программ по сравнению с программами, оттранслированными в машинно-специфический код. Для преодоления этой проблемы JVM может работать в режиме *динамической компиляции (just-in-time-compilation, JIT)*, в котором байт-код на лету компилируется в машинно-зависимый, а часто исполняемые участки кода подвергаются дополнительной оптимизации.

В настоящем курсе мы рассмотрим ряд элементов платформ J2EE и J2SE, имеющих большое значение для разработки Web-приложений. Читателей, интересующихся деталями устройства и использования J2ME и Java Card, мы отсылаем к документации по этим платформам [3] и [4].

.NET [5] представляет собой похожий набор стандартов, инструментов и библиотек, но разработка приложений в рамках .NET возможна с использованием различных языков программирования. Основой .NET являются виртуальная машина для *промежуточного языка (Intermediate Language, IL,* иногда встречается сокращение *MSIL, Microsoft IL)*, в который транслируются все .NET программы, также называемая *общей средой выполнения (Common Language Runtime, CLR)*, и общая библиотека классов (.NET Framework class library), доступная из всех .NET приложений.

Промежуточный язык является полноценным языком программирования, но он не предназначен для использования людьми. Разработка в рамках .NET ведется на одном из языков, для которых имеется транслятор в промежуточный язык —Visual Basic.NET, C++, C#, Java (транслятор Java в .NET называется J#, и он не обеспечивает одинаковой работы программ на Java, оттранслированных в .NET, и выполняемых на JVM) и пр. Однако разные языки достаточно сильно отличаются друг от друга, и чтобы гарантировать возможность из одного языка работать с компонентами, написанными на другом языке, необходимо при разработке этих компонентов придерживаться *общих правил (Common Language Specifications, CLS)*, определяющих, какими конструкциями можно пользоваться во всех .NET языках без потери возможности взаимодействия между результатами. Наиболее близок к промежуточному языку C# — этот язык был специально разработан вместе с платформой .NET.

Некоторым отличием от Java является то, что код на промежуточном языке в .NET не интерпретируется, а всегда выполняется в режиме динамической компиляции (JIT).

Компания Microsoft инициировала разработку платформы .NET и принятие стандартов, описывающих ее отдельные элементы (к сожалению, пока не все), и она же является основным поставщиков реализаций этой платформы и инструментов разработки. Благодаря наличию стандартов возможна независимая реализация .NET (например, такая реализация разработана в рамках проекта Mono [6]), но, в силу молодости платформы и опасений по поводу монопольного влияния Microsoft на ее дальнейшее развитие, реализации .NET не от Microsoft используются достаточно редко.

Прежде чем перейти к более детальному рассмотрению компонентных технологий Java и .NET, ознакомимся с языками, на которых создаются компоненты в их рамках.

Для Java технологий базовым языком является Java, а при изучении правил построения компонентов для .NET мы будем использовать язык C#. Он наиболее удобен при работе в этой среде и наиболее похож на Java.

Оба этих языка сейчас активно развиваются — в сентябре 2004 года вышла версия 1.5, она же была объявлена версией 5 платформы J2SE и языка Java. При переходе между версиями 1.4 и 1.5 Java претерпела наиболее серьезные изменения за всю свою историю (достаточно полное описание этих изменений стало доступно только в середине 2005 года). Во время написания данной лекции готовится выход J2EE 1.5, пока доступной только в виде предварительных спецификаций.

Существенное обновление C# также должно произойти в версии 2.0, выходящей в ноябре 2005 года. Поэтому мы сопоставляем характеристики уже имеющейся на момент написания версии Java 5 с готовящейся к выходу версией C# 2.0. Это представляется обоснованным еще и потому, что эти версии двух языков достаточно близки по набору поддерживаемых ими конструкций.

Данная лекция дает лишь базовую информацию о языках Java и C#, которая достаточна для понимания приводимого далее кода компонентов и общих правил, регламентирующих их разработку и может служить основой для дальнейшего их изучения. Оба языка достаточно сложны, и всем их деталям просто невозможно уделить внимание в рамках двух лекций. Для более глубокого изучения этих языков (особенно необходимого при разработке инструментов для работы с ними) рекомендуется обратиться к соответствующим стандартам [7] и [8] (ссылки приведены на последние версии стандартов на момент написания этой лекции, кроме того некоторые элементы C# 2.0 не вошли в [8], они описываются согласно [9]).

Оба языка имеют мощные выразительные возможности объектно-ориентированных языков последнего поколения, поддерживающих автоматическое управление памятью и работу в многопоточном режиме. Они весьма похожи, но имеют большое число мелких отличий в деталях. Наиболее существенны для построения программ различия, касающиеся наличия в C# невиртуальных методов, возможности объявления и использования пользовательских типов значений и делегатных типов в C# и возможности передачи значений параметров в C# по ссылке. Обсуждение и одновременно сравнение характеристик языков мы будем проводить по следующему плану.

1. Лексика
2. Общая структура программ
3. Базовые типы и операции над ними
4. Инструкции и выражения
5. Пользовательские типы
6. Средства создания многопоточных программ

Общие черты Java и C# описываются далее обычным текстом, а особенности — в колонках.

В левой колонке будут описываться особенности Java.	В правой колонке будут описываться особенности C#.

Лексика

Программы на обоих рассматриваемых языках, C# и Java, могут быть написаны с использованием набора символов Unicode, каждый символ в котором представляется при помощи 16-ти бит. Поскольку последние версии стандарта Unicode [10] определяют более широкое множество символов, включая символы от U+10000 до U+10FFFF (т.е. имеющие коды от 2^{16} до $2^{20}+2^{16}-1$), такие символы представляются в кодировке UTF-16, т.е. двумя 16-битными символами, первый в интервале U+D800–U+DBFF, второй — U+DC00–U+DFFF.

Лексически программы состоят из разделителей строк (символы возврата каретки, перевода строки или их комбинация), комментариев, пустых символов (пробелы и табуляции), идентификаторов, ключевых слов, литералов, операторов и разделительных символов.

В обоих языках можно использовать как однострочный комментарий, начинающийся с символов `//` и продолжающийся до конца строки, так и выделительный, открывающийся символами `/*` и заканчивающийся при помощи `*/`.

Идентификаторы должны начинаться с буквы (символа, который считается буквой в Unicode, или символа _) и продолжаться буквами или цифрами. В качестве символа идентификатора может использоваться последовательность `\uxxxx`, где `x` — символы `0-9`, `a-f` или `A-F`, обозначающая символ Unicode с шестнадцатеричным кодом `xxxx`. Корректными идентификаторами являются, например, `myIdentifier123`, `αρετη_μυσ`, `идентификатор765` (если последние два представлены в Unicode). Ключевые слова устроены также (без возможности использовать Unicode-

последовательности в C#), но используются для построения деклараций, инструкций и выражений языка или для обозначения специальных констант.

В Java ключевые слова не могут использоваться в качестве идентификаторов.

Добавив в начало ключевого слова символ `@`, в C# можно получить идентификатор, посимвольно совпадающий с этим ключевым словом. Этот механизм используется для обращения к элементам библиотек .NET, написанным на других языках, в которых могут использоваться такие идентификаторы — `@`**`class`**, `@`**`int`**, `@`**`public`** и пр.

Можно получать идентификаторы, добавляя `@` и в начало идентификатора, но делать это не рекомендуется стандартом языка.

Другой способ получить идентификатор, совпадающий по символам с ключевым словом, — использовать в нем Unicode-последовательность вместо соответствующего символа ASCII.

Кроме того, в C# есть специальные идентификаторы, которые только в некотором контексте используются в качестве ключевых слов. Таковы **`add`**, **`alias`**, **`get`**, **`global`**, **`partial`**, **`remove`**, **`set`**, **`value`**, **`where`**, **`yield`**.

В обоих языках имеется литерал **`null`** для обозначения пустой ссылки на объект, булевские литералы **`true`** и **`false`**, символьные и строковые литералы, целочисленные литералы и литералы, представляющие числа с плавающей точкой.

Символьный литерал, обозначающий отдельный символ, представляется как этот символ, заключенный в одинарные кавычки (или апострофы). Так, например, можно представить символы `'a'`, `'#'`, `'Ы'`. Чтобы представить символы одинарной кавычки, обратного слэша и некоторые другие используются так называемые ESC-последовательности, начинающиеся с обратного слэша — `'\''` (одинарная кавычка), `'\\'` (обратный слэш), `'\"'` (обычная кавычка), `'\n'` (перевод строки), `'\r'` (возврат каретки), `'\t'` (табуляция). Внутри одинарных кавычек можно использовать и Unicode-последовательности, но осторожно — если попытаться представить так, например, символ перевода строки `\u000a`, то, поскольку такие последовательности заменяются соответствующими символами в самом начале лексического анализа, кавычки будут разделены переводом строки, что вызовет ошибку.

В Java можно строить символьные литералы в виде восьмеричных ESC-последовательностей из не более чем трех цифр — `'\010'`, `'\142'`, `'\377'`. Такая последовательность может представлять только символы из интервала U+0000–U+00FF.

В C# можно использовать шестнадцатеричные ESC-последовательности из не более чем четырех цифр для построения символьных литералов. Такая последовательность обозначает Unicode-символ с соответствующим кодом.

Строковые литералы представляются последовательностями символов (за исключением переводов строк) в кавычках. В качестве символов могут использоваться и ESC-последовательности, разрешенные в данном языке. Строковый литерал может быть разбит на несколько частей, между которыми стоят знаки `+`. Значения литералов `"Hello, world"` и `"Hello," + " world"` совпадают.

В C# можно строить *буквальные строковые литералы (verbatim string literals)*, в которых ESC-последовательности и Unicode-

последовательности не преобразуются в их значения. Для этого нужно перед открывающей кавычкой поставить знак `@`. В такой строке могут встречаться любые символы, кроме `"`. Чтобы поместить туда и кавычку, надо повторить ее два раза.

Например, `"Hello \t world"` отличается от `@"Hello \t world"`, а `"\""` совпадает с `@""""`.

Целочисленные литералы представляют собой последовательности цифр, быть может, со знаком — `1234`, `-7654`. Имеются обычные десятичные литералы и шестнадцатеричные, начинающиеся с `0x` или `0X`. По умолчанию целочисленные литералы относятся к типу **`int`**. Целочисленные литералы, имеющие тип длинного целого числа **`long`**, оканчиваются на букву `l` или `L`.

В Java имеются также восьмеричные целочисленные литералы, которые начинаются с цифры `0`.

В C#, в отличие от Java, имеются беззнаковые целочисленные типы **`uint`** и **`ulong`**. Литералы этих типов оканчиваются на буквы `u` или `U`, и на любую комбинацию букв `u`/`U` и `l`/`L`, соответственно.

Литералы, представляющие числа с плавающей точкой, могут быть представлены в обычной записи (`3.1415926`) или экспоненциальной (`314.15926e-2` и `0.31415926e1`). По умолчанию такие литералы относятся к типу **`double`**, и могут иметь в конце символ `d` или `D`. Литералы типа `float` оканчиваются буквами `f` или `F`.

В Java литералы с плавающей точкой могут иметь шестнадцатеричное представление с двоичной экспонентой. При этом литерал начинается с `0x` или `0X`, экспонента должна быть обязательно и должна начинаться с буквы `p` или `P`.

В C# есть тип с плавающей точкой **`decimal`** для более точного представления чисел при финансовых расчетах. Литералы этого типа оканчиваются на букву `m` или `M`.

Операторы и разделители обоих языков:

```
(     )     {     }     [     ]     ;     ,     .     :     ?     ~
=     <     >     !     +     -     *     /     %     &     |     ^
==    <=    >=    !=    +=    -=    *=    /=    %=    &=    |=    ^=
&&    ||    ++    --    <<    >>    <<=   >>=
```

Дополнительные операторы Java:

```
>>>   >>>=
```

Дополнительные операторы C#:

```
->    ::    ??
```

В C#, помимо ранее перечисленных лексических конструкций, имеются директивы препроцессора, служащие для управления компиляцией. Директивы препроцессора не могут находиться внутри кавычек, начинаются со знака `#` и пишутся в отдельной строке, эта же строка может заканчиваться комментарием.

Директивы `#`**`define`** и `#`**`undef`** служат для того, чтобы определять и удалять опции для условной компиляции (такая опция может быть произвольным идентификатором, отличным от **`true`** и **`false`**).

Директивы `#`**`if`**, `#`**`elif`**, `#`**`else`** и `#`**`endif`** служат

для того, чтобы вставлять в код и выбрасывать из него некоторые части в зависимости от декларированных с помощью предыдущих директив опций. В качестве условий, проверяемых директивами `#if` и `#elif`, могут использоваться выражения, составленные из опций и констант `true` и `false` при помощи скобок и операций `&&`, `||`, `==`, `!=`.

Например

```
using System;
#define Debug
public class Assert
{
  public void Assert (bool x)
  {
    #if Debug
    if(!x) throw
      new Exception("Assert failed");
    #endif
  }
}
```

Директивы `#error` и `#warning` служат для генерации сообщений об ошибках и предупреждениях, аналогичных таким же сообщениям об ошибках компиляции. В качестве сообщения выдается весь текст, следующий в строке за такой директивой.

Директива `#line` служит для управления механизмом сообщений об ошибках с учетом строк. Вслед за такой директивой в той же строке может следовать число, число и имя файла в кавычках или слово `default`. В первом случае компилятор считает, что строка, следующая после строки с этой директивой, имеет указанный номер, во втором — помимо номера строки в сообщениях изменяется имя файла, в третьем компилятор переключается в режим по умолчанию, забывая об измененных номерах строк.

Директива `#pragma warning`, добавленная в C# 2.0, служит для включения или отключения предупреждений определенного вида при компиляции. Она используется в виде

```
#pragma warning disable n_1, …, n_k
#pragma warning restore n_1, …, n_k
```

где `n_1, … , n_k` — номера отключаемых/включаемых предупреждений.

Общая структура программы

Программа на любом из двух рассматриваемых языков представляет собой набор пользовательских типов данных — в основном, классов и интерфейсов, с их методами. При запуске программы выполняется определенный метод некоторого типа. В ходе работы программы

создаются объекты различных типов и выполняются их методы (операции над ними). Объектами особого типа представляются различные потоки выполнения, которые могут быть запущены параллельно.

Во избежание конфликтов по именам и для лучшей структуризации программ пользовательские типы размещаются в специальных пространствах имен, которые в Java называются *пакетами (packages)*, а в C# *пространствами имен (namespaces)*. Имена пакетов и пространств имен могут состоять из нескольких идентификаторов, разделенных точками. Из любого места можно сослаться на некоторый тип, используя его длинное имя, состоящее из имени содержащего его пространства имен или пакета, точки и имени самого типа.

В обоих случаях программный код компилируется в бинарный код, исполняемый виртуальной машиной. Правила размещения исходного кода по файлам несколько отличаются.

Код пользовательских типов Java размещается в файлах с расширением .java.	Код пользовательских типов C# размещается в файлах с расширением .cs.
При этом каждый файл относится к тому пакету, чье имя указывается в самом начале файла с помощью декларации **`package`** `mypackage;`	Декларация пространства имен начинается с конструкции **`namespace`** `mynamespace {` и заканчивается закрывающей фигурной скобкой. Все типы, описанные в этих фигурных скобках, попадают в это пространство имен. Типы, описанные вне декларации пространства имен, попадают в пространство имен с пустым именем.

При отсутствии этой декларации код такого файла попадает в пакет с пустым именем.	Пространства имен могут быть вложены в другие пространства имен. При этом следующие декларации дают эквивалентные результаты.
	`namespace` `A.B { … }`
	`namespace` `A`
	`{`
	`  `**`namespace`** `B { … }`
	`}`

В одном файле может быть описан только один общедоступный (**`public`**) пользовательский тип верхнего уровня (т.е. не вложенный в описание другого типа), причем имя этого типа должно совпадать с именем файла без расширения.	В одном файле можно декларировать много типов, относящихся к разным пространствам имен, элементы одних и тех же пространств имен могут описываться в разных файлах.
В том же файле может быть декларировано сколько угодно необщедоступных типов.	
Пользовательский тип описывается полностью в одном файле.	Пользовательский тип описывается целиком в одном файле, за исключением *частичных типов* (введены в C# 2.0), помеченных модификатором **`partial`** — их элементы можно описывать в разных файлах, и эти описания объединяются, если не противоречат друг другу.
Чтобы ссылаться на типы, декларированные в других пакетах, по их коротким именам, можно воспользоваться директивами импорта.	Чтобы ссылаться на типы, декларированные в других пространствах имен, по их коротким именам, можно воспользоваться директивами использования.
Если в начале файла после декларации пакета присутствует директива **`import`** `java.util.ArrayList;`	Директива **`using`** `System.Collections;`
то всюду в рамках этого файла можно ссылаться на тип `ArrayList` по его короткому	делает возможным ссылки с помощью короткого имени на любой тип (или вложенное

имени.

Если же присутствует директива

```
import java.util.*;
```

то в данном файле можно ссылаться на любой тип пакета `java.util` по его короткому имени.

Директива

```
import static java.lang.Math.cos;
```

(введена в Java 5) позволяет в рамках файла вызывать статический метод `cos()` класса `java.lang.Math` просто по его имени, без указания имени объемлющего типа.

Во всех файлах по умолчанию присутствует директива

```
import java.lang.*;
```

Таким образом, на типы из пакета `java.lang` можно ссылаться по их коротким именам (если, конечно, в файле не декларированы типы с такими же именами — локально декларированные типы всегда имеют преимущество перед внешними).

Файлы должны располагаться в файловой системе определенным образом.

Выделяется одна или несколько корневых директорий, которые при компиляции указываются в опции `-sourcepath` компилятора. Файлы из пакета без имени должны лежать в одной из корневых директорий. Все остальные должны находиться в поддиректориях этих корневых директорий так, чтобы имя содержащего пакета, к которому файл относится, совпадало бы с именем содержащей сам файл директории относительно включающей ее корневой (с заменой точки на разделитель имен директорий).

Результаты компиляции располагаются в файлах с расширением .class, по одному типу на файл. Хранящие их директории организуются по тому же принципу, что и исходный код, — в соответствии с именами пакетов, начиная от некоторого (возможно другого) набора корневых директорий. Указать компилятору корневую директорию, в которую нужно складывать результаты компиляции, можно с помощью опции -d.

Чтобы эти типы были доступны при компиляции других, корневые директории, содержащие соответвующие им .class файлы, должны быть указаны в опции компилятора `-classpath`.

В этой же опции могут быть указаны архивные файлы с расширением .jar, в которых много

пространство имен) пространства имен `System.Collections` в рамках кода пространства имен или типа, содержащего эту директиву или в рамках всего файла, если директива не вложена ни в какое пространство имен.

Можно определять новые имена (синонимы или алиасы) для декларированных извне типов и пространств имен. Например, директива

```
using Z=System.Collections.ArrayList;
```

позволяет затем ссылаться на тип `System.Collections.ArrayList` по имени `Z`.

Нет никаких ограничений на именование файлов и содержащихся в них типов, а также на расположение файлов в файловой системе и имена декларированных в них пространств имен.

Результат компиляции C# программы — динамически загружаемая библиотека (с расширением .dll в системе Windows) или исполняемый файл (.exe), имеющие особую структуру. Такие библиотеки называются *сборками (assembly)*.

Для того чтобы использовать типы, находящиеся в некоторой сборке с расширением .dll, достаточно указать ее файл компилятору в качестве внешней библиотеки.

.class файлов хранится в соответствии со структурой пакетов.

Входной точкой программы является метод **public static void** `main (String[])`

одного из классов. Его параметр представляет собой массив строк, передаваемых как параметры командной строки при запуске.

При этом полное имя класса, чей метод `main()` выбирается в качестве входной точки, указывается в качестве параметра виртуальной машине при запуске (параметры командной строки следуют за ним).

Входной точкой программы является метод **public static void** `Main ()`

одного из классов. Такой метод может также иметь параметр типа `string[]` (представляющий параметры командной строки, как и в Java) и/или возвращать значение типа **int**.

Класс, чей метод выбирается в качестве входной точки, указывается в качестве стартового класса при сборке исполняемого файла. Собранный таким образом файл всегда будет запускать метод `Main()` указанного класса.

Ниже приведены программы на обоих языках, вычисляющие и печатающие на экране значение факториала неотрицательного целого числа (0! = 1, n! = 1·2·…·n), передаваемого им в качестве первого аргумента командной строки. Также приводятся командные строки для их компиляции и запуска.

В отличие от Java, параметры компилятора и способ запуска программ в C# не стандартизованы. Приведена командная строка для компилятора, входящего в состав Microsoft Visual Studio 2005 Beta 2. Предполагается, что директории, в которых находятся компиляторы, указаны в переменной окружения `$path` или `%PATH%`, и все команды выполняются в той же директории, где располагаются файлы с исходным кодом.

Компилятор Java и Java-машина располагаются в поддиректории `bin` той директории, в которую устанавливается набор для разработки Java Development Kit. Компилятор C# располагается в поддиректории `Microsoft.NET\Framework\v<номер версии установленной среды .NET>` системной директории Windows (обычно `Windows` или `WINNT`).

```
public class Counter
{
  public int factorial(int n)
  {
    if(n == 0) return 1;
    else if(n > 0)
      return n * factorial(n - 1);
    else
      throw new IllegalArgumentException(
        "Argument should be >= 0, " +
        "current value n = " + n);
  }

  public static void main(String[] args)
  {
    int n = 2;
    if(args.length > 0)
    {
      try
      {
        n = Integer.parseInt(args[0]);
      }
      catch(NumberFormatException e)
      {
        n = 2;
      }
    }
```

```
using System;

public class Counter
{
  public int Factorial(int n)
  {
    if (n == 0) return 1;
    else if (n > 0)
      return n * Factorial(n - 1);
    else
      throw new ArgumentException(
        "Argument should be >= 0, " +
        "current value n = " + n);
  }

  public static void Main(string[] args)
  {
    int n = 2;
    if (args.Length > 0)
    {
      try
      {
        n = Int32.Parse(args[0]);
      }
      catch (Exception)
      {
        n = 2;
      }
    }
```

```
    if(n < 0) n = 2;
    Counter f = new Counter();
    System.out.println(f.factorial(n));
  }
}
```

Компиляция

```
javac Counter.java
```

Выполнение

```
java Counter 5
```

Результат

```
120
```

```
    if (n < 0) n = 2;
    Counter f = new Counter();
    Console.WriteLine(f.Factorial(n));
  }
}
```

Компиляция

```
csc.exe Counter.cs
```

Выполнение

```
Counter.exe 5
```

Результат

```
120
```

Базовые типы и операции над ними

В обоих рассматриваемых языках имеются ссылочные типы и типы значений. Объекты ссылочных типов имеют собственную идентичность, на такой объект можно иметь ссылку из другого объекта, они передаются по ссылке, если являются аргументами или результатами методов. Объекты типов значений представляют собой значения, не имеющие собственной идентичности, — все равные между собой значения неотличимы друг от друга, никак нельзя сослаться только на одно из них.

В обоих языках есть примитивные типы, являющиеся типами значений, для представления простых данных: логических, числовых и символьных.

В Java только примитивные типы являются типами значений, все другие типы — ссылочные, являются наследниками класса `java.lang.Object`.

Для каждого примитивного типа есть класс-обертка, который позволяет представлять значения этого типа в виде объектов.

Между значениями примитивного типа и объектами соответствующего ему класса-обертки определены преобразования по умолчанию — *упаковка* и *распаковка* (*autoboxing* и *auto-unboxing*, введены в Java 5), позволяющие во многих случаях не создавать объект по значению и не вычислять значение по объекту явно. Но можно производить их и явно.

Однако вызывать методы у значений примитивных типов нельзя.

В C# есть возможность декларировать пользовательские типы значений — структурные типы и перечисления. Ссылочные типы называются классами и интерфейсами. Структурные типы, так же как и ссылочные, наследуют классу `System.Object`, который также можно использовать под именем **`object`**.

Для каждого примитивного типа есть структурный тип-обертка. Преобразования между ними производятся неявно, компилятор считает их различными именами одного и того же типа.

Поэтому все элементы класса **`object`** имеются во всех примитивных типах — у их значений можно, как у обычных объектов, вызывать методы.

Вполне законны, например, выражения `2.Equals(3)` и `(-175).ToString()`.

Логический тип

В Java он назван **`boolean`**, а его обертка — `java.lang.Boolean`.

В C# он назван **`bool`**, а его обертка — `System.Boolean`.

Значения этого типа — логические значения, их всего два — **`true`** и **`false`**. Нет никаких неявных преобразований между логическими и целочисленными значениями. Над значениями этого типа определены следующие операции.

- `==` и `!=` — сравнения на равенство и неравенство.
- `!` — отрицание.

- `&&` и `||` — условные (короткие) конъюнкция и дизъюнкция (‘и’ и ‘или’). Второй аргумент этих операций не вычисляется, если по значению первого уже ясно, чему равно значение выражения, т.е., в случае конъюнкции — если первый аргумент равен **`false`**, а в случае дизъюнкции — если первый аргумент равен **`true`**. С помощью условного оператора `?:` их можно записать так: `(x && y)` — `((x)?(y):false)`, `(x || y)` — `((x)?true:(y))`. Напомним, что означает условный оператор — выражение `a?x:y` вычисляет значение `a`, если оно **`true`**, то вычисляется и возвращается значение `x`, иначе вычисляется и возвращается значение `y`.
- `&` и `|` — (длинные) конъюнкция и дизъюнкция (‘и’ и ‘или’). У этих операций оба аргумента вычисляются всегда.
- `^` — исключающее ‘или’ или сумма по модулю 2.
- Для операций `&`, `|`, `^` имеются соответствующие операторы присваивания `&=`, `|=`, `^=`. Выражение `x op= y`, где `op` — одна из операций `&`, `|`, `^`, имеет тот же эффект, что и выражение `x = ((x) op (y))`, за исключением того, что значение `x` вычисляется ровно один раз.

Целочисленные типы

В обоих языках имеются следующие целочисленные типы.

- Тип байтовых целых чисел, называемый в Java **`byte`**, а в C# — **`sbyte`**. Его значения лежат между -2^7 и (2^7-1) (т.е. между -128 и 127)
- **`short`**, чьи значения лежат в интервале -2^{15} – $(2^{15}-1)$ (-32768 – 32767)
- **`int`**, чьи значения лежат в интервале -2^{31} – $(2^{31}-1)$ (-2147483648 – 2147483647)
- **`long`**, чьи значения лежат в интервале -2^{63} – $(2^{63}-1)$ (-9223372036854775808 – 9223372036854775807)

В C# имеются беззнаковые аналоги всех перечисленных выше типов:

свой тип **`byte`** со значениями от 0 до $(2^8-1 = 255)$.

`ushort` со значениями от 0 до $(2^{16}-1) = 65535$

`uint` со значениями от 0 до $(2^{32}-1) = 4294967295$

`ulong` со значениями от 0 до $(2^{64}-1) = 18446744073709551615$

Классы-обертки целочисленных типов называются так:

```
java.lang.Byte
java.lang.Short
java.lang.Integer
java.lang.Long
```

Типы-обертки целочисленных типов называются так:

```
System.SByte
System.Byte
System.Int16
System.UInt16
System.Int32
System.UInt32
System.Int64
System.UInt64
```

Минимальные и максимальные значения примитивных типов можно найти в их типах-обертках в виде констант (**`static final`** полей) `MIN_VALUE` и `MAX_VALUE`.

Минимальные и максимальные значения примитивных типов можно найти в их типах-обертках в виде констант `MinValue` и `MaxValue`.

Над значениями целочисленных типов определены следующие операции.

- `==`, `!=` — сравнение на равенство и неравенство.

- `<, <=, >, >=` — сравнение на основе порядка.
- `+, -, *, /, %` — сложение, вычитание, умножение, целочисленное деление, взятие остатка по модулю.
- `++, --` — увеличение и уменьшение на единицу. Если такой оператор написан до операнда, то значение всего выражения совпадает с измененным значением операнда, если после — то с неизмененным.
  В результате выполнения последовательности действий
  `x = 1; y = ++x; z = x++;`
  значение `x` станет равно 3, а значения `y` и `z` — 2.
- `~, &, |, ^` — побитовые операции дополнения, конъюнкции, дизъюнкции и исключающего ‘или’.
- `<<, >>` — операторы, сдвигающие биты своего первого операнда влево и вправо на число позиций, равное второму операнду.

В Java оператор >> сдвигает вправо биты числа, дополняя его слева значением знакового бита — нулем для положительных чисел и единицей для отрицательных.

Специальный оператор >>> используется для сдвига вправо с заполнением освобождающихся слева битов нулями.

В C# оператор >> сдвигает вправо биты числа, дополняя его слева для чисел со знаком значением знакового бита, а для беззнаковых чисел — нулем.

- Для операций `+, -, *, /, %, ~, &, |, ^, <<, >>` (и Java-специфичной операции >>>) имеются соответствующие операции присваивания. При этом выражение `x op= y`, где `op` — одна из этих операций, эквивалентно выражению `x = (T)((x) op (y))`, где `T` — тип `x`, за исключением того, что значение `x` вычисляется ровно один раз.

В Java результаты арифметических действий вычисляются в зависимости от типа этих результатов, с отбрасыванием битов, «вылезающих» за размер типа.

Таким образом, эти операции реализуют арифметику по модулю 2^n для n, подходящего для данного типа.

Арифметические операции над целыми числами приводят к созданию исключений только в трех случаях: при делении на 0 или вычислении остатка по модулю 0, при конвертации в примитивный тип ссылки на объект класса обертки, равной **`null`**, а также при исчерпании доступной Java-машине памяти, которое может случиться из-за применения операций `--` и `++` с одновременным созданием объектов классов-оберток.

В C# результат арифметических действий над целочисленными данными, приводящих к переполнению, зависит от контекста, в котором эти действия производятся.

Если действие происходит в unchecked контексте (т.е. внутри блока или выражения, помеченных ключевым словом **unchecked**), то вычисления производятся в арифметике по подходящему модулю 2^n.

Если же эти действия производятся в checked контексте (т.е. внутри блока или выражения, помеченных модификатором **checked**), то переполнение приводит к созданию исключения.

По умолчанию действия, производимые в ходе выполнения, происходят в unchecked контексте, а действия, которые выполняются над константами во время компиляции — в checked контексте. При этом создание исключения во время компиляции приводит к выдаче сообщения об ошибке.

Любые целочисленные типы можно явно приводить друг к другу, а неявные преобразования переводят из меньших типов в б*о*льшие, если при этом нет перехода от типа со знаком к беззнаковому (обратный переход возможен).

В обоих языках целочисленным типом считается и тип **`char`**, чьими значениями являются 16-битные символы (от `'\u0000'` до `'\uffff'`). Для него определен тот же набор операций, но преобразования между ним и другими типами по умолчанию не производятся (явные преобразования возможны).

Типы чисел с плавающей точкой

Представление типов значений с плавающей точкой, **`float`** и **`double`**, а также операции с ними, соответствуют стандарту на вычисления с плавающей точкой IEEE 754 (он же — IEC 60559) [11,12]. Согласно этому стандарту значение такого типа состоит из знакового бита, мантиссы и экспоненты (у значения **`float`** 23 бита отводятся на мантиссу и 8 на экспоненту, у **`double`** — 52 бита на мантиссу и 11 на экспоненту).

Помимо обычных чисел значения обоих типов включают -0.0 (кстати, написав так, вы получите обычный 0.0, поскольку этот текст будет воспринят как константа 0.0, к которой применен унарный оператор -; единственный способ получить -0.0 — конвертировать его битовое представление — в шестнадцатеричном виде для типа **`float`** он представляется как `0x80000000`, а для **`double`** — `0x8000000000000000`), положительные и отрицательные бесконечности (для типа **`float`** это `0x7f800000` и `0xff800000`, а для **`double`** — `0x7ff0000000000000` и `0xfff0000000000000`), а также специальное значение `NaN` (*Not-A-Number, не число*; оно может быть представлено любыми значениями, у которых экспонента максимальна, а мантисса не равна 0).

Для значений с плавающей точкой определены следующие операции.

- `==, !=` — сравнения на равенство и неравенство. В соответствии с IEEE 754 `NaN` не равно ни одному числу, в том числе самому себе. -0.0 считается равным 0.0.
- `<, <=, >, >=` — сравнения на основе порядка. +∞ больше, чем любой обычное число и -∞, а -∞ меньше любого конечного числа. `NaN` несравнимо ни с одним числом, даже с самим собой — это значит, что любая указанная операция возвращает **`false`**, если один из ее операндов — `NaN`. -0.0 считается равным, а не меньше, чем 0.0.
- `+, -, *, /, %` — сложения, вычитание, умножение, деление, взятие остатка по модулю, а также соответствующие операции присваивания с одновременным выполнением одного из этих действий. Все эти операции действуют согласно IEEE 754, кроме операции вычисления остатка, которая реализована так, чтобы при всех конечных `a` и `b` (`b != 0`) выполнялось `a%b == a - b*n`, где `n` — самое большое по абсолютной величине целое число, не превосходящее `|a/b|`, знак которого совпадает со знаком `a/b`. По абсолютной величине `a%b` всегда меньше `b`, знак `a%b` совпадает со знаком `a`.
  Согласно стандарту IEEE 754 все арифметические операции определены для бесконечных аргументов «естественным» образом: 1.0/0.0 дает +∞, -1.0/0.0 дает -∞, 0.0/0.0 — `NaN`, конечное `x` в сумме с +∞ дает +∞, а +∞+(-∞) — `NaN`. Если один из операндов `NaN`, то результат операции тоже `NaN`.
- `++, --` — увеличение и уменьшение на единицу. Для бесконечностей и `NaN` результат применения этих операторов совпадает с операндом.

В Java в классах `java.lang.Float` и `java.lang.Double` есть константы, равные максимальному конечному значению типа, минимальному положительному значению типа, положительной и отрицательной бесконечностям и `NaN`.

`Float.MAX_VALUE` $= (2-2^{-23})\cdot 2^{127}$

`Float.MIN_VALUE` $= 2^{-149}$

`Double.MAX_VALUE` $= (2-2^{-59})\cdot 2^{1023}$

`Double.MIN_VALUE` $= 2^{-1074}$

Бесконечности и `NaN` в обоих случаях

В C# соответствующие классы `System.Single` и `System.Double` также хранят эти значения в виде констант `MaxValue`, `Epsilon`, `PositiveInfinity`, `NegativeInfinity` и `NaN`.

называются `POSITIVE_INFINITY`, `NEGATIVE_INFINITY` и `NaN`.

В C# есть еще один тип для представления чисел с плавающей точкой — **`decimal`** (тип-обертка для него называется `System.Decimal`).

Значения этого типа представляются 128 битами, из которых один используется для знака, 96 — для двоичной мантиссы, еще 5 — для представления десятичной экспоненты, лежащей от 0 до 28. Остальные биты не используются.

Представляемое знаком s (+1 или -1), мантиссой m (0–(2^{96}-1)) и экспонентой e (0–28) значение равно $(-1)^{s} \cdot m \cdot 10^{-e}$.

Таким образом, значения этого типа могут, в отличие от стандартных типов **`float`** и **`double`**, представлять десятичные дроби с 28-ю точными знаками и используются для финансовых вычислений. Такая точность необходима, поскольку в этих вычислениях ошибки округления в сотые доли процента при накоплении за счет больших сумм и большого количества транзакций в течение нескольких лет могут привести к значительным суммам убытков для разных сторон.

Для типа **`decimal`** определены все те же операции, что и для типов с плавающей точкой, однако при выходе их результатов за рамки значений типа создается исключительная ситуация. Этот тип не имеет специальных значений -0.0, `NaN` и бесконечностей.

В Java классы, методы и инициализаторы могут быть помечены модификатором **`strictfp`**. Он означает, что при вычислениях с плавающей точкой в рамках этих деклараций все промежуточные результаты должны быть представлены в рамках того типа, к которому они относятся, согласно стандарту IEEE 754.

Иначе, промежуточные результаты вычислений со значениями типа **`float`** могут быть представлены в более точном виде, что может привести к отличающимся итоговым результатам вычислений.

Инструкции и выражения

Выражения

В обоих языках выражения строятся при помощи применения операторов к именам и литералам. Условно можно считать, что имеется следующий общий набор операторов.

- `x.y` — оператор уточнения имени, служит для получения ссылки на элемент пространства имен или типа, либо для получения значения поля (или свойства в C#);

- `f(x)` — оператор вызова метода (а также делегата в C#) с заданным набором аргументов;
- `a[x]` — оператор вычисления элемента массива (а также обращения к индексеру в C#);
- **`new`** — оператор создания нового объекта (или значения в C#), используется вместе с обращением к одному из конструкторов типа — **`new`** `MyType("Yes", 2)` (в Java с его помощью нельзя создавать значения примитивных типов);
- `++`, `--` — префиксные и постфиксные унарные операторы увеличения/уменьшения на 1;
- `(T)x` — оператор явного приведения к типу T;
- `+`, `-` — унарные операторы сохранения/изменения знака числа;
- `!` — унарный оператор логического отрицания;
- `~` — унарный оператор побитового отрицания;
- `*`, `/`, `%`, `+`, `-` — бинарные операторы умножения, деления, взятия остатка по модулю, сложения и вычитания;
- `<<`, `>>` — бинарные операторы побитовых сдвигов влево/вправо;
- `<`, `>`, `<=`, `>=` — бинарные операторы сравнения по порядку;
- `==`, `!=` — бинарные операторы сравнения на равенство/неравенство;
- `&`, `|`, `^` — бинарные операторы логических или побитовых операций: конъюнкции, дизъюнкции, сложения по модулю 2;
- `&&`, `||` — бинарные операторы условных конъюнкции и дизъюнкции, `(x && y)` эквивалентно `(x?y:false)`, а `(x || y)` — `(x?true:y)`;
- `?:` — тернарный условный оператор, выражение `a?x:y` вычисляет значение `a`, если оно **`true`**, то вычисляется и возвращается значение `x`, иначе вычисляется и возвращается значение `y`;
- `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `|=`, `^=` — бинарные операторы присваивания, все они, кроме первого, сначала производят некоторую операцию над старым значением левого операнда и значением правого, а затем присваивают полученный результат левому операнду.

Операторы	**Ассоциативность**
`x.y`, `f(x)`, `a[x]`, **`new`**, `x++`, `x--`	
`+`, `-`, `!`, `~`, `++x`, `--x`, `(T)x`	
`*`, `/`, `%`	левая
`+`, `-`	левая
`<<`, `>>`	левая
`<`, `>`, `<=`, `>=`	левая
`==`, `!=`	левая
`&`	левая
`^`	левая
`|`	левая
`&&`	левая
`||`	левая
`?:`	правая
`=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `|=`, `^=`	правая

Таблица 10. Приоритет и ассоциативность операторов.

В Таблице 10 операторы перечисляются сверху вниз в порядке уменьшения их приоритета, а также приводится ассоциативность всех операторов. Оператор `op` называется ***левоассоциативным***, если выражение `(x op y op z)` трактуется компилятором как `((x op y) op z)`, и ***правоассоциативным***, если оно трактуется как `(x op (y op z))`.

Помимо перечисленных выше операторов имеются также общие для обоих языков операции, которые выполняются при помощи различных конструкций — это получение объекта, представляющего тип, который задан по имени, и проверка принадлежности объекта или значения типу. В каждом из языков есть также несколько операторов, специфических для данного языка.

Получение объекта, представляющего тип, связано с механизмом ***рефлексии (reflection)***, имеющимся в обоих языках. Этот механизм обеспечивает отображение сущностей языка (типов, операций над ними, полей их данных и пр.) в объекты самого языка. В обоих языках операция получения объекта, представляющего тип, входит в группу операций с высшим приоритетом.

Любой тип Java однозначно соответствует некоторому объекту класса `java.lang.Class`, любой метод описывается с помощью одного из объектов класса `java.lang.reflect.Method`, любое поле — с помощью одного из объектов класса `java.lang.reflect.Field`.

Получить объект типа `Class`, представляющий тип `T` (даже если `T` = **`void`**), можно с помощью конструкции `T.`**`class`**.

Для проверки того, что выражение `x` имеет тип `T`, в Java используется конструкция
`(x` **`instanceof`** `T)`,
возвращающая значение логического типа.

В обоих языках операция проверки типа имеет такой же приоритет, как операторы <, >, <=, >=.

В Java есть дополнительный оператор сдвига числового значения вправо >>>, заполняющий освобождающиеся слева биты нулями.

Он имеет такой же приоритет, как и остальные операторы сдвига, и левую ассоциативность.

В C# типы представляются объектами класса `System.Type`, методы — объектами `System.Reflection.MethodInfo`, а поля — объектами `System.Reflection.FieldInfo`.

Объект типа `System.Type`, представляющий тип `T`, можно получить при помощи конструкции **`typeof`**`(T)`.

Для проверки того, что выражение `x` имеет тип `T`, в C# используется конструкция
`(x` **`is`** `T)`,
имеющая логический тип.

Эта проверка использует естественные преобразования типов (подтипа в более общий тип или наоборот, если точный тип объекта является подтипом `T`) и автоупаковку/распаковку, не затрагивая определенных пользователем неявных преобразований.

В C# имеется и другой оператор, связанный с преобразованием типа.

Для преобразования объекта `x` к заданному ссылочному типу `T` можно использовать конструкцию
`(x` **`as`** `T)`,

тип результата которой — `T`.

Если в результате естественных преобразований типов и автоупаковки/распаковки, значение `x` не преобразуется к типу `T`, то результат этого выражения — **`null`**.

Приоритет этого оператора такой же, как у оператора **`is`**, а ассоциативность — левая.

Соответствующий оператор присваивания `>>>=` имеет такой же приоритет, как и другие операторы присваивания, и правую ассоциативность.

В C# можно строить выражения, в рамках которых переполнения при арифметических действиях вызывают (или не вызывают) исключения при помощи оператора **`checked`**`(x)` (**`unchecked`**`(x)`), где `x` — выражение, контекст вычисления которого мы хотим определить (см. раздел о целочисленных типах).

Оба этих оператора входят в группу операторов с высшим приоритетом.

Выражение **`default`**`(T)` используется в C# 2.0 для получения значения типа `T` по умолчанию.

Для ссылочных типов это **`null`**, для числовых типов — 0, для логического типа — **`false`**, а для остальных типов значений определяется на основе их структуры.

Это выражение используется для инициализации данных в шаблонных типах, зависящих от типового параметра, который может быть как ссылочным типом, так и типом значений.

Этот оператор входит в группу с высшим приоритетом.

Оператор `??` *(null coalescing operator)* используется в C# 2.0 в следующем смысле. Выражение `(x??y)` эквивалентно `((x == null)?y:x)`, только значение `x` вычисляется однократно, т.е., если значение `x` не равно **`null`**, то результатом этой операции является `x`, а иначе `y`.

Этот оператор имеет приоритет меньший, чем приоритет условной дизъюнкции `||`, но больший, чем приоритет условного оператора `?:`. Он правоассоциативен.

В C# 2.0 введен дополнительный оператор `::` для разрешения контекста имен в рамках глобального пространства имен или определенных синонимов.

Дело в том, что в C# при разрешении имен, построенных с помощью точек, разделяющих идентификаторы, возможны многочисленные проблемы, связанные с непредвиденными модификациями библиотек.

Например, если мы написали директиву **`using`** `System.IO;`, чтобы использовать класс `FileStream` с коротким именем, и одновременно определяем в этом контексте

класс `EmptyStream`, то, если в будущем в `System.IO` появится класс `EmptyStream`, полученный код перестанет компилироваться.

Эту ситуацию можно разрешить при помощи синонимов, определив, например, для `System.IO` синоним `SIO`, а для нашей собственной библиотеки, куда входит `EmptyStream`, синоним `MIO`, и используя имена классов только вместе с синонимами — `SIO.FileStream`, `MIO.EmptyStream`. Однако если в одной из используемых библиотек будет введено пространство имен `MIO`, проблема возникнет вновь.

Чтобы однозначно отделить типы из внешних библиотек от внутрипрограммных, можно использовать оператор `::`. При этом левый аргумент такого оператора может иметь два вида. Либо он имеет значение **`global`** — тогда имя, заданное правым аргументом, ищется в глобальном пространстве имен и конфликтует с внутренними именами. Либо он является именем синонима — тогда имя, заданное правым аргументом, ищется только в пространстве имен, определяемом этим синонимом.

Этот оператор входит в группу с высшим приоритетом.

Тип или часть его операций, или даже отдельный блок в C# могут быть помечены модификатором **`unsafe`**. При этом содержимое помеченного типа, операции или блока попадает в *небезопасный контекст (unsafe context)*. В рамках небезопасного контекста можно использовать указатели и операции над указателями, в частности, доступ к элементу данных по указателю с помощью оператора `->`, построение указателей на данные и разыменование указателей, арифметические действия над указателями.

В данном курсе мы не будем больше касаться правил написания небезопасного кода в C#, предоставляя заинтересованному читателю самому разобраться в них с помощью [8].

Инструкции

Большинство видов инструкций в Java и C# являются общими и заимствованы из языка C. В обоих языках есть понятие *блока* — набора инструкций, заключенного в фигурные скобки.

- Пустая инструкция `;` допускается в обоих языках.
- Декларации локальных переменных устроены совершенно одинаково — указывается тип переменной, затем ее идентификатор, а затем, возможно, инициализация. Инициализировать переменную можно каким-то значением ее типа. Использование неинициализированных переменных во многих случаях определяется компилятором и

считается ошибкой (но не всегда). Однако даже при отсутствии инициализации переменной, ей все равно будет присвоено значение по умолчанию для данного типа. Массивы могут быть инициализированы с помощью специальных выражений, перечисляющих значения элементов массива, например
`int[][] array = new int[][]{{0, 1}, {2, 3, 4}};`

- Инструкция может быть помечена с помощью метки, которая стоит перед самой инструкцией и отделяется от нее с помощью двоеточия.
- Инструкция может быть построена добавлением точки с запятой в конец выражения определенного вида. Такое выражение должно быть одним из следующих:
  - присваиванием;
  - выражением, в котором последним оператором было уменьшение или увеличение на единицу (++, --), все равно, префиксное или постфиксное;
  - вызовом метода в объекте или классе (в C# — еще и вызовом делегата);
  - созданием нового объекта.
- Условная инструкция имеет вид
  `if(`*`expression`*`)` *`statement`*
  или
  `if(`*`expression`*`)` *`statement`* `else` *`statement1`*
  где *`expression`* — выражение логического типа (или приводящегося к логическому), а *`statement`* и *`statement1`* — инструкции.
- Инструкция выбора имеет вид
  `switch(`*`expression`*`) { … }`
  Внутри ее блока различные варианты действий для различных значений выражения *`expression`* описываются с помощью списков инструкций, помеченных либо меткой **`case`** с возможным значением выражения, либо меткой **`default`**. Группа инструкций, помеченная **`default`**, выполняется, если значение выражения выбора не совпало ни с одним из значений, указанных в метках **`case`**. Один набор инструкций может быть помечен несколькими метками. Наборы инструкций могут отделяться друг от друга инструкциями **`break;`**
  Тип *`expression`* может быть целочисленным или приводящимся к нему, либо перечислимым типом. В C# допускается использование для выбора выражений типа **`string`**.
  Значения, которые используются в метках **`case`**, должны быть константными выражениями.

В Java группа инструкций для одного значения может оканчиваться инструкцией **`break`**, а может и не оканчиваться. Во втором случае после ее выполнения управление переходит на следующую группу инструкций.

```
public class A
{
  public static void main(String[] args)
  {
    if(args.length > 0)
    {
      int n = Integer.parseInt(args[0]);
      switch(n)
      {
        case 0:
          System.out.println("n = 0");

        case 1:
```

В C# группа инструкций для одного значения (включая и группу, помеченную **`default`**) всегда должна оканчиваться либо **`break`**, либо **`goto default`**, либо **`goto case`** *`value`* с каким-то из значений, указанных в рамках той же инструкции выбора.

```
using System;

public class A
{
  public static void Main(string[] args)
  {
    if(args.Length > 0)
    {
      int n = Int32.Parse(args[0]);
      switch(n)
      {
        case 0:
          Console.WriteLine("n = 0");
          goto case 1;
        case 1:
```

```
          System.out.println
            ("n = 0 or n = 1");
          break;
        case 2:case 3:
          System.out.println
            ("n = 2 or n = 3");
          break;
        default:
          System.out.println
            ("n is out of [0..3]");

      }
    }
    else
      System.out.println("No arguments");
  }
}
```

```
          Console.WriteLine
            ("n = 0 or n = 1");
          break;
        case 2:case 3:
          Console.WriteLine
            ("n = 2 or n = 3");
          break;
        default:
          Console.WriteLine
            ("n is out of [0..3]");
          break;
      }
    }
    else
      Console.WriteLine("No arguments");
  }
}
```

- Циклы **while** и **do** в обоих языках устроены одинаково.
  while(*expression*) *statement*
  do *statement* **while**(*expression*);
  Здесь expression — логическое выражение, условие цикла, statement — тело цикла. Правила выполнения этих циклов фактически заимствованы из языка C. Первый на каждой итерации проверяет условие и, если оно выполнено, выполняет свое тело, а если нет — передает управление дальше. Второй цикл сначала выполняет свое тело, а потом проверяет условие.
- Цикл **for** в обоих языках заимствован из языка C.
  for(A; B; C) *statement*
  выполняется практически как
  A; **while**(B) { *statement* C; }
  Любой из элементов A, B, C может отсутствовать, B должно быть выражением логического типа (при отсутствии оно заменяется на **true**), A и C должны быть наборами выражений (A может включать и декларации переменных), разделенных запятыми.
  Помимо обычного **for** в обоих языках имеется специальная конструкция для цикла, перебирающего элементы коллекции.

В Java синтаксис цикла перебора элементов коллекции такой

for (**final**$_{opt}$ *type id* : *expression*)
  statement

При этом выражение *expression* должно иметь тип `java.lang.Iterable` или тип массива.

В первом случае такой цикл эквивалентен следующему (T далее обозначает тип результат метода `iterator()` у *expression*, v — нигде не используемое имя).

```
for(T v = expression.iterator();
    v.hasNext(); )
{
  final_opt type id = v.next();
  statement
}
```

Во втором случае, когда *expression* — массив типа `T[]`, эта конструкция эквивалентна следующей (a, i — нигде не используемые имена)

В C# синтаксис цикла перебора элементов коллекции такой

foreach (*type id* **in** *expression*)
  statement

Выражение *expression* должно быть массивом, или иметь тип `System.Collections.IEnumerable` или `System.Collections.Generic.IEnumerable<T>`, или же его тип должен иметь метод `GetEnumerator()`, результат которого, в свою очередь, должен иметь свойство `Current` и метод `MoveNext()`.

Тип результата метода `GetEnumerator()` во всех случаях, кроме массива, называется *типом итератора (enumerator type)*. Тип свойства `Current`, которое имеется у типа итератора, должен совпадать с *type*.

Пусть тип итератора E, а e — неиспользуемое имя. Тогда приведенная конструкция, с точностью до некоторых деталей, эквивалентна

```
T[] a = expression;
for(int i = 0; i < a.length; i++)
{
  finalopt type id = v.next();
  statement
}
```

следующей.

```
E e = expression.GetEnumerator();
while(e.MoveNext())
{
  type id = (type)e.Current;
  statement
}
```

Опущенные детали касаются освобождения ресурсов, используемых итератором (см. далее описание инструкции **using**).

Пример использования перебора элементов коллекции:

```
public class A
{
  public static void main(String[] args)
  {
    int i = 1;
    for(String s : args)
      System.out.println((i++) +
        "-th argument is " + s);
  }
}
```

Пример использования перебора элементов коллекции:

```
using System;

public class A
{
  public static void Main(string[] args)
  {
    int i = 1;
    foreach (string s in args)
      Console.WriteLine((i++) +
        "-th argument is " + s);
  }
}
```

- Инструкции прерывания **break** и **continue** также заимствованы из C.
  Инструкция **break** прерывает выполнение самого маленького содержащего ее цикла и передает управление первой инструкции после него. Инструкция **continue** прерывает выполнение текущей итерации и переходит к следующей, если она имеется (т.е. условие цикла выполнено в сложившейся ситуации), иначе тоже выводит цикла.
  При выходе с помощью **break** или **continue** за пределы блока **try** (см. ниже) или блока **catch**, у которых имеется соответствующий блок **finally**, сначала выполняется содержимое этого блока **finally**.

В Java инструкция **break** используется для прерывания выполнения не только циклов, но и обычных блоков (наборов инструкций, заключенных в фигурные скобки).

Более того, после **break** (или **continue**) может стоять метка. Тогда прерывается выполнение того блока/цикла (или же начинается новая итерация того цикла), который помечен этой меткой. Этот блок (или цикл) должен содержать такую инструкцию внутри себя.

- Инструкция возврата управления **return** используется для возврата управления из операции (метода, оператора, метода доступа к свойству и пр., см. далее). Если операция должна вернуть значение некоторого типа, после **return** должно стоять выражение этого же типа.
- Инструкция создания исключительной ситуации **throw** используется для выброса исключительной ситуации. При этом после **throw** должно идти выражение, имеющее тип исключения.
  Исключение (exception) представляет собой объект, содержащий информацию о какой-то особой (исключительной) ситуации, в которой операция не может вернуть обычный результат. Вместо обычного результата из нее возвращается объект-исключение — при этом говорят, что исключение *было выброшено* из операции. Механизм этого возвращения

несколько отличается от механизма возвращения обычного результата, и обработка исключений оформляется иначе (см. следующий вид инструкций), чем обработка обычных результатов работы операции.

- Исключения в обоих языках относятся к особым типам — классам исключений. Только объекты таких классов могут быть выброшены в качестве исключений. Классами исключений являются все наследники классов `java.lang.Throwable` в Java и `System.Exception` в C#.
- Объекты-исключения содержат, как минимум, следующую информацию.
  - Сообщение о возникшей ситуации (его должен определить автор кода операции, выбрасывающей это исключение).
    В Java это сообщение можно получить с помощью метода `String getMessage()`, а в C# — с помощью свойства **`string`** `Message`.
  - Иногда возникают цепочки «наведенных» исключений, если обработка одного вызывает выброс другого. Каждый объект-исключение содержит ссылку на другое исключение, непосредственно вызвавшее это. Если данное исключение не вызвано никаким другим, эта ссылка равна **`null`**.
    В Java эту ссылку можно получить с помощью метода `Throwable getCause()`, а в C# — с помощью свойства `System.Exception.InnerException`.
  - Для описания ситуации, в которой возникло исключение, используется состояние стека исполнения программы — список методов, которые вызывали друг друга перед этим, и указание на место в коде каждого такого метода. Это место обозначает место вызова следующего метода по стеку или, если это самый последний метод, то место, где и возникло исключение. Обычно указывается номер строки, но иногда он недоступен, если соответствующий метод присутствует в системе только в скомпилированном виде или является внешним для Java машины.
    Информация о состоянии стека на момент возникновения исключения, как и его сообщение, автоматически выводится в поток сообщений об ошибках, если это исключение остается необработанным в программе.
    В Java состояние стека для данного исключения можно получить с помощью метода `StackTraceElement[] getStackTrace()`, возвращающего массив элементов стека. Каждый такой элемент несет информацию о файле (`String getFileName()`), классе (`String getClassName()`) и методе (`String getMathodName()`), а также о номере строки (**`int`** `getLineNumber()`).
    В C# можно сразу получить полное описание состояния стека в виде одной строки с помощью свойства **`string`** `StackTrace`.
- Блок обработки исключительных ситуаций выглядит так.

```
try                    {  statements   }
catch ( type_1 e_1 ) { statements_1 }
…
catch ( type_n e_n ) { statements_n }
finally                { statements_f }
```

Если во время выполнения одной из инструкций в блоке, следующем за **`try`**, возникает исключение, управление передается на первый блок **`catch`**, обрабатывающий исключения такого же или более широкого типа. Если подходящих блоков **`catch`** нет, выполняется блок **`finally`** и исключение выбрасывается дальше.

Блок **`finally`** выполняется всегда — сразу после блока **`try`**, если исключения не возникло, или сразу после обрабатывавшего исключение блока **`catch`**, даже если он выбросил новое исключение.

В этой конструкции могут отсутствовать блоки **`catch`** или блок **`finally`**, но не то и другое одновременно. В C# разрешается опускать имя объекта-исключения в **`catch`**, если он не используется при обработке соответствующей исключительной ситуации.

```
public class A
{
  public static void main(String[] args)
  {
    try {
      if(args.length > 0)
        System.out.println
     ("Some arguments are specified");
      else throw new
        IllegalArgumentException
        ("No arguments specified");
    }
    catch(RuntimeException e)
    {
      System.out.println
        ("Exception caught");
      System.out.println
        ("Exception type is " +
           e.getClass().getName());
      System.out.println
        ("Exception message is \"" +
           e.getMessage() + "\"");
    }
    finally
    {
      System.out.println
        ("Performing finalization");
    }
  }
}
```

В Java, начиная с версии 1.4, появилась инструкция **`assert`**, предназначенная для выдачи отладочных сообщений.

Эта инструкция имеет один из двух видов:

`assert` *`expression`* `;`

`assert` *`expression`* `:` *`expression_s`* `;`

Выражение *`expression`* должно иметь логический тип, а выражение *`expression_s`* — произвольный.

Проверка таких утверждений может быть выключена. Тогда эта инструкция ничего не делает, и значения входящих в нее выражений не вычисляются.

Если проверка утверждений включена, то вычисляется значение *`expression`*. Если оно равно **`true`**, управление переходит дальше, иначе в обоих случаях выбрасывается исключение `java.lang.AssertionError`.

Во втором случае еще до выброса исключения вычисляется значение выражения *`expression_s`*, оно преобразуется в строку и записывается в качестве сообщения в создаваемое исключение.

```
using System;

public class A
{
  public static void Main(string[] args)
  {
    try {
      if(args.Length > 0)
        Console.WriteLine
     ("Some arguments are specified");
      else throw new
        ArgumentException
        ("No arguments specified");
    }
    catch(Exception e)
    {
      Console.WriteLine
        ("Exception caught");
      Console.WriteLine
        ("Exception type is " +
          e.GetType().FullName);
      Console.WriteLine
        ("Exception message is \"" +
          e.Message + "\"");
    }
    finally
    {
      Console.WriteLine
        ("Performing finalization");
    }
  }
}
```

В C# имеется возможность использовать инструкцию **`goto`**. Эта инструкция передает

управления на инструкцию, помеченную меткой, которая следует за **goto**.

Как мы уже видели, помимо обычных меток, в **goto** могут использоваться метки **case** вместе со значениями и метка **default**. В этих случаях инструкция **goto** должна находиться внутри блока **switch**, в котором имеются эти метки.

При выходе с помощью **goto** из блока **try** или блока **catch**, у которых имеется соответствующий блок **finally**, сначала выполняется содержимое этого блока **finally**.

Ключевые слова **checked** и **unchecked** в C# могут помечать блок, определяя тем самым контекст вычислений в рамках этого блока (см. раздел о целочисленных типах).

Инструкция **using** может быть использована в C#, чтобы выполнить действия, требующие захвата каких-либо ресурсов, без необходимости заботиться потом об их освобождении.

Эта инструкция имеет вид

using (*expression*) *statement* или
using (*declaration*) *statement*

где *declaration* — это декларация одной или нескольких переменных.

Первый вид этой инструкции сводится ко второму — если тип используемого выражения `T`, и имя `v` нигде не используется, то он эквивалентен
using (`T v =` *expression*) *statement*

Эта конструкция, в свою очередь, эквивалентна следующей.

```
{
  T v = expression;
  try { statement }
  finally { disposal }
}
```

Здесь *disposal* представляет вызов метода `Dispose()`, который должен быть у типа `T`, с возможной предварительной проверкой того, что переменная `v` не равна **null**, и приведением ее к типу `System.IDisposable`, если `T` является его подтипом.

В версии 2.0 в C# введены две инструкции **yield**, предназначенные для более удобного построения итераторов.

Блок, содержащий инструкцию **yield**, называется *итерационным (iterator block)* и может быть телом метода, оператора или метода доступа к свойству и не должен быть

блоком **finally**, **catch** или блоком **try**, у которого есть соответствующие **catch**-блоки. Этот блок порождает последовательность значений одного типа. Сам метод или оператор должен возвращать объект одного из четырех типов:

`System.Collections.IEnumerable`,

`System.Collections.IEnumerator`,

`System.Collections.Generic.IEnumerable<T>`,

`System.Collections.Generic.IEnumerator<T>`.

В первых двух случаях порождаются объекты типа **object**, во вторых двух — значения типа `T`.

Для возвращения одного из этой последовательности значений используется инструкция **yield return** *expression;* Выражение в ней должно иметь соответствующий тип, **object** или `T`.

Для указания на то, что порождаемая итерационным блоком последовательность значений завершилась, используется инструкция **yield break**;

Пример реализации итератора коллекции с использованием **yield** приведен ниже.

```
using System;

public class MyArrayList<T>
{
  T[] items = new T[10];
  int size = 0;

  public int Count
  { get { return size; } }

  public T this[int i]
  {
    get
    {
      if(i < 0 || i >= size) throw new
        IndexOutOfRangeException();
      else return items[i];
    }
    set
    {
      if(i < 0 || i > size) throw new
        IndexOutOfRangeException();
      else if (i == size)
      {
        T[] newItems =
          new T[size + 10];
        Array.Copy
          (items, newItems, size++);
      }
      items[i] = value;
    }
  }
```

```
  public IEnumerator<T> GetEnumerator
    ()
  {
    for(int i = 0; i < size; i++)
      yield return items[i];
  }
}

public class A
{
  public static void Main()
  {
    MyArrayList<string> l =
      new MyArrayList<string>();


    l[0] = "First";
    l[1] = "Second";
    l[2] = "Third";

    foreach (string s in l)
      Console.WriteLine(s);
  }
}
```

Инструкции обоих языков, предназначенные для синхронизации работы нескольких потоков, рассматриваются в следующей лекции, в разделе, посвященном разработке многопоточных программ.

Пользовательские типы

В обоих рассматриваемых языках имеются ссылочные типы и типы значений. Объекты ссылочных типов имеют собственную идентичность, а значения такой идентичности не имеют. Объекты ссылочных типов можно сравнивать на совпадение или несовпадение при помощи операторов == и !=. В C# эти операторы могут быть перегружены, поэтому, чтобы сравнить объекты на идентичность, лучше привести их сначала к типу **`object`**.

В обоих языках можно создавать пользовательские ссылочные типы, определяя классы и интерфейсы. Кроме того, можно использовать массивы значений некоторого типа. В C# можно определять пользовательские типы значений, а в Java типами значений являются только примитивные.

Класс представляет собой ссылочный тип, объекты которого могут иметь сложную структуру и могут быть задействованы в некотором наборе операций. Структура данных объектов класса задается набором ***полей (fields)*** этого класса. В каждом объекте каждое поле имеет определенное значение, могущее быть ссылкой на другой или тот же самый объект.

Над объектом класса можно выполнять операции, определенные в этом классе. Термин «операция» будет употребляться для обозначения методов обоих языков, а также операторов, методов доступа к свойствам, индексерам и событиям в C# (см. ниже). Для каждой операции в классе определяются ее ***сигнатура*** и ***реализация***.

Полная сигнатура операции — это ее имя, список типов, значения которых она принимает в качестве параметров, а также тип ее результата и список типов исключений, которые могут быть выброшены из нее. Просто ***сигнатурой*** будем называть имя и список типов параметров операции — этот набор обычно используется для однозначного определения операции в рамках класса. Все операции одного класса должны различаться своими (неполными) сигнатурами, хотя некоторые из них могут иметь одинаковые имена. Единственное исключение из этого правила касается только C# и будет описано ниже.

Реализация операции представляет собой набор инструкций, выполняемых каждый раз, когда эта операция вызывается. ***Абстрактный класс*** может не определять реализации для некоторых

своих операций — такие операции называются ***абстрактными***. И абстрактные классы, и их абстрактные операции помечаются модификатором **`abstract`**.

Поля и операции могут быть ***статическими (static)***, т.е. относиться не к объекту класса, а к классу в целом. Для получения значения такого поля достаточно указать класс, в котором оно определено, а не его объект. Точно так же, для выполнения статической операции не нужно указывать объект, к которому она применяется.

Интерфейс — ссылочный тип, отличающийся от класса тем, что он не определяет структуры своих объектов (не имеет полей) и не задает реализаций для своих операций. Интерфейс — это абстрактный тип данных, над которыми можно выполнять заданный набор операций. Какие конкретно действия выполнятся для данного объекта, зависит от его точного типа (в обоих случаях это может быть класс, реализующий данный интерфейс, а в C# — еще и структурный тип, тоже реализующий данный интерфейс).

Из последней фразы может быть понятно, что и в Java, и в C# объект может относиться сразу к нескольким типам. Один из этих типов, самый узкий, — ***точный тип объекта***, а остальные (более широкие) являются классами-предками этого типа или реализуемыми им интерфейсами. Точным типом объекта не может быть интерфейс или абстрактный класс, потому что для них не определены точные действия, выполняемые при вызове (некоторых) их операций.

Классы и интерфейсы (а также отдельные операции) в обоих языках могут быть ***шаблонными (generic)***, т.е. иметь типовые параметры (соответствующие конструкции введены в Java 5 и C# 2.0). При создании объекта такого класса нужно указывать конкретные значения его типовых параметров.

Примеры деклараций классов и интерфейсов для обоих языков приведены ниже. В обоих случаях определяется шаблонный интерфейс очереди, которая хранит объекты типа-параметра, и класс, реализующий очередь на основе ссылочной структуры. Показан также пример использования такой очереди.

```
public interface Queue <T>
{
  void put (T o);
  T    get ();
  int  size();
}
public class LinkedQueue <T>
  implements Queue <T>
{
  public void put (T o)
  {
    if(last == null)
    {
      first = last = new Node <T> (o);
    }
    else
    {
      last.next = new Node <T> (o);
      last = last.next;
    }
    size++;
  }

  public T    get ()
  {
    if(first == null) return null;
    else
    {
      T result = first.o;
      if(last == first) last = null;
      first = first.next;
```

```
using System;

public interface IQueue <T>
{
  void Put (T o);
  T    Get ();
  int  Size();
}
public class LinkedQueue <T>
  : IQueue <T>
{
  public void Put (T o)
  {
    if(last == null)
    {
      first = last = new Node <T> (o);
    }
    else
    {
      last.next = new Node <T> (o);
      last = last.next;
    }
    size++;
  }

  public T    Get ()
  {
    if(first == null) return default(T);
    else
    {
      T result = first.o;
      if(last == first) last = null;
      first = first.next;
```

```
      size--;
      return result;
    }
  }

  public int  size()
  {
    return size;
  }


  private Node <T> last  = null;
  private Node <T> first = null;
  private int      size  = 0;

  private static class Node <E>
  {

    E        o    = null;
    Node<E>  next = null;

    Node (E o)
    {
      this.o = o;
    }
  }
}
public class Program
{
  public static void main(String[] args)
  {
    Queue<Integer> q =
      new LinkedQueue<Integer>();

    for(int i = 0; i < 10; i++)
      q.put(i*i);

    while(q.size() != 0)
      System.out.println
        ("Next element + 1: " +
           (q.get()+1));
  }
}
```

```
      size--;
      return result;
    }
  }

  public int  Size()
  {
    return size;
  }


  private Node <T> last  = null;
  private Node <T> first = null;
  private int      size  = 0;

  internal class Node <E>
  {

    internal E        o    = default(E);
    internal Node <E> next = null;

    internal Node <E> (E o)
    {
      this.o = o;
    }
  }
}
public class Program
{
  public static void Main()
  {
    Queue<int> q =
      new LinkedQueue<int>();

    for(int i = 0; i < 10; i++)
      q.Put(i*i);

    while(q.Size() != 0)
      Console.WriteLine
        ("Next element + 1: " +
          (q.Get()+1));
  }
}
```

Обе программы выдают на консоль текст

```
Next element + 1: 1
Next element + 1: 2
Next element + 1: 5
Next element + 1: 10
Next element + 1: 17
Next element + 1: 26
Next element + 1: 37
Next element + 1: 50
Next element + 1: 65
Next element + 1: 82
```

На основе пользовательского или примитивного типа можно строить *массивы* элементов данного типа. Тип массива является ссылочным и определяется на основе типа элементов массива. Количество элементов массива в обоих языках — это свойство конкретного объекта-массива, которое задается при его построении и далее остается неизменным. В обоих языках можно строить массивы массивов и пр.

В Java можно строить только одномерные массивы из объектов, которые, однако, сами

В C# есть возможность строить многомерные массивы в дополнение к массивам массивов.

могут быть массивами.

```
int[] array = new int[3];
String[] array1 =
  new String[]{"First","Second"};
int[][] arrayOfArrays = new int[][]
  {{1, 2, 3}, {4, 5}, {6}};
```

Количество элементов в массиве доступно как значение поля `length`, имеющегося в каждом типе массивов.

```
int[] array = new int[3];
string[] array1 =
  new string[]{"First","Second"};
int[][] arrayOfArrays = new int[][]
  {{1, 2, 3}, {4, 5}, {6}};
int[,] twoDimensionalArray =
  new int[,] {{1, 2}, {3, 4}};
```

Любой тип массива наследует системному типу `System.Array`, и любой объект-массив имеет все свойства и методы этого типа.

Общее количество элементов в массиве (во всех размерностях) доступно как значение, возвращаемое свойством `Length`. Количество измерений в массиве — значение свойства `Rank`.

В обоих языках есть возможность декларировать ***перечислимые типы (enums)***, объекты которых представляются именованными константами. Однако реализована эта возможность по-разному.

В Java перечислимые типы (введены в Java 5) являются ссылочными, частным случаем классов. По сути, набор констант перечислимого типа — это набор статически (т.е. во время компиляции, а не в динамике, во время работы программы) определенных объектов этого типа.

Невозможно построить новый объект перечислимого типа — декларированные константы ограничивают множество его возможных значений. Любой его объект совпадает с одним из объектов-констант, поэтому их можно сравнивать при помощи оператора `==`.

Пример декларации перечислимого типа приведен ниже.

```
public enum Coin
{
  PENNY  ( 1),
  NICKEY ( 5),
  DIME   (10),
  QUARTER(25);

  Coin(int value)
  { this.value = value; }

  public int value()
  { return value; }

  private int value;
}
```

Как видно из примеров, перечисления в Java устроены несколько сложнее, чем в C#.

Возможны декларации методов перечислимого

В C# перечислимые типы являются типами значений, определяемыми на основе некоторого целочисленного типа (называемого *базовым;* по умолчанию это **`int`**). Каждая константа представляет собой некоторое значение базового типа. Однако можно искусственно построить другие значения перечислимого типа из значений его базового типа.

Пример декларации перечислимого типа приведен ниже.

```
public enum Coin : uint
{
  PENNY   =  1,
  NICKEY  =  5,
  DIME    = 10,
  QUARTER = 25
}
```

типа и их отдельная реализация для каждой из констант.

```
public enum Operation
{
  ADD {
    public int eval(int a, int b)
    { return a + b; }
  },
  SUBTRACT {
    public int eval(int a, int b)
    { return a - b; }
  },
  MULTIPLY {
    public int eval(int a, int b)
    { return a * b; }
  },
  DIVIDE {

    public int eval(int a, int b)
    { return a / b; }
  };

  public abstract int eval
                      (int a, int b);
}
```

В C# имеется возможность декларировать пользовательские типы значений, помимо перечислимых. Такие типы называются ***структурами***.

Структуры описываются во многом похоже на классы, они так же могут реализовывать интерфейсы и наследовать классам, но имеют ряд отличий при использовании, инициализации переменных структурных типов и возможности описания различных членов. Все эти особенности связаны с тем, что структуры — типы значений. Две переменных структурного типа не могут иметь одно значение — только равные. Значение структурного типа содержит все значения полей. При инициализации полей структуры используются значения по умолчанию для их типов (0 для числовых, **`false`** для логического типа, **`null`** для ссылочных типов и построенное так же значение по умолчанию для структурных).

Пример декларации структуры приведен ниже.

```
public struct Point2D
{
  private double x;
  private double y;

  public Point2D(double x, double y)
  {
    this.x = x;
    this.y = y;
  }
```

```
  public double X() { return x; }
  public double Y() { return y; }
}
```

Все структуры считаются наследующими ссылочному типу **object**, поэтому возможно приведение значения структуры к этому типу. Наоборот, если известно, что данный объект типа **object** представляет собой значение некоторого структурного типа, он может быть приведен к этому типу.

При переводе значений между структурным типом и **object** производятся их преобразования, называющиеся ***упаковкой (autoboxing)*** — построение объекта, хранящего значение структурного типа, — и ***распаковкой (auto-unboxing)*** — выделение значения из хранящего его объекта.

Эти преобразования строят копии преобразуемых значений. Поэтому изменения значения, хранимого в объекте, никак не отражаются на значениях, полученных ранее из него распаковкой.

При присваивании переменной структурного типа некоторого значения также происходит копирование этого значения. При присваивании переменной ссылочного типа в нее копируется значение ссылки, а сам объект, на который указывает эта ссылка, не затрагивается.

В Java, помимо явно описанных типов, можно использовать *анонимные классы (anonymous classes)*.

Анонимный класс всегда реализует какой-то интерфейс или наследует некоторому классу. Когда объект анонимного класса создается в каком-то месте кода, все описание элементов соответствующего класса помещается в том же месте. Имени у анонимного класса нет.

Ниже приведен пример использования объекта анонимного класса, реализующего интерфейс стека.

```
interface Stack <T>
{
  void push(T o);
  T    pop ();
}

public class B
{
  public void m()
  {
    Stack<Integer> s =
      new Stack<Integer>() {
      final static int maxSize = 10;
      int[] values = new int[maxSize];
      int last = -1;
```

```
    public void push(Integer i) {
      if(last + 1 == maxSize)
        throw new
        TooManyElementsException();
      else values[++last] = i;
    }

    public Integer pop() {
      if(last - 1 < -1)
        throw new
          NoElementsException();
      else return values[last--];
    }
  };

  s.push(3);
  s.push(4);
  System.out.println(s.pop() + 1);
 }
}
```

В C# 2.0 есть специальная конструкция для ***обнуляемых (nullable)*** типов значений — переменная такого типа может иметь либо определенное значение, либо не иметь значения, что представляется как равенство **`null`**.

Эта возможность еще отсутствует в версии стандарта [8], о ней можно прочитать в [9].

В основе обнуляемого типа значений всегда лежит обычный тип значений — примитивный, структурный или перечислимый.

```
bool? mayBeNullFlag     = null;
int? mayBeNullNumber    = 5;
Point2D? mayBeNullPoint = null;
```

Обозначение `T?` является сокращением от `System.Nullable<T>`. Этот тип имеет свойства `HasValue`, возвращающее **`true`** тогда, когда его значение является значением типа `T`, а не **`null`**, и `Value`, возвращающее это значение, если оно не **`null`**.

Определено неявное преобразование значений типа `T` в `T?`.

Для обнуляемых типов, построенных на основе примитивных, определены все те же операции.

Арифметические действия над значениями обнуляемых типов возвращают **`null`**, если один из операндов равен **`null`**.

Сравнения по порядку (<, >, <=, >=) возвращают **`false`**, если один из операндов равен **`null`**.

Можно использовать сравнение значений обнуляемых типов на равенство или неравенство **`null`**.

Для типа **`bool`**`?` операции `&` и `|` возвращают не **`null`**, если их результату можно приписать

логическое значение. Т.е. **false** & **null** == **false**, а **true** | **null** == **true**. Также выполнены равенства, получаемые при перестановке операндов в указанных примерах.

Имеется специальный оператор ??, применимый к объектам ссылочных типов или к значениям обнуляемых типов.

Значение a??b равно (a != **null**)?a:b.

Кроме перечисленных разновидностей типов, в C# имеется возможность определять ссылочные типы, являющиеся аналогами указателей на функцию в C — ***делегатные типы (delegate types)***.

Делегатный тип объявляется примерно так же, как абстрактный метод, не имеющий реализации.

Объект делегатного типа можно инициализировать с помощью подходящего по типам параметров и результата метода или с помощью *анонимного метода (anonymous method,* введены в C# 2.0*)* [9].

В приведенном ниже примере объявляется делегатный тип BinaryOperation и 6 объектов этого типа, инициализируемых различными способами.

Объекты op1 и op3 инициализируются при помощи статического метода A.Op1(), объекты op2 и op4 — при помощи метода Op2(), выполняемого в объекте a, объекты op5 и op6 — при помощи анонимных методов.

```
public delegate int BinaryOperation
                      (int x, int y);
public class A
{
  private int x = 0;
  public A(int x) { this.x = x; }

  public static int Op1(int a, int b)
  { return a + b; }

  public int Op2(int a, int b)
  { return x + a + b; }

  public static A a = new A(15);
  BinaryOperation op1 = A.Op1;
  BinaryOperation op2 = a.Op2;
  BinaryOperation op3 =
    new BinaryOperation(A.Op1);
  BinaryOperation op4 =
    new BinaryOperation(a.Op2);
  BinaryOperation op5 =
    delegate(int c, int d)
    { return c * d; };
  BinaryOperation op6 =
    delegate { return 10; };
}
```

По идее, объекты делегатных типов предназначены служить обработчиками некоторых событий. Т.е. при наступлении заданного события надо вызвать соответствующий делегат.

Обработчики событий часто надо изменять в ходе выполнения программы — добавлять в них одни действия и удалять другие.

Поэтому каждый объект-делегат представляет некоторый *список операций (invocation list)*. При этом пустой список представляется как **`null`**.

Добавлять элементы в конец этого списка можно при помощи операторов `+` и `+=`, применяемых к делегатам (и методам, которые неявно преобразуются в объекты делегатного типа, как видно из инициализации `op1` и `op2` в примере). При объединении двух делегатов список операций результата получается конкатенацией списков их операций — список операций правого операнда пристраивается в конце списка операций левого операнда. Списки операций операндов не меняются.

Удалять операции из делегатов можно при помощи операторов `-` и `-=`. При вычитании одного делегата из другого находится последнее вхождение списка операций второго операнда как подсписка в список операций первого операнда. Список операций результата получается как результат удаления этого подсписка из списка операций первого операнда. Если этот список пуст, результат вычитания делегатов равен **`null`**. Если такого подсписка нет, список операций результата совпадает со списком операций первого операнда.

Объект делегатного типа можно вызвать, передав ему набор аргументов. При этом вызываются друг за другом все операции из представляемого им списка. Если объект-делегат равен **`null`**, в результате вызова выбрасывается исключение типа `System.NullReferenceException`.

```
using System;

public class A
{
  delegate void D();

  static void M1()
  { Console.WriteLine("M1 called"); }
  static void M2()
  { Console.WriteLine("M2 called"); }

  public static void Main()
  {
```

```
        D d1 = M1, d2 = M2;
        d1 += M1;
        d2 = d1 + d2 + d1;

        d1();
        Console.WriteLine("------------");
        d2();
        Console.WriteLine("------------");
        (d1 + d2)();
        Console.WriteLine("------------");
        (d1 - d2)();
        Console.WriteLine("------------");
        (d2 - d1)();
    }
}
```

Программа из приведенного примера выдает следующий результат.

```
M1 called
M1 called
------------
M1 called
M1 called
M2 called
M1 called
M1 called
------------
M1 called
M1 called
M1 called
M1 called
M2 called
M1 called
M1 called
------------
M1 called
M1 called
------------
M1 called
M1 called
M2 called
```

В следующей лекции продолжается рассмотрение способов описания пользовательских типов в Java и C#.

Литература к Лекции 10

[1] Страница платформы J2SE http://java.sun.com/j2se/index.jsp.
[2] Страница платформы J2EE http://java.sun.com/j2ee/index.jsp.
[3] Страница платформы J2ME http://java.sun.com/j2me/index.jsp.
[4] Страница платформы Java Card http://java.sun.com/products/javacard/index.jsp.
[5] Страница для разработчиков на .NET http://www.microsoft.com/net/developers.mspx.
[6] Страница проекта Mono http://www.mono-project.com/Main_Page.
[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. Java Language Specification, 3-rd edition. Addison Wesley Professional, 2005.
Доступна как http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf.
[8] C# Language Specification. Working Draft 2.7. ECMA, June 2004.
Доступна как http://download.microsoft.com/download/8/1/6/81682478-4018-48fe-9e5e-f87a44af3db9/standard.pdf.
[9] C# Langauge Specification 2.0, March 2005 Draft.
Доступна как http://download.microsoft.com/download/8/1/6/81682478-4018-48fe-9e5e-f87a44af3db9/CSharp%202.0%20Specification.doc.

[10] The Unicode Consortium. The Unicode Standard, Version 4.0. Boston, MA, Addison-Wesley Developers Press, 2003.
[11] IEEE 754-1985 IEEE Standard for Binary Floating-Point Arithmetic. Revised 1990, IEEE, New York, 1990.
[12] IEC 60559:1989 Binary floating-point arithmetic for microprocessor systems. (2-nd ed., previously designated IEC 559:1989) International Electrotechnical Commission, 1989.

Лекция 11. Основные конструкции языков Java и C# (продолжение)

Аннотация

Продолжается рассмотрение основных конструкций языков Java и C#. Рассказывается о правилах описания связей между типами, определения операций над ними и о создании многопоточных программ. Вкратце рассказывается об основных библиотеках Java и .NET.

Ключевые слова

Наследование типов, перегрузка операций, поле класса, операция, метод, конструктор, инициализатор, константа, свойство, индексированное свойство, событие, оператор, поток, синхронизация потоков.

Текст лекции

В этой лекции мы продолжаем рассмотрение элементов Java 5 и C# 2.0 на основе описывающих их стандартов [1] и [2,3].

Наследование

Отношение вложенности между типами определяется *наследованием*. Обычно говорят, что класс ***наследует*** другому классу или является его ***потомком***, ***наследником***, если он определяет более узкий тип, т.е. все объекты этого класса являются также и объектами наследуемого им. Второй класс в этом случае называют ***предком*** первого. Взаимоотношения между интерфейсами описываются в тех же терминах, но вместо «класс наследует интерфейсу» обычно говорят, что класс ***реализует*** интерфейс.

В обоих языках класс может наследовать только одному классу и реализовывать несколько интерфейсов. Интерфейс может наследовать многим интерфейсам.

Классы, которые не должны иметь наследников, помечаются в Java как **`final`**, а в C# как **`sealed`**.

В Java все классы (но не интерфейсы!) считаются наследниками класса `java.lang.Object`.

Примитивные типы не являются его наследниками, в отличие от своих классов-оберток.

В C# все классы, структурные, перечислимые и делегатные типы (но не интерфейсы!) рассматриваются как наследники класса `System.Object`, на который обычно ссылаются как на **`object`**.
При этом, однако, типы значений (перечислимые и структурные типы, наследники `System.ValueType`) преобразуются к типу **`object`** с помощью упаковки, строящей каждый раз новый объект.
Структурный тип может реализовывать один или несколько интерфейсов, но не может наследовать классу или другому структурному типу.

При наследовании, т.е. сужении типа, возможно определение дополнительных полей и дополнительных операций. Возможно также определение в классе-потомке поля, имеющего то же имя, что и некоторое поле в классе-предке. В этом случае происходит *перекрытие имен* — определяется новое поле, и в коде потомка по этому имени становится доступно только оно.

Если же необходимо получить доступ к соответствующему полю предка, нужно использовать разные подходы в зависимости от того, статическое это поле или нет, т.е. относится ли оно к самому классу или к его объектам. К статическому полю можно обратиться, указав его полное имя, т.е. `ClassName.fieldName`, к нестатическому полю из кода класса-потомка можно обратиться с помощью конструкций **`super`**`.fieldName` в Java и **`base`**`.fieldName` в C# (естественно, если оно не перекрыто в каком-то классе, промежуточном между данными предком и потомком). Конструкции

super в Java и **base** в C# можно использовать и для обращения к операциям, декларированным в предке данного класса. Для обращения к полям и операциям самого объекта в обоих языках можно использовать префикс **this**, являющийся ссылкой на объект, в котором вызывается данная операция.

Основная выгода от использования наследования — возможность ***перегружать (override)*** реализации операций в типах-наследниках. Это значит, что при вызове операции с данной сигнатурой в объекте наследника может быть выполнена не та реализация этой операции, которая определена в предке, а совсем другая, определенная в точном типе объекта. Такие операции называют ***виртуальными (virtual)***. Чтобы определить новую реализацию некоторой виртуальной операции предка в потомке, нужно определить в потомке операцию с той же сигнатурой. При этом необходимо следовать общему принципу, обеспечивающему корректность системы типов в целом — *принципу подстановки (Liskov substitution principle)* [4,5]. Поскольку тип-наследник является более узким, чем тип-предок, его объект может использоваться всюду, где может использоваться объект типа-предка. Принцип подстановки, обеспечивающий это свойство, требует соблюдения двух правил.

- Во всякой ситуации, в которой можно вызвать данную операцию в предке, ее вызов должен быть возможен и в наследнике. Говоря по-другому, предусловие операции при перегрузке не должно усиливаться.
- Множество ситуаций, в которых система в целом может оказаться после вызова операции в наследнике, должно быть подмножеством набора ситуаций, в которых она может оказаться в результате вызова этой операции в предке. То есть, постусловие операции при перегрузке не должно ослабляться.

Статические операции, относящиеся к классу в целом, а не к его объектам, не виртуальны. Они не могут быть перегружены, но могут быть перекрыты, если в потомке определяются статические операции с такими же сигнатурами.

В Java все нестатические методы классов являются виртуальными, т.е. перегружаются при определении метода с такой же сигнатурой в классе-потомке.

Но в Java, в отличие от C#, можно вызывать статические методы и обращаться к статическим полям класса через ссылки на его объекты (в том числе, и через **this**). Поэтому работу невиртуальных методов можно смоделировать с помощью обращений к статическим операциям.

В C# нестатические операции (обычные методы и методы доступа к свойствам, индексерам и событиям) могут быть как виртуальными, т.е. перегружаемыми, так и невиртуальными.

Для декларации виртуального метода (свойства, индексера или события) необходимо пометить его модификатором **virtual**. Метод (свойство, индексер, событие), перегружающий его в классе-потомке надо в этом случае пометить модификатором **override**. Если же мы не хотим перегружать элемент предка, а хотим определить *новый* элемент с такой же сигнатурой (т.е. перекрыть старый), то его надо пометить модификатором **new**.

Элемент, не помеченный как **virtual**, не является перегружаемыми — его можно только перекрыть. При вызове операции с такой сигнатурой в некотором объекте будет вызвана ее реализация, определяемая по декларированному типу объекта.

Приводимые ниже примеры на обоих языках иллюстрируют разницу в работе виртуальных и невиртуальных операций.

```
class A
{
  public void m()
```

```
using System;

class A
{
  public virtual void m()
```

```
  {
    System.out.println("A.m() called");
  }

  public static void n()
  {
    System.out.println("A.n() called");
  }
}

class B extends A
{
  public void m()
  {
    System.out.println("B.m() called");
  }

  public static void n()
  {
    System.out.println("B.n() called");
  }
}

public class C
{
  public static void main(String[] args)
  {
    A a = new A();
    B b = new B();
    A c = new B();

    a.m();
    b.m();
    c.m();
    System.out.println("-----");
    a.n();
    b.n();
    c.n();
  }
}
```

Представленный в примере код выдает следующие результаты.

```
A.m() called
B.m() called
B.m() called
-----
A.n() called
B.n() called
A.n() called
```

```
  {
    Console.WriteLine("A.m() called");
  }

  public void n()
  {
    Console.WriteLine("A.n() called");
  }
}

class B : A
{
  public override void m()
  {
    Console.WriteLine("B.m() called");
  }

  public new void n()
  {
    Console.WriteLine("B.n() called");
  }
}

public class C
{
  public static void Main()
  {
    A a = new A();
    B b = new B();
    A c = new B();

    a.m();
    b.m();
    c.m();
    Console.WriteLine("-----");
    a.n();
    b.n();
    c.n();
  }
}
```

Если в приведенном примере убрать модификатор **new** у метода `n()` в классе `B`, ошибки компиляции не будет, но будет выдано предупреждение о перекрытии имен, возможно случайном.

Представленный в примере код выдает следующие результаты.

```
A.m() called
B.m() called
B.m() called
-----
A.n() called
B.n() called
A.n() called
```

Элементы типов

Элементы или *члены (members)* пользовательских типов могут быть методами, полями (в классах) и вложенными типами. В классе можно также объявлять конструкторы, служащие для создания объектов этого класса. В обоих языках описание конструктора похоже на описание метода, только тип результата не указывается, а вместо имени метода используется имя самого класса.

В обоих языках поля можно только перекрывать в наследниках, а методы можно и перегружать. Вложенные типы, как и поля, могут быть перекрыты.

У каждого элемента класса могут присутствовать модификаторы, определяющие доступность этого элемента из разных мест программы, а также его *контекст* — относится ли он к объектам этого класса (нестатический элемент) или к самому классу (статический элемент, помечается как **`static`**).

Для указания доступности в обоих языках могут использоваться модификаторы **`public`**, **`protected`** и **`private`**, указывающие, соответственно, что данный элемент доступен везде, где доступен содержащий его тип, доступен только в описаниях типов-наследников содержащего типа, или только в рамках описания самого содержащего типа. Доступность по умолчанию, без указания модификатора трактуется в рассматриваемых языках различно.

Нестатические методы в обоих языках (а также свойства, индексированные свойства и события в C#) могут быть объявлены *абстрактными (abstract)*, т.е. не задающими реализации соответствующей операции. Такие методы (а также свойства, индексированные свойства и события в C#) помечаются модификатором **`abstract`**. Вместо кода у абстрактного метода сразу после описания полной сигнатуры идет точка с запятой.

Методы (свойства, индексированные свойства и события в C#), которые не должны быть перегружены в наследниках содержащего их класса, помечаются в Java как **`final`**, а в C# как **`sealed`**.

В обоих языках можно использовать операции, реализованные на других языках. Для этого в C# используются стандартные механизмы .NET — класс реализуется на одном из языков, поддерживаемых .NET, с учетом ограничений на общие библиотеки этой среды и становится доступен из любого другого кода на поддерживаемом .NET языке.

В Java для этого предусмотрен механизм Java Native Interface, JNI [6,7]. Класс Java может иметь ряд внешних методов, помеченных модификатором **`native`**. Вместо кода у таких методов сразу после описания полной сигнатуры идет точка с запятой. Они по определенным правилам реализуются в виде функций на языке C (или на другом языке, если можно в результате компиляции получить библиотеку с интерфейсом на C). Внешние методы, а также свойства, индексированные свойства, события и операторы, привязываемые по определенным правилам к функциям, которые имеют интерфейс на C и не вложены в среду .NET, есть и в C# — там такие операции помечаются как **`extern`**.

В Java, помимо перечисленных членов типов, имеются *инициализаторы*. Их описание приведено ниже.

Инициализаторы относятся только к тому классу, в котором они определены, их нельзя перегрузить.

В C# члены типов, помимо методов, полей, конструкторов и вложенных типов, могут быть *константами*, *свойствами*, *индексированными свойствами*, *событиями* или *операторами*. Кроме этого, в типе можно определить *деструктор* и *статический конструктор*.

Нестатические свойства, индексированные свойства и события можно перегружать в наследниках. Остальные из перечисленных элементов относятся только к тому классу, в котором описаны.

Для многих из дополнительных разновидностей членов типов, имеющихся в C#, есть аналогичные идиомы в компонентной модели JavaBeans [8,9], предназначенной для построения элементов пользовательского интерфейса и широко используемой в рамках Java технологий для создания компонентов, структуру которых можно анализировать динамически на основе предлагаемых JavaBeans соглашений об именовании методов. Далее вместе с примерами кода на C# в правом столбце в левом приводится аналогичный код, написанный в соответствии JavaBeans.

Константы в Java принято оформлять в виде полей с модификаторами **`final static`**. Модификатор **`final`** для поля означает, что

Константы являются единожды вычисляемыми и неизменными далее значениями, хранящимися в классе или

присвоить ему значение можно только один раз и сделать это нужно либо в статическом инициализаторе класса (см. ниже), если поле статическое, либо в каждом из конструкторов, если поле нестатическое.

```
public class A2
{
  public static final double PHI =
    1.61803398874989;
}
```

Компонентная модель JavaBeans определяет ***свойство (property)*** класса `A`, имеющее имя *`name`* и тип *`T`*, как набор из одного или двух методов, декларированных в классе `A` — *`T`* `get`*`Name`*`()` и `void set`*`Name`*`(`*`T`*`)`, называемых ***методами доступа (accessor methods)*** к свойству.

Свойство может быть доступным только для чтения, если имеется лишь метод `get`, и только для записи, если имеется лишь метод `set`.

Если свойство имеет логический тип, для метода чтения этого свойства используется имя `is`*`Name`*`()`.

Эти соглашения широко используются в разработке Java программ, и такие свойства описываются не только у классов, предназначенных стать компонентами JavaBeans.

Они и стали основанием для введения специальной конструкции для описания свойств в C#.

```
public class MyArrayList
{
  private int[] items = new int[10];
  private int size = 0;

  public int getSize()
  {
    return size;
  }

  public int getCapacity()
  {
    return items.Length;
  }

  public void setCapacity(int value)
  {
    int[] newItems = new int[value];
    System.arraycopy
      (items, 0, newItems, 0, size);
    items = newItems;
  }

  public static void main(String[] args)
  {
```

структуре.

Пример объявления константы приведен ниже.

```
public class A2
{
  public const double Phi =
    1.61803398874989;
}
```

Свойства (properties) представляют собой «виртуальные» поля. Каждое свойство имеет один или оба *метода доступа* **`get`** и **`set`**, которые определяют действия, выполняемые при чтении и модификации этого свойства. Оба метода доступа описываются внутри декларации свойства. Метод **`set`** использует специальный идентификатор **`value`** для ссылки на устанавливаемое значение свойства.

Обращение к свойству — чтение (возможно, только если у него есть метод **`get`**) или изменение значения свойства (возможно, только если у него есть метод **`set`**) — происходит так же, как к полю.

При перегрузке свойства в наследниках перегружаются методы доступа к нему.

Пример объявления свойств и их использования приведен ниже.

```
using System;

public class MyArrayList
{
  private int[] items = new int[10];
  private int size = 0;

  public int Size
  {
    get { return size; }
  }

  public int Capacity
  {
    get { return items.Length; }
    set
    {
      int[] newItems = new int[value];
      Array.Copy
        (items, newItems, size);
      items = newItems;
    }
  }

  public static void Main()
  {
```

```
    MyArrayList l = new MyArrayList();
    System.out.println(l.getSize());
    System.out.println(l.getCapacity());
    l.setCapacity(50);
    System.out.println(l.getSize());
    System.out.println(l.getCapacity());
  }
}
```

JavaBeans определяет ***индексированное свойство (indexed property)*** класса `A`, имеющее имя *`name`* и тип *`T`*, как один или пару методов *`T`* `get`*`Name`*`(int)` и `void set`*`Name`*`(int, `*`T`*`)`.

Свойства могут быть индексированы только одним целым числом. В дальнейшем предполагалось ослабить это ограничение и разрешить индексацию несколькими параметрами, которые могли бы иметь разные типы. Однако с 1997 года, когда появилась последняя версия спецификаций JavaBeans [9], этого пока сделано не было.

```
public class MyArrayList
{
  int[] items = new int[10];
  int size = 0;

  public int getItem(int i)
  {
    if (i < 0 || i >= 10) throw new
       IllegalArgumentException();
    else return items[i];
  }


  public void setItem(int i, int value)
  {
    if (i < 0 || i >= 10) throw new
       IllegalArgumentException();
    else items[i] = value;
  }


  public static void main(String[] args)
  {
    MyArrayList l = new MyArrayList();
    l.setItem(0, 23);
    l.setItem(1, 75);
    l.setItem(1, l.getItem(1)-1);
    l.setItem(0,
```

```
    MyArrayList l = new MyArrayList();
    Console.WriteLine( l.Size );
    Console.WriteLine( l.Capacity );
    l.Capacity = 50;
    Console.WriteLine( l.Size );
    Console.WriteLine( l.Capacity );
  }
}
```

Индексированное свойство или ***индексер (indexer)*** — это свойство, зависящее от набора параметров.

В C# может быть определен только один индексер для типа и данного набора типов параметров. Т.е. нет возможности определять свойства с разными именами, но одинаковыми наборами индексов.

Обращение к индексеру объекта (или класса, т.е. статическому) производится так, как будто этот объект (класс) был бы массивом, индексированным набором индексов соответствующих типов.

При перегрузке индексера в наследниках перегружаются методы доступа к нему.

Индексеры должны быть нестатическими.

Обращение к индексеру класса-предка в индексере наследника организуется с помощью конструкции **`base`**`[…]`.

Пример декларации и использования индексера приведен ниже.

```
using System;

public class MyArrayList
{
  int[] items = new int[10];
  int size = 0;

  public int this[int i]
  {
    get
    {
      if (i < 0 || i >= 10) throw new
         IndexOutOfRangeException();
      else return items[i];
    }
    set
    {
      if (i < 0 || i >= 10) throw new
         IndexOutOfRangeException();
      else items[i] = value;
    }
  }

  public static void Main()
  {
    MyArrayList l = new MyArrayList();
    l[0] = 23;
    l[1] = 75;
    l[0] += (--l[1]);
    Console.WriteLine(l[0]);
```

```
        l.getItem(0) + l.getItem(1));
    System.out.println (l.getItem(0));
    System.out.println (l.getItem(1));
  }
}
```

События (events) в модели JavaBeans служат для оповещения набора *объектов-наблюдателей (listeners)* о некоторых изменениях в состоянии *объекта-источника (source)*.

При этом класс *EventType* объектов, представляющих события определенного вида, должен наследовать `java.util.EventObject`. Все объекты-наблюдатели должны реализовывать один интерфейс *EventListener*, в котором должен быть метод обработки события (обычно называемый так же, как и событие) с параметром типа *EventType*. Интерфейс *EventListener* должен наследовать интерфейсу `java.util.EventListener`.

Класс источника событий должен иметь методы для регистрации наблюдателей и их удаления из реестра. Эти методы должны иметь сигнатуры

public void add*EventListener*
(*EventListener*)
public void remove*EventListener*
(*EventListener*).

Можно заметить, что такой способ реализации обработки событий воплощает образец проектирования «Подписчик».

В приведенном ниже примере все **public** классы и интерфейсы должны быть описаны в разных файлах.

```
public class MouseEventArgs { ... }
```

```
public class MouseEventObject
  extends java.util.EventObject
{
  MouseEventArgs args;

  MouseEventObject
  (Object source, MouseEventArgs args)
  {
    super(source);
    this.args = args;
  }
}
```

```
public interface MouseEventListener
  extends java.util.EventListener
{
  void mouseUp(MouseEventObject e);
  void mouseDown(MouseEventObject e);
}
```

```
import java.util.ArrayList;
```

```
public class MouseEventSource
```

```
    Console.WriteLine(l[1]);
  }
}
```

Событие (event) представляет собой свойство специального вида, имеющее делегатный тип. У события, в отличие от обычного свойства, методы доступа называются **add** и **remove** и предназначены они для добавления или удаления обработчиков данного события, являющихся делегатами (это аналоги различных реализаций метода обработки события в интерфейсе наблюдателя в JavaBeans) при помощи операторов += и -=.

Событие может быть реализовано как поле делегатного типа, помеченное модификатором event. В этом случае декларировать соответствующие методы **add** и **remove** необязательно — они автоматически реализуются при применении операторов += и -= в к этому полю как к делегату.

Если же программист хочет реализовать какое-то специфическое хранение обработчиков события, он должен определить методы **add** и **remove**.

В приведенном ниже примере одно из событий реализовано как событие-поле, а другое — при помощи настоящего поля и методов **add** и **remove**, дающих совместно тот же результат.

При перегрузке события в наследниках перегружаются методы доступа к нему.

```
public class MouseEventArgs { ... }
```

```
public delegate void MouseEventHandler
  (object source, MouseEventArgs e);
```

```
public class MouseEventSource
```

```
{
  private ArrayList<MouseEventListener>
    listeners = new ArrayList
    <MouseEventListener >();


  public synchronized void
  addMouseEventListener
    (MouseEventListener l)
  { listeners.add(l); }

  public synchronized void
  removeMouseEventListener
    (MouseEventListener l)
  { listeners.remove(l); }


 protected void notifyMouseUp
    (MouseEventArgs a)
  {
    MouseEventObject e =
      new MouseEventObject(this, a);
    ArrayList<MouseEventListener> l;
    synchronized(this)
    {
      l =
      (ArrayList<MouseEventListener>)
        listeners.clone();
      for(MouseEventListener el : l)
        el.mouseUp(e);
    }
  }

  protected void notifyMouseDown
    (MouseEventArgs a)
  {
    MouseEventObject e =
      new MouseEventObject(this, a);

    ArrayList<MouseEventListener> l;
    synchronized(this)
    {
      l =
       (ArrayList<MouseEventListener>)
        listeners.clone();
      for(MouseEventListener el : l)
        el.mouseDown(e);
    }
  }
}

public class HandlerConfigurator
{
  MouseEventSource s =
    new MouseEventSource();

  MouseEventListener listener =
    new MouseEventListener()
  {
    public void mouseUp
     (MouseEventObject e) { ... }
    public void mouseDown
     (MouseEventObject e) { ... }
  };
```

```
{
  public event MouseEventHandler MouseUp;

  private MouseEventHandler mouseDown;
  public event
    MouseEventHandler MouseDown
  {
    add
    {
      lock(this)
      { mouseDown += value; }
    }
    remove
    {
      lock(this)
      { mouseDown -= value; }
    }
  }

  protected void
    OnMouseUp(MouseEventArgs e)
  {
    MouseUp(this, e);
  }

  protected void
    OnMouseDown(MouseEventArgs e)
  {
    mouseDown(this, e);
  }
}

public class HandlerConfigurator
{
  MouseEventSource s =
    new MouseEventSource();

  public void UpHandler
    (object source, MouseEventArgs e)
  { ... }
  public void DownHandler
    (object source, MouseEventArgs e)
  { ... }
```

```
  public void configure()
  {
    s.addMouseEventListener(listener);
  }
}
```

```
  public void Configure()
  {
    s.MouseUp += UpHandler;
    s.MouseDown += DownHandler;
  }
}
```

Методы доступа к свойствам, индексерам или событиям в классах-наследниках могут перегружаться по отдельности, т.е., например, метод чтения свойства перегружается, а метод записи — нет.

В C# 2.0 введена возможность декларации различной доступности у таких методов. Например, метод чтения свойства можно сделать общедоступным, а метод записи — доступным только для наследников.

Для этого можно описать свойство так:

```
public int Property
{
  get { … }
  protected set { … }
}
```

В Java никакие операторы переопределить нельзя.

Вообще, в этом языке имеются только операторы, действующие на значениях примитивных типах, сравнение объектов на равенство и неравенство, а также оператор + для строк (это объекты класса `java.lang.String`), обозначающий операцию конкатенации.

Оператор + может применяться и к другим типам аргументов, если один из них имеет тип `String`. При этом результатом соответствующей операции является конкатенация его и результата применения метода `toString()` к другому операнду в порядке следования операндов.

Некоторые операторы в C# можно переопределить (перекрыть) для данного пользовательского типа. Переопределяемый оператор всегда имеет модификатор **`static`**.

Переопределяемые унарные операторы (их единственный параметр должен иметь тот тип, в рамках которого они переопределяются, или *объемлющий тип*): +, -, !, ~ (в качестве типа результата могут иметь любой тип), ++,-- (тип их результата может быть только подтипом объемлющего), **`true`**, **`false`** (тип результата **`bool`**).

Переопределяемые бинарные операторы (хотя бы один из их параметров должен иметь объемлющий тип, а возвращать они могут результат любого типа): +, -, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=. Для операторов сдвига << и >> ограничения более жесткие — первый их параметр должен иметь объемлющий тип, а второй быть типа **`int`**.

Можно определять также операторы приведения к другому типу или приведения из другого типа, причем можно объявить такое приведение неявным с помощью модификатора **`implicit`**, чтобы компилятор сам вставлял его там, где оно необходимо для соблюдения правил соответствия типов. Иначе надо использовать модификатор **`explicit`** и всегда явно приводить один тип к другому.

Некоторые операторы можно определять только парами — таковы **`true`** и **`false`**, == и !=, < и >,

<= и >=.

Операторы **true** и **false** служат для неявного преобразования объектов данного типа к соответствующим логическим значениям.

Если в типе `T` определяются операторы & и |, возвращающие значение этого же типа, а также операторы **true** и **false**, то к объектам типа можно применять условные логические операторы `&&` и `||`. Их поведение в этом случае может быть описано соотношениями `(x && y) = (T.false(x)? x : (x & y))` и `(x || y) = (T.true(x)? x : (x | y))`.

Аналогичным образом автоматически переопределяются составные операторы присваивания, если переопределены операторы +, -, *, /, %, &, |, ^, << или >>.

Ниже приведен пример переопределения и использования операторов. Обратите внимание, что оператор приведения типа `MyInt` в **int** действует неявно, а для обратного перехода требуется явное приведение.

Другая тонкость — необходимость приведения объектов типа `MyInt` к **object** в методе `AreEqual` — если этого не сделать, то при обращении к оператору == возникнет бесконечный цикл, поскольку сравнение `i1 == null` тоже будет интерпретироваться как вызов этого оператора.

```
using System;

public class MyInt
{
  int n = 0;

  public MyInt(int n) { this.n = n; }

  public override bool Equals(object obj)
  {
    MyInt o = obj as MyInt;
    if (o == null) return false;
    return o.n == n;
  }

  public override int GetHashCode()
  { return n; }

  public override string ToString()
  { return n.ToString(); }

  public static bool AreEqual
    (MyInt i1, MyInt i2)
  {
    if ((object)i1 == null)
      return ((object)i2 == null);
    else return i1.Equals(i2);
  }
```

```
  public static bool operator ==
    (MyInt i1, MyInt i2)
  { return AreEqual(i1, i2); }

  public static bool operator !=
    (MyInt i1, MyInt i2)
  { return !AreEqual(i1, i2); }

  public static bool operator true
    (MyInt i)
  { return i.n > 0; }

  public static bool operator false
    (MyInt i)
  { return i.n <= 0; }

  public static MyInt operator ++
    (MyInt i)
  { return new MyInt(i.n + 1); }

  public static MyInt operator --
    (MyInt i)
  { return new MyInt(i.n - 1); }

  public static MyInt operator &
    (MyInt i1, MyInt i2)
  { return new MyInt(i1.n & i2.n); }

  public static MyInt operator |
    (MyInt i1, MyInt i2)
  { return new MyInt(i1.n | i2.n); }

  public static implicit operator int
    (MyInt i)
  { return i.n; }

  public static explicit operator
    MyInt (int i)
  { return new MyInt(i); }

  public static void Main()
  {
    MyInt n = (MyInt)5;
    MyInt k = (MyInt)(n - 3 * n);
    Console.WriteLine("k = " + k +
      " , n = " + n);
    Console.WriteLine("n == n : " +
      (n == n));
    Console.WriteLine("n == k : " +
      (n == k));
    Console.WriteLine(
      "(++k) && (n++) : " +
      ((++k) && (n++)));
    Console.WriteLine("k = " + k +
      " , n = " + n);
    Console.WriteLine(
      "(++n) && (k++) : " +
       ((++n) && (k++)));
    Console.WriteLine("k = " + k +
      " , n = " + n);
  }
}
```

Аналогом деструктора в Java является метод **protected void** `finalize()`, который можно

Деструктор предназначен для освобождения каких-либо ресурсов, связанных с объектом и не освобождаемых автоматически средой .NET,

перегрузить для данного класса.

Так же, как и деструктор в C#, этот метод вызывается на некотором шаге уничтожения объекта после того, как тот был помечен сборщиком мусора как неиспользуемый.

либо для оптимизации использования ресурсов за счет их явного освобождения.

Деструктор вызывается автоматически при уничтожении объекта в ходе работы механизма управления памятью .NET. В этот момент объект уже должен быть помечен сборщиком мусора как неиспользуемый.

Деструктор оформляется как особый метод, без возвращаемого значения и с именем, получающимся добавлением префикса ‘~’ к имени класса

```
public class MyFileReader
{
  java.io.FileReader input;

  public MyFileReader(String path)
    throws FileNotFoundException
  {
    input = new java.io.FileReader
     (new java.io.File(path));
  }

  protected void finalize()
  {
    System.out.println("Destructor");
    try { input.close(); }
    catch (IOException e)
    {
      e.printStackTrace();
    }
  }
}
```

```
using System;

public class MyFileStream
{
  System.IO.FileStream input;

  public MyFileStream(string path)
  {
    input = System.IO.File.Open
      (path, System.IO.FileMode.Open);
  }


  ~MyFileStream()
  {
    Console.WriteLine("Destructor");
    input.Close();
  }
}
```

Инициализаторы представляют собой блоки кода, заключенные в фигурные скобки и расположенные непосредственно внутри декларации класса.

Эти блоки выполняются вместе с инициализаторами отдельных полей — выражениями, которые написаны после знака = в объявлениях полей — при построении объекта данного класса, в порядке их расположения в декларации.

Статические инициализаторы — такие же блоки, помеченные модификатором **static** — выполняются вместе с инициализаторами статических полей по тем же правилам в момент первой загрузки класса в Java-машину.

Статический конструктор класса представляет собой блок кода, выполняемый при первой загрузке класса в среду .NET, т.е. в момент первого использования этого класса в программе. Это аналог статического инициализатора в Java.

Оформляется он как конструктор с модификатором **static**.

```
public class A
{
  static
```

```
using System;

public class A
{
  static A()
```

```
  {
    System.out.println("Loading A");
  }

  static int x = 1;

  static
  {
    System.out.println("x = " + x);
    x++;
  }

  static int y = 2;

  static
  {
    y = x + 3;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }

  public static void main(String[] args)
  {}
}
```

Приведенный выше код выдает результат

```
Loading A
x = 1
x = 2
y = 5
```

В Java нестатические вложенные типы трактуются очень специфическим образом — каждый объект такого типа считается привязанным к определенному объекту объемлющего типа. У нестатического вложенного типа есть как бы необъявленное поле, хранящее ссылку на объект объемлющего типа.

Такая конструкция используется, например, для определения классов итераторов для коллекций — объект-итератор всегда связан с объектом-коллекцией, которую он итерирует. В то же время, пользователю не нужно знать, какого именно типа данный итератор, — достаточно, что он реализует общий интерфейс всех итераторов, позволяющий проверить, есть ли еще объекты, и получить следующий объект.

Получить этот объект внутри декларации вложенного типа можно с помощью конструкции `ClassName.this`, где `ClassName` — имя объемлющего типа.

При создании объекта такого вложенного класса необходимо указать объект объемлющего класса, к которому тот будет привязан.

```
public class ContainingClass
{
  static int counter = 1;
  static int ecounter = 1;
```

```
  {
    Console.WriteLine("Loading A");
    Console.WriteLine("x = " + x);
    x++;
    y = x + 3;
    Console.WriteLine("x = " + x);
    Console.WriteLine("y = " + y);
  }

  static int x = 1;

  static int y = 2;

  public static void Main() {}
}
```

Приведенный выше код выдает результат

```
Loading A
x = 1
x = 2
y = 5
```

В C# модификатор **`static`** у класса, все равно, вложенного в другой или нет, обозначает, что этот класс является контейнером набора констант и статических операций. Все его элементы должны быть декларированы как **`static`**.

```
  int id = counter++;

  class EmbeddedClass
  {
    int eid = ecounter++;

    public String toString()
    {
      return "" +
        ContainingClass.this.id +
        '.' + eid;
    }
  }

  public String toString()
  {
    return "" + id;
  }

  public static void main
    (String[] args)
  {
    ContainingClass
       c  = new ContainingClass()
     , c1 = new ContainingClass();
    System.out.println(c);
    System.out.println(c1);

    EmbeddedClass
        e  = c.new EmbeddedClass()
      , e1 = c.new EmbeddedClass()
      , e2 = c1.new EmbeddedClass();
    System.out.println(e);
    System.out.println(e1);
    System.out.println(e2);
  }
}
```

В C# класс может определить различные реализации для операций (методов, свойств, индексеров, событий) с одинаковой сигнатурой, если они декларированы в различных реализуемых классом интерфейсах.

Для этого при определении таких операций нужно указывать имя интерфейса в качестве расширения их имени.

```
using System;

public interface I1
{
  void m();
}

public interface I2
{
  void m();
}

public class A : I1, I2
{
  public void m()
  {
    Console.WriteLine("A.m() called");
  }
```

```
  void I1.m()
  {
    Console.WriteLine
      ("I1.m() defined in A called");
  }

  void I2.m()
  {
    Console.WriteLine
     ("I2.m() defined in A called");
  }

  public static void Main()
  {
    A f = new A();
    I1 i1 = f;
    I2 i2 = f;

    f.m();
    i1.m();
    i2.m();
  }
}
```

Результат работы приведенного выше примера следующий.

```
A.m() called
I1.m() defined in A called
I2.m() defined in A called
```

Последовательность выполнения инициализаторов полей и конструкторов классов-предков и наследников при построении объектов в Java и C# различается достаточно сильно.

О правилах, определяющих эту последовательность, можно судить по результатам работы следующих примеров.

```
public class A
{
  static
  {
    System.out.println
      ("Static initializer of A");
  }

  {
    System.out.println
      ("Initializer of A");
  }

  static int sinit()
  {
    System.out.println
    ("Static field initializer of A");
    return 0;
  }

  static int init()
  {
    System.out.println
      ("Field initializer of A");
    return 0;
  }

  static int sf = sinit();
```

```
using System;

public class A
{
  static A()
  {
    Console.WriteLine
      ("Static constructor of A");
  }

  static int sinit()
  {
    Console.WriteLine
    ("Static field initializer of A");
    return 0;
  }

  static int init()
  {
    Console.WriteLine
      ("Field initializer of A");
    return 0;
  }

  static int sf = sinit();
```

```
  int f = init();

  public A()
  {
    System.out.println
      ("Constructor of A");
  }
}

public class B extends A
{
  static int sf = sinit();
  int f = init();

  static
  {
    System.out.println
      ("Static initializer of B");
  }

  {
    System.out.println
      ("Initializer of B");
  }

  static int sinit()
  {
    System.out.println
    ("Static field initializer of B");
    return 0;
  }

  static int init()
  {
    System.out.println
      ("Field initializer of B");
    return 0;
  }

  public B()
  {
    System.out.println
      ("Constructor of B");
  }
}

public class C
{
  public static void main(String[] args)
  {
    B b = new B();
  }
}
```

Результат работы приведенного примера такой.

```
Static initializer of A
Static field initializer of A
Static field initializer of B
Static initializer of B
Initializer of A
Field initializer of A
Constructor of A
Field initializer of B
Initializer of B
Constructor of B
```

```
  int f = init();

  public A()
  {
    Console.WriteLine
      ("Constructor of A");
  }
}

public class B : A
{
  static int sf = sinit();
  int f = init();

  static B()
  {
    Console.WriteLine
      ("Static constructor of B");
  }

  static int sinit()
  {
    Console.WriteLine
    ("Static field initializer of B");
    return 0;
  }

  static int init()
  {
    Console.WriteLine
      ("Field initializer of B");
    return 0;
  }

  public B()
  {
    Console.WriteLine
      ("Constructor of B");
  }
}

public class C
{
  public static void Main()
  {
    B b = new B();
  }
}
```

Результат работы приведенного примера такой.

```
Static field initializer of B
Static constructor of B
Field initializer of B
Static field initializer of A
Static constructor of A
Field initializer of A
Constructor of A
Constructor of B
```

В Java элемент типа, помимо доступности, указываемой с помощью модификаторов **`private`**, **`protected`** и **`public`**, может иметь пакетную доступность. Такой элемент может использоваться в типах того же пакета, к которому относится содержащий его тип.

Именно пакетная доступность используется в Java по умолчанию.

`protected` элементы в Java также доступны из типов того пакета, в котором находится содержащий эти элементы тип, т.е. **`protected`**-доступность шире пакетной.

Типы, не вложенные в другие, могут быть либо **`public`** (должно быть не более одного такого типа в файле), либо иметь пакетную доступность.

В C# доступность элемента типа по умолчанию — **`private`**.

Имеется дополнительный модификатор доступности — **`internal`**. Элемент, имеющий такую доступность, может быть использован всюду в рамках того же файла, где находится определение этого элемента.

Кроме того, в C# можно использовать комбинированный модификатор **`protected internal`** для указания того, что к данному элементу имеют доступ как наследники содержащего его типа, так и типы в рамках содержащего его файла.

Типы, не вложенные в другие, могут быть либо **`public`**, либо иметь доступность по умолчанию — **`private`**, что означает, что они могут использоваться только в рамках содержащего их пространства имен.

В C# все элементы типов могут быть помечены модификатором **`new`** для точного указания на то, что такой элемент — новый и никак не соотносится с элементами предков данного типа с тем же именем или той же сигнатурой.

В Java для полей классов дополнительно к модификаторам доступности и контекста могут использоваться модификаторы **`final`**, **`transient`** и **`volatile`**.

Модификатор **`final`** обозначает, что такое поле не может быть изменено во время работы, но сначала должно быть инициализировано: статическое — в одном из статических инициализаторов, нестатическое — к концу работы каждого из конструкторов. В инициализаторах или конструкторах такое поле может модифицироваться несколько раз.

Модификатор **`final`** у локальных переменных и параметров методов может использоваться примерно в том же значении — невозможность модификации их значений после инициализации.

Поля, помеченные модификатором **`transient`**, считаются не входящими в состояние объекта или класса, подлежащее хранению или передаче по сети.

В Java имеются соглашения о том, как должен быть оформлен интерфейс класса, объекты которого могут быть сохранены или переданы по сети — эти соглашения можно найти в документации по интерфейсу `java.io.Serializable`, который должен реализовываться таким классом. Имеются и другие подобные наборы соглашений,

Поля классов или структурных типов в C# могут иметь в качестве дополнительных модификаторов **`readonly`** и **`volatile`**.

Модификатор **`readonly`** по своему значению аналогичен модификатору **`final`** в Java. **`readonly`** поля должны инициализироваться к концу работы конструкторов и дальше не могут быть изменены.

Такие поля используются для представления постоянных в ходе работы программы объектов и значений, типы которых не допустимы для поддерживаемых языком констант, а также таких, значение которых не может быть вычислено во время компиляции.

привязанные к определенным библиотекам или технологиям в рамках Java.

Стандартный механизм Java, обеспечивающий сохранение и восстановление объектов, реализующих интерфейс `java.io.Serializable`, по умолчанию сохраняет значения всех полей, кроме помеченных модификатором **`transient`**.

Методы классов в Java могут быть дополнительно помечены модификаторами **`strictfp`** (такой же модификатор могут иметь инициализаторы) и **`synchronized`**.

Значение модификатора **`strictfp`** описано в Лекции 10, в разделе о типах с плавающей точкой.

Значение модификатора **`synchronized`** описывается ниже, в разделе, посвященном многопоточным приложениям.

Шаблонные типы и операции

В последних версиях обоих языков введены шаблонные, т.е. имеющие типовые параметры, типы и операции.

Ниже приводятся примеры декларации шаблонного метода и его использования в Java и C#. В последнем вызове в обоих примерах явное указание типового аргумента у метода `getTypeName()` необязательно, поскольку он вычисляется из контекста вызова. Если вычислить типовые аргументы вызова метода нельзя, их нужно указывать явно.

```
public class A
{
  public static <T> String getTypeName
    (T a)
  {
    if(a == null) return "NullType";
    else return
      a.getClass().getName();
  }

  public static void main(String[] args)
  {
    String  y = "ABCDEFG";

    System.out.println( getTypeName(y) );
    System.out.println
      ( getTypeName(y.length()) );
    System.out.println
      ( A.<Character>getTypeName
          (y.charAt(1)) );
  }
}
```

В Java в качестве типовых аргументов могут использоваться только ссылочные типы.

Примитивный тип не может быть аргументом шаблона — вместо него нужно использовать соответствующий класс-обертку.

```
using System;

public class A
{
  public static string getTypeName<T>
    (T a)
  {
    if(a == null) return "NullType";
    else return
      a.GetType().FullName;
  }

  public static void Main()
  {
    string y = "ABCDEFG";

    Console.WriteLine( getTypeName(y) );
    Console.WriteLine
      ( getTypeName(y.Length) );
    Console.WriteLine
      ( getTypeName<char>(y[1]) );
  }
}
```

В C# любой тип может быть аргументом шаблона.

В Java типовые аргументы являются элементами конкретного объекта — они фактически представляют собой набор дополнительных параметров конструктора объекта или метода, если речь идет о шаблонном методе. Поэтому статические элементы шаблонного типа являются общими для всех экземпляров этого типа с разными типовыми аргументами.

```
public class A<T>
{
  public static int c = 0;
  public T t;
}

public class B
{
  public static void main
    (String[] args)
  {
    A.c = 7;

    System.out.println( A.c );
  }
}
```

В C# каждый экземпляр шаблонного класса, интерфейса или структурного типа с определенными аргументами имеет свой набор статических элементов, которые являются общими для всех объектов такого полностью определенного типа.

```
using System;

public class A<T>
{
  public static int c = 0;
  public T t;
}

public class B
{
  public static void Main()
  {
    A<string>.c = 7;

    Console.WriteLine( A<int>.c );
    Console.WriteLine( A<string>.c );
  }
}
```

В C# можно определить и использовать шаблонные делегатные типы.

```
public delegate bool Predicate<T>
  (T value);

public class I
{
  public bool m(int i)
  { return i == 0; }

  public void f()
  {
    Predicate<int> pi = m;
    Predicate<string> ps =
      delegate(string s)
      { return s == null; };
  }
}
```

В обоих языках имеются конструкции для указания ограничений на типовые параметры шаблонных типов и операций. Такие ограничения позволяют избежать ошибок, связанных с использованием операций типа-параметра, точнее, позволяют компилятору обнаруживать такие ошибки.

Ограничения, требующие от типа-параметра наследовать некоторому другому типу, позволяют использовать операции и данные типа-параметра в коде шаблона.

В Java можно указать, что тип-параметр данного шаблона должен быть наследником некоторого класса и/или реализовывать определенные интерфейсы.

В приведенном ниже примере параметр `T` должен наследовать классу `A` и реализовывать интерфейс `B`.

В C# можно указать, что тип-параметр должен быть ссылочным, типом значения, наследовать определенному классу и/или определенным интерфейсам, а также иметь конструкторы с заданной сигнатурой.

В приведенном ниже примере параметр `T` должен быть ссылочным типом, параметр `V` —

```
public class A
{

  public int m() { ... }
}

public interface B
{
  public String n();
}

public class C<T extends A & B>
{
  T f;

  public String k()
  {
    return f.n() + (f.m()*2);
  }
}
```

Кроме того, в Java можно использовать *неопределенные типовые параметры (wildcards)* при описании типов. Неопределенный типовой параметр может быть ограничен требованием наследовать определенному типу или, наоборот, быть предком определенного типа.

Неопределенные типовые параметры используют в тех случаях, когда нет никаких зависимостей между этими параметрами, между ними и типами полей, типами результатов методов и исключений. В таких случаях введение специального имени для типового параметра не требуется, поскольку оно будет использоваться только в одном месте — при описании самого этого параметра.

В приведенном ниже примере первый метод работает с коллекцией произвольных объектов, второй — с коллекцией объектов, имеющих (не обязательно точный) тип `T`, третий — с такой коллекцией, в которую можно добавить элемент типа `T`.

```
public class A
{
  public void addAll
    (Collection<?> c)
  { ... }

  public <T> void addAll
    (Collection<? extends T> c)
  { ... }

  public <T> void addToCollection
    (T e, Collection<? super T> c)
  { ... }
}
```

типом значений, а параметр `U` — наследовать классу `A`, реализовывать интерфейс `IList<T>` и иметь конструктор без параметров.

```
public class A { ... }


public class B<T, U, V>
  where T : class
  where U : A, IList<T>, new()
  where V : struct
{ ... }
```

Дополнительные элементы описания операций

В обоих языках (в Java — начиная с версии 5) имеются конструкции, позволяющие описывать операции с неопределенным числом параметров (как в функции `printf` стандартной библиотеки C). Для этого последний параметр нужно пометить специальным образом. Этот параметр интерпретируется как массив значений указанного типа. При вызове такой операции можно указать обычный массив в качестве ее последнего параметра, но можно и просто перечислить через запятую значения элементов этого массива или ничего не указывать в знак того, что он пуст. Детали декларации таких параметров несколько отличаются.

В Java нужно указать тип элемента массива, многоточие и имя параметра.

```
public class A
{
  public int f(int ... a)
  {
    return a.length;
  }

  public static void main(String[] args)
  {
    A a = new A();

    System.out.println
      ( a.f(new int[]{9, 0}) );
    System.out.println
      ( a.f(1, 2, 3, 4) );
  }
}
```

В C# нужно указать модификатор **`params`**, тип самого массива и имя параметра.

```
using System;

public class A
{
  public int f(params int[] a)
  {
    return a.Length;
  }

  public static void Main
  {
    A a = new A();

    Console.WriteLine
      ( a.f(new int[]{9, 0}) );
    Console.WriteLine
      ( a.f(1, 2, 3, 4) );
  }
}
```

В Java требуется указывать некоторые типы исключений, возникновение которых возможно при работе метода, в заголовке метода.

Точнее, все исключения делятся на два вида — *проверяемые (checked)* и *непроверяемые (unchecked)*. Непроверяемыми считаются исключения, возникновение которых может быть непреднамеренно — обращение к методу или полю по ссылке, равной **`null`**, превышение ограничений на размер стека или занятой динамической памяти, и пр. Проверяемые исключения предназначены для передачи сообщений о возникновении специфических ситуаций и всегда явно создаются в таких ситуациях.

Непроверяемое исключение должно иметь класс, наследующий `java.lang.Error`, если оно обозначает серьезную ошибку, которая не может быть обработана в рамках обычного приложения, или `java.lang.RuntimeException`, если оно может возникать при нормальной работе виртуальной машины Java. Если класс исключения не наследует одному из этих классов, оно считается проверяемым.

Все классы проверяемых исключений, возникновение которых возможно при работе метода, должны быть описаны в его заголовке

В C# исключения, возникновение которых возможно при работе метода, никак не описываются.

после ключевого слова **throws**.

Если некоторый метод вызывает другой, способный создать проверяемое исключение типа T, то либо этот вызов должен быть в рамках **try**-блока, для которого имеется обработчик исключений типа T или более общего типа, либо вызывающий метод тоже должен указать тип T или более общий среди типов исключений, возникновение которых возможно при его работе.

```
public void m(int x)
  throws MyException
{
  throw new MyException();
}

public void n(int x)
  throws MyException
{
  m(x);
}

public void k(int x)
{
  try { m(x); }
  catch(MyException e)
  { ... }
}
```

В Java все параметры операций передаются по значению.

Поскольку все типы, кроме примитивных, являются ссылочными, значения таких типов — ссылки на объекты. При выполнении операции можно изменить состояние объекта, переданного ей в качестве параметра, но не ссылку на него.

В C# можно определить параметры операций, передаваемые по ссылке, и выходные параметры операций.

Параметры, передаваемые по ссылке, помечаются модификаторов **ref**. Выходные параметры помечаются модификатором **out**.

При вызове операции значения этих параметров должны быть помечены так же.

```
using System;

public class A
{
  public void f
    (int a, ref int b, out int c)
  {
    c = b - a;
    b += a;
  }

  public static void Main()
  {
    A a = new A();
    int n = 3, m = 0;

    Console.WriteLine
      ("n = " + n + " , m = " + m);
    a.f(1, ref n, out m);
    Console.WriteLine
      ("n = " + n + " , m = " + m);
  }
}
```

Описание метаданных

В обоих языках (в Java — начиная с версии 5) имеются встроенные средства для некоторого их расширения, для описания так называемых метаданных — данных, описывающих элементы кода. Это специальные модификаторы у типов, элементов типов и параметров операций, называемые в Java ***аннотациями (annotations)***, а в C# — ***атрибутами (attributes)***. Один элемент кода может иметь несколько таких модификаторов.

Такие данные служат для указания дополнительных свойств классов, полей, операций и параметров операций. Например, можно пометить специальным образом поля класса, которые должны записываться при преобразовании объекта этого класса в поток байтов для долговременного хранения или передачи по сети. Можно пометить методы, которые должны работать только в рамках транзакций или, наоборот, только вне транзакций.

Метаданные служат встроенным механизмом расширения языка, позволяя описывать простые дополнительные свойства сущностей этого языка в нем самом, не разрабатывая каждый раз специализированные трансляторы. Обработка метаданных должна, конечно, осуществляться дополнительными инструментами, но такие инструменты могут быть достаточно просты — им не нужно реализовывать функции компилятора исходного языка.

В обоих языках аннотации могут иметь структуру — свойства или параметры, которым можно присваивать значения. Эту структуру можно определить в описании специального аннотационного типа.

В приведенных ниже примерах определяются несколько типов аннотаций, которые затем используются для разметки элементов кода. Класс `A` помечен аннотацией, имеющей указанные значения свойств, оба метода помечены простой аннотаций (указывающей, например, что такой метод должен быть обработан особым образом), кроме того, метод `n()` помечен еще одной аннотацией. Параметр метода `n()` также помечен простой аннотацией.

```
@interface SimpleMethodAnnotation {}


@interface SimpleParameterAnnotation{}


@interface ComplexClassAnnotation
{

  int id();
  String author() default
    "Victor Kuliamin";
  String date();
}

@interface AdditionalMethodAnnotation
{

  String value() default "";
}

@ComplexClassAnnotation
(
  id     = 126453,
  date   = "23.09.2005"
)
public class A
{
  @SimpleMethodAnnotation
  public void m() { ... }

  @SimpleMethodAnnotation
```

```
class SimpleMethodAttribute
  : Attribute
{}

class SimpleParameterAttribute
  : Attribute
{}

class ComplexClassAttribute
  : Attribute
{
  public int id;
  public String author =
    "Victor Kuliamin";
  public String date;
}

class AdditionalMethodAttribute
  : Attribute
{
  public String value = "";
}

[ComplexClassAttribute
 (
    id     = 126453,
    date   = "23.09.2005"
 )]
public class A
{
  [SimpleMethodAttribute]
  public void m() { ... }

  [SimpleMethodAttribute,
```

```
  @AdditionalMethodAnnotation
  ( value = "123" )
  public void n
    (@SimpleParameterAnnotation int k)
  { ... }
}
```

В Java аннотации могут помечать также пакеты (т.е. использоваться в директиве декларации пакета, к которому относится данный файл), декларации локальных переменных и константы перечислимых типов.

Аннотационный тип декларируется с модификатором `@interface` и неявно наследует интерфейсу `java.lang.annotation.Annotation`.

Такой тип не может иметь типовых параметров или явным образом наследовать другому типу.

Свойства аннотационного типа описываются как абстрактные методы без параметров, с возможным значением по умолчанию.

При определении значений свойств аннотации через запятую перечисляются пары <имя свойства> = <значение>.

Помимо свойств, в аннотационном типе могут быть описаны константы (`public static final` поля) и вложенные типы, в том числе аннотационные.

Свойство аннотационного типа может иметь примитивный тип, тип `String`, `Class`, экземпляр шаблонного типа `Class`, перечислимый тип, аннотационный тип или быть массивом элементов одного из перечисленных типов.

```
   AdditionalMethodAttribute
   (value = "123")]
  public void n
  ([SimpleParameterAttribute] int k)
  { ... }
}
```

В C# могут быть описаны глобальные атрибуты, помечающие сборку, в рамках кода которой они встречаются. Такие атрибуты помещаются вне описаний пространств имен.

Также атрибутами могут помечаться отдельные методы доступа к свойствам, индексерам и событиям.

Атрибутный тип описывается как класс, наследующий `System.Attribute` или его наследнику.
Такой тип не может иметь типовых параметров.

Свойства атрибутного типа могут быть *именованными параметрами* и *позиционными параметрами*.

Позиционные параметры определяются при помощи конструкторов атрибутного типа.

Именованные параметры определяются как доступные для чтения и записи нестатические поля и свойства атрибутного типа.

При определении значений свойств атрибута сначала через запятую перечисляются значения позиционных параметров, а затем пары <имя именованного параметра> = <значение>.

В атрибутном типе могут быть описаны такие же элементы, как и в обычном классе.

Ниже приведен пример использования позиционного параметра.

```
class ClassAttribute : Attribute
{
  public ClassAttribute(int id)
  {
    this.id = id;
  }

  int id;
  public string value;
}

[ClassAttribute(4627, value = "")]
public class A { ... }
```

Свойство атрибутного типа может иметь один из типов `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`, `string`, `object`, `System.Type`, перечислимый тип или быть массивом элементов одного из таких типов.

У атрибутов может указываться цель, к которой они привязываются. Для атрибутов, помещаемых вне рамок пространств имен, указание такой цели — assembly — обязательно.

```
[assembly :
  MyAttribute
  (
    id = 4627,
    author = "Victor Kuliamin"
  )]
```

В C# принято соглашение, согласно которому окончание `Attribute` в именах атрибутных типов может отбрасываться при использовании атрибутов такого типа.

Поэтому, имея атрибутный тип `ClassAttribute`, можно использовать в атрибутах как полное имя `ClassAttribute`, так и сокращенное `Class`.

В последнем случае компилятор будет пытаться найти атрибутный тип с именем `Class` или с именем `ClassAttribute`. При этом возможна неоднозначность — оба таких типа могут существовать в рамках сборки и доступных библиотек. Для ее разрешения можно использовать точное имя атрибутного типа с префиксом `@`. Увидев атрибут `@Class`, компилятор будет искать атрибутный тип в точности с именем `Class`.

Средства создания многопоточных программ

Как в Java, так и в C# возможно создание многопоточных приложений. Вообще говоря, каждая программа на этих языках представляет собой набор ***потоков (threads)***, выполняющихся параллельно. Каждый поток является исполняемым элементом, имеющим свой собственный поток управления и стек вызовов операций. Все потоки в рамках одного ***процесса*** (одной виртуальной машины Java или одного процесса среды .NET) имеют общий набор ресурсов, общую память, общий набор объектов, с которыми могут работать.

Каждый поток представляется в языке объектом некоторого класса (`java.lang.Thread` в Java и `System.Threading.Thread` в C#). Для запуска некоторого кода в виде отдельного потока необходимо определить особую операцию в таком объекте и выполнить другую его операцию.

В Java это можно сделать двумя способами.

Первый — определить класс-наследник `java.lang.Thread` и перегрузить в этом классе метод **`public void`** `run()`. Этот метод, собственно и будет выполняться в виде отдельного потока.

Другой способ — определить класс, реализующий интерфейс `java.lang.Runnable` и его метод **`void`** `run()`. После чего построить объект класса `Thread` на основе объекта только что определенного класса.

В C# также можно использовать два способа. С помощью первого можно создать обычный поток, с помощью второго — поток, которому при запуске нужно передать какие-то данные.

Для этого нужно определить метод, который будет выполняться в рамках потока. Этот метод должен иметь тип результата **`void`**. Список его параметров в первом случае должен быть пустым, во втором — состоять из одного параметра типа **`object`**.

В первом варианте на основе этого метода

В обоих случаях для запуска выполнения потока нужно вызвать в объекте класса `Thread` (в первом случае — его наследника) метод **`void`** `start()`.

```
class T extends Thread
{
  int id = 0;
  public T(int id)
  { this.id = id; }

  public void run()
  {
    System.out.println
    ("Thread " + id + " is working");
  }
}

public class A
{
  public static void main(String[] args)
  {
    Thread th1 = new T(1),
           th2 = new T(2),
           th3 = new Thread(
      new Runnable() {
        public void run()
        {
          System.out.println
          ("Runnable is working");
        }
      });

    th1.start();
    th2.start();
    th3.start();
  }
}
```

создается делегат типа `System.Threading.ThreadStart`, во втором — типа `System.Threading.ParameterizedThreadStart`.

Этот делегат передается в качестве аргумента конструктору объекта класса `System.Thread`.

Поток запускается выполнением метода `Start()` у объекта класса `Thread` в первом случае, или метода `Start(`**`object`**`)` во втором.

```
using System;
using System.Threading;

class T
{
  int id;
  public T(int id)
  { this.id = id; }

  public void m()
  {
    Console.WriteLine
    ("Thread " + id + " is working");
  }
}

public class A
{
  static void m()
  {
    Console.WriteLine
    ("Nonparameterized thread" +
     " is working");
  }

  static void m(object o)
  {
    Console.WriteLine
    ("Thread with object " + o +
     " is working");
  }

  public static void Main()
  {
    Thread th1 = new Thread(
      new ThreadStart(m)),
           th2 = new Thread(
      new ThreadStart(new T(1).m)),
           th3 = new Thread(
    new ParameterizedThreadStart(m));

    th1.Start();
    th2.Start();
    th3.Start(2);
  }
}
```

При разработке приложений, основанных на параллельном выполнении нескольких потоков, большое значение имеют вопросы синхронизации работы этих потоков. Синхронизация позволяет согласовывать их действия и аккуратно передавать данные, полученные в одном потоке, в другой. И недостаточная синхронизация, и избыточная приводят к серьезным проблемам. При недостаточной синхронизации один поток может начать работать с данными, которые еще

находятся в обработке у другого, что приведет к некорректным итоговым результатам. При избыточной синхронизации как минимум производительность приложения может оказаться слишком низкой, а в большинстве случаев приложение просто не будет работать из-за возникновения *тупиковых ситуаций (deadlocks)*, в которых два или более потоков не могут продолжать работу, поскольку ожидают друг от друга освобождения необходимых им ресурсов.

В обоих языках имеются конструкции, которые реализуют синхронизационный примитив, называемый ***монитором (monitor)***. Монитор представляет собой объект, позволяющий потокам «захватывать» и «отпускать» себя. Только один поток может «держать» монитор в некоторый момент времени — все остальные, попытавшиеся захватить монитор после его захвата этим потоком, будут приостановлены до тех пор, пока этот поток не отпустит монитор.

Для синхронизации используется конструкция, гарантирующая, что некоторый участок кода в каждый момент времени выполняется не более чем одним потоком. В начале этого участка нужно захватить некоторый монитор, в качестве которого может выступать любой объект ссылочного типа, в конце — отпустить его. Такой участок помещается в блок (или представляется в виде одной инструкции), которому предшествует указание объекта-монитора с ключевым словом **`synchronized`** в Java или **`lock`** в C#.

```
public class PingPong extends Thread
{
  boolean odd;
  PingPong (boolean odd)
  { this.odd = odd; }

  static int counter = 1;
  static Object monitor = new Object();

  public void run()
  {
    while(counter < 100)
    {
      synchronized(monitor)
      {
        if(counter%2 == 1 && odd)
        {
          System.out.print("Ping ");
          counter++;
        }
        if(counter%2 == 0 && !odd)
        {
          System.out.print("Pong ");
          counter++;
        }
      }
    }
  }

  public static void main
    (String[] args)
  {
    Thread th1 = new PingPong (false),
           th2 = new PingPong (true);

    th1.start();
    th2.start();
  }
}
```

```
using System;
using System.Threading;

public class PingPong
{
  bool odd;

  PingPong (bool odd) { this.odd = odd; }

  static int counter = 1;
  static object monitor = new object();

  public void Run()
  {
    while(counter < 100)
    {
      lock(monitor)
      {
        if(counter%2 == 1 && odd)
        {
          Console.Write("Ping ");
          counter++;
        }
        if(counter%2 == 0 && !odd)
        {
          Console.Write("Pong ");
          counter++;
        }
      }
    }
  }

  public static void Main()
  {
    Thread th1 = new Thread(
      new ThreadStart(
        new PingPong(false).Run)),
           th2 = new Thread(
      new ThreadStart(
        new PingPong(true).Run));

    th1.Start();
    th2.Start();
  }
}
```

Кроме того, в Java любой метод класса может быть помечен как **`synchronized`**. Это значит, что не более чем один поток может выполнять этот метод в каждый момент времени в рамках объекта, если метод нестатический и в рамках всего класса, если он статический.

Такой модификатор эквивалентен помещению всего тела метода в блок, синхронизированный по объекту **`this`**, если метод нестатический, а если метод статический — по выражению **`this.`**`getClass()`, возвращающему объект, который представляет класс данного объекта.

В Java также имеется стандартный механизм использования любого объекта ссылочного типа в качестве монитора для создания более сложных механизмов синхронизации.

Для этого в классе `java.lang.Object` имеются методы `wait()`, приостанавливающие текущий поток до тех пор, пока другой поток не вызовет метод `notify()` или `notifyAll()` в том же объекте, или пока не пройдет указанное время. Все эти методы должны вызываться в блоке, синхронизированном по данному объекту.

Однако это механизм достаточно сложен в использовании и не очень эффективен. Для реализации более сложной синхронизации лучше пользоваться библиотечными классами из пакетов `java.util.concurrent` и `java.util.concurrent.locks`, появившихся в JDK версии 5 (см. ниже).

Библиотеки

Обе платформы, как Java, так и .NET, в большой степени опираются на библиотеки готовых компонентов. Сообщество разработчиков ведет постоянную работу по выработке стандартных интерфейсов компонентов для решения различных задач в разных предметных областях. По достижении определенной степени зрелости такие интерфейсы включаются в стандартные библиотеки.

Для Java этот процесс начался значительно раньше и захватил гораздо больше участников в силу более раннего появления платформы и ее большей открытости и стандартизованности. Поэтому далеко не все часто используемые библиотеки классов распространяются в составе платформ J2SE и J2EE. Разработка библиотек .NET ведется, в основном, силами программистов, работающих в Microsoft и некоторых ее партнерах.

В данном разделе дается краткий обзор основных библиотек, подробности см. в [10] и [11].

Основные классы языка Java содержатся в пакете `java.lang`.

Базовым классом для всех ссылочных типов Java служит класс `java.lang.Object`.

Этот класс содержит следующие методы.

`boolean` `equals(Object)` — предназначен для сравнения объектов с учетом их внутренних данных, перегружается в наследниках. В классе

Основные классы C# содержатся в пространстве имен `System` в сборке mscorlib.

Базовым типом для всех типов C# служит класс `System.Object`, который также имеет имя **`object`**.

`bool` `Equals(`**`object`**`)` — аналог метода `equals()` в Java.

`static bool` `Equals(`**`object, object`**`)` —

`Object` сравнивает объекты на совпадение.

`int` `hashCode()` — возвращает хэш код данного объекта, используется в хэширующих коллекциях. Должен перегружаться одновременно с методом `equals()`.

`String toString()` — преобразует данный объект в строку, перегружается в наследниках. В классе `Object` выдает строку из имени класса и уникального кода объекта в JVM.

`Class<?` **`extends`** `Object> getClass()` — возвращает объект, представляющий класс данного объекта.

`protected void` `finalize()` — вызывается сборщиком мусора на одном из этапов удаления объекта из памяти. Может быть перегружен.

`protected` `Object clone()` — предназначен для построения копий данного объекта, перегружается в наследниках. В классе `Object` копирует поля данного объекта в новый, если класс данного объекта реализует интерфейс `java.lang.Cloneable`, иначе выбрасывает исключение.

`void` `wait()`, **`void`** `wait(`**`long`**` timeout)`, **`void`** `wait(`**`long`**` timeout, `**`int`**` nanos)` — методы, приостанавливающие выполнение текущего потока до вызова `notify()` или `notifyAll()` другим потоком в данном объекте или до истечения заданного интервала времени.

`void` `notify()`, **`void`** `notifyAll()` — методы, оповещающие потоки, которые ждут оповещения по данному объекту. Первый метод «отпускает» только один из ждущих потоков, второй — все.

Класс `System` предоставляет доступ к элементам среды выполнения программы и ряд полезных утилит. Все его элементы — статические.

Поля `in`, `out` и `err` в этом классе представляют собой ссылки на стандартные потоки ввода, вывода и вывода информации об ошибках. Они могут быть изменены при помощи методов `setIn()`, `setOut()` и `setErr()`.

Методы **`long`** `currentTimeMillis()` и **`long`** `nanoTime()` служат для получения текущего значения времени.

`void` `exit(`**`int`**` status)` — прекращает выполнение Java машины, возвращая указанное число внешней среде в качестве кода выхода.

`void` `gc()` — запускает сборку мусора. Время от времени сборка мусора запускается и

сравнивает два объекта с помощью `Equals()` или на равенство обеих ссылок **`null`**.

`static bool` `ReferenceEquals(`**`object, object`**`)` — сравнивает ссылки на заданные объекты.

`int` `GetHashCode()` — аналог метода `hashCode()` в Java. Должен перегружаться одновременно с методом `Equals()`.

`string` `ToString()` — аналог метода `toString()` в Java. В **`object`** выдает только имя типа данного объекта.

`System.Type GetType()` — возвращает объект, представляющий тип данного объекта.

`protected object` `MemebrwiseClone()` — создает копию данного объекта, имеющую те же значения всех полей.

Данные о среде выполнения можно получить с помощью класса `Environment`.

В нем имеются методы `GetEnvironmentVariables()` и `GetEnvironmentVariable()` для получения значений переменных окружения, методы для получения командной строки, метод `Exit(`**`int`**`)` для прекращения работы текущего процесса, свойства с данными о машине и текущем пользователе, свойство `TickCount`, хранящее количество миллисекунд с момента запуска системы, и пр.

Управлять стандартным вводом-выводом можно с помощью класса `Console`.

Он содержит свойства `In`, `Out`, `Err`, методы для чтения из потока стандартного ввода и для записи в поток стандартного вывода, а также много других свойств консоли.

самостоятельно.

Методы `getenv()`, `getProperties()` и `getProperty()` служат для получения текущих значений переменных окружения и свойств Java машины, задаваемых ей при запуске с опцией –d.

`load()`, `loadLibrary()` — служат для загрузки библиотек, например, реализующих native интерфейсы.

`void` `arraycopy()` — используется для быстрого копирования массивов.

`int` `identityHashCode(Object)` — возвращает уникальный числовой идентификатор данного объекта в Java машине.

Другой класс, содержащий методы работы со средой выполнения, — `Runtime`.

Для работы со строками используются классы `String`, `StringBuffer` и `StringBuilder` (последний появился в Java 5). Все они реализуют интерфейс последовательностей символов `CharSequence`.

Класс `String` представляет неизменяемые строки, два других класса — изменяемые. Отличаются они тем, что все операции `StringBuffer` синхронизованы, а операции `StringBuilder` — нет. Соответственно, первый класс нужно использовать для представления строк, с которыми могут работать несколько потоков, а второй — для повышения производительности в рамках одного потока.

В пакете `java.lang` находятся классы-обертки примитивных типов `Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, `Double`.

Все числовые классы наследуют классу `java.lang.Number`.

Эти классы содержат константы, представляющие крайние значения соответствующих типов, методы для преобразования значений соответствующих типов в строку и для получения этих значений из строк, а также методы для работы с битовым представлением значений числовых типов.

Для форматированного представления чисел используются классы `java.text.NumberFormat` и `java.text.DecimalFormat`.

Набор математических функций и констант реализован в виде элементов класса `java.lang.Math`.

Для генерации псевдослучайных чисел можно использовать как метод `Math.random()`, так и

Для работы со строками используются классы `System.String`, представляющий неизменяемые строки, и `System.Text.StringBuilder`, представляющий изменяемые строки.

Обертками примитивных типов C# служат следующие структурные типы из пространства имен `System`.

`Boolean`, `Byte`, `SByte`, `Int16`, `Int32`, `Int64`, `UInt16`, `UInt32`, `UInt64`, `Single`, `Double`, `Decimal`.

Они также содержат аналогичные константы и методы.

Для работы с форматированным строковым представлением чисел используются методы `Parse()` и `ToString()` тех же классов с дополнительными параметрами, реализующими интерфейс `IFormatProvider`. Чаще всего это объекты класса `System.Globalization.NumberFormatInfo`.

Набор математических функций и констант реализован в виде элементов класса `System.Math`.

Для генерации псевдослучайных чисел используется класс `System.Random`.

обладающий большей функциональностью класс `java.util.Random`.

Для более сложных вычислений можно использовать классы пакета `java.math` — `BigInteger` и `BigDecimal`, представляющие целые числа произвольной величины и десятичные дроби произвольной точности.

`java.lang.Thread` — класс, объекты которого представляют потоки в Java машине.

Он содержит методы, позволяющие прервать ожидание данным потоком синхронизационной операции (`interrupt()`), подождать конца работы данного потока (`join()`), запустить поток (`start()`), приостановить выполнение текущего потока на какое-то время (`sleep()`), получить текущий поток (`currentThread()`), а также получить различные характеристики данного потока (приоритет, имя, и пр.).

Класс `java.lang.ThreadLocal` служит для хранения значений, которые должны быть специфичны для потока. Т.е. значение, получаемое методом `get()` из объекта этого класса, — то самое, которое текущий поток сохранил ранее с помощью метода `set()`.

В пакете `java.lang` находится и набор наиболее важных интерфейсов.

`CharSequence` — интерфейс последовательности символов.

`Cloneable` — маркирующий интерфейс объектов, для которых можно строить копии. Сам по себе он не требует реализации каких-либо методов, хотя для построения копий нужно перегрузить метод `clone()`, а лишь позволяет использовать функциональность этого метода в классе `Object`.

`Iterable<T>` — интерфейс итерируемых коллекций. Объекты, реализующие этот интерфейс, могут, наравне с массивами, использоваться в инструкции цикла по коллекции.

`Comparable<T>` — интерфейс, реализуемый классами, объекты которых линейно упорядочены, т.е. любые два объекта этого класса сравнимы по отношению «больше/меньше». Используется, например, для реализации коллекций с быстрым поиском.

В пакете `java.lang` также находится ряд классов, объекты которых представляют элементы самого языка — `Class<T>`, представляющий классы, `Enum<T>`, представляющий перечислимые типы, и

Классом, представляющим потоки .NET в C#, является `System.Threading.Thread`.

Текущий поток может быть получен с помощью его свойства `CurrentThread`.

Этот класс тоже содержит методы `Interrupt()`, `Join()`, `Start()`, `Sleep()` и др.

Аналогичные интерфейсы в C# следующие.

`System.ICloneable` — требует реализации метода `Clone()` для создания копий.

`System.Collections.IEnumerable`, `System.Collections.Generic.IEnumerable<T>` — интерфейсы итерируемых коллекций.

`System.IComparable`, `System.IComparable<T>` — интерфейсы типов, объекты которых линейно упорядочены.

В C#, в отличии от Java, где шаблонные типы всегда имеют и непривязанный к типам-аргументам интерфейс, шаблоны с типизированным интерфейсом просто добавляются в библиотеки к уже существовавшим там до появления C# 2.0 нетипизированным классам и интерфейсам.

Для C# аналогичную роль в механизме рефлексии играют классы `System.Type`, `System.ValueType`, `System.Enum` и классы, входящие в пространство имен `System.Reflection` — `MemberInfo`, `MethodInfo`,

`Package`, представляющий пакеты.

Все эти классы, а также классы, находящиеся в пакете `java.lang.reflect` (`Field`, `Method`, `Constructor`) используются в рамках механизма ***рефлексии (reflection)*** для доступа к информации об элементах языка во время выполнения программы. На механизме рефлексии построены очень многие среды и технологии, входящие в платформу Java, например JavaBeans.

`FieldInfo`, и т.д.

Для анализа структуры и элементов массивов во время работы программы можно использовать методы класса `java.lang.reflect.Array`, позволяющие определить тип элементов массива, получить их значения, создавать новые массивы и пр.

Для манипуляций с массивами — поиска, сортировки, сравнения и пр. — используются методы класса `java.util.Arrays`.

Аналогом обоих классов Java для работы с массивами служит класс `System.Array`.

Он содержит как методы анализа структуры массива, так и операции для поиска, упорядочения, копирования и сравнения массивов.

В обоих языках имеются механизмы, позволяющие определять ***слабые ссылки***. Объект, хранящийся по слабой ссылке, считается сборщиком мусора недоступным по ней и поэтому может быть уничтожен при очередном запуске процедуры сборки мусора, если обычных ссылок на него не осталось.

Слабые ссылки удобны для организации хранилищ объектов, которые не должны мешать их уничтожению, если эти объекты стали недоступны во всех других местах, т.е. ссылки на них остались только из такого хранилища.

Пакет `java.lang.ref` содержит классы для организации сложной работы со ссылками.

Например, класс `java.lang.ref.WeakReference` представляет слабые ссылки.

Другие классы представляют более хитрые виды ссылок.

В C# слабые ссылки реализуются при помощи класса `System.WeakReference`.

Пакет `java.util` содержит классы и интерфейсы, представляющие разнообразные коллекции объектов.

`Collection<T>` — общий интерфейс коллекций Java.

`Collections` — предоставляет набор общеупотребительных операций над коллекциями: построение коллекций с различными свойствами, поиск, упорядочение и пр.

`Set<T>`, `HashSet<T>`, `TreeSet<T>` — интерфейс множества объектов и различные его реализации. `TreeSet<T>` требует, чтобы объекты типа `T` были линейно упорядоченными, и предоставляет быстро работающие (за логарифмическое время от числа объектов в множестве) функции добавления, удаления и

Набор интерфейсов и классов коллекций C# находится в пространствах имен `System.Collections` и `System.Collections.Generic`. В первом находятся интерфейсы и классы нетипизированных коллекций, во втором — шаблоны. Далее упоминаются только шаблонные типы, если их нетипизированный аналог называется соответствующим образом.

Базовые интерфейсы коллекций — `Generic.IEnumerable<T>`, `Generic.ICollection<T>`.

Интерфейс отображений и классы, реализующие отображения — `Generic.IDictionary<K, V>`, `Generic.Dictionary<K, V>`, `Hastable` (нетипизированный).

поиска.

`Map<K, V>`, `HashMap<K, V>`, `TreeMap<K, V>` — интерфейс ассоциативных массивов или отображений (maps) и его различные реализации. `TreeMap<K, V>` требует, чтобы объекты-ключи были линейно упорядоченными, и предоставляет быстрые операции с отображением.

`List<T>`, `ArrayList<T>`, `LinkedList<T>` — интерфейс расширяемого списка и различные его реализации.

`BitSet` — реализует расширяемый список флагов-битов.

`IdentityHashMap<K, V>` реализует отображение, сравнивающее свои ключи по их совпадению, а не с помощью метода `equals()`, как это делают остальные реализации `Map<K, V>` (и `Set<T>`).

`WeakHashMap<K, V>` хранит ключи с помощью слабых ссылок, что позволяет автоматически уничтожать хранящиеся в таком отображении пары ключ-значение, если на объект-ключ других ссылок не осталось.

Много полезных классов-коллекций можно найти вне стандартных библиотек, например, в библиотеке Jakarta Commons [12], части обширного проекта Apache Jakarta Project [13].

Для определения линейного порядка на объектах типа `T` используется интерфейс `java.util.Comparator<T>`.

Наиболее часто применяется его реализация для строк — `java.text.Collator`, абстрактный класс, позволяющий создавать специфические объекты, сравнивающие строки в различных режимах, включая игнорирование регистра символов, использование национально-специфических символов и пр.

Классы `java.util.Calendar`, `java.util.GregorianCalendar`, `java.util.Date` и `java.text.DateFormat` используются для работы с датами.

Первые два класса используются для представления информации о календаре, объекты класса `Date` представляют даты и моменты времени, а последний класс используется для их конвертации в форматированный текст и обратно.

Класса для представления временных интервалов в стандартной библиотеке нет.

Гораздо более широкий набор средств для работы с датами и временами предоставляет

Интерфейс и классы списков — `Generic.IList<T>`, `Generic.List<T>`, `ArrayList` (нетипизированный).

`BitArray` — реализует расширяемый список битов.

Аналогичный интерфейс в C# — `System.Collections.Generic. IComparer<T>`.

Его реализация для строк — абстрактный класс `System.StringComparer`.

Для представления различных календарей используются подклассы `System.Globalization.Calendar`, тоже находящиеся в пространстве имен `System.Globalization`.

Для представления моментов времени — `System.DateType`.

Для интервалов времени — `System.TimeSpan`.

Для форматированного текстового представления дат и времени — `System.Globalization. DateTimeFormatInfo`.

библиотека Joda [14].

Классы `java.util.Timer` и `java.util.TimerTask` служат выполнения определенных действий в заданное время или через определенное время.	Аналоги в C# — `System.Threading.Timer` и делегатный тип `System.Threading.TimerCallback`. Еще одни аналоги находятся в пространстве имен `System.Timers` сборки System.
Для построения и анализа форматированных строк, содержащих данные различных типов, полезен класс `java.util.Formatter`. С помощью реализации интерфейса `java.util.Formattable` можно определить разные виды форматирования для объектов пользовательских типов.	В C# преобразование в форматированную строку осуществляется методом `ToString()` с параметром типа `System.IFormatProvider`. Такой метод есть в типах, реализующих интерфейс `System.IFormattable`. Обычно в качестве объектов, задающих форматирование, используются объекты классов `System.Globalization. CultureInfo`, `System.Globalization. NumberFormatInfo` и `System.Globalization. DateTimeFormatInfo`.
В пакете `java.util` есть несколько классов, представляющих регионально- или национально-специфическую информацию С помощью объектов класса `Currency` представляют различные валюты. Информация о региональных или национально-специфических настройках и параметрах представляется в виде объектов класса `Locale`. Набор объектов, имеющих локализованные варианты, например, строки сообщений на разных языках, может храниться с помощью подклассов `ResourceBundle` — `ListResiourceBundle`, `PropertyResourceBundle`.	Аналогичную роль в C# играют классы пространства имен `System.Globalization` — `RegionInfo` и `CultureInfo`. Для хранения наборов объектов вместе с их аналогами для нескольких культур используются объекты класса `System.Resources.ResourceSet`.
Работа с различными кодировками текста организуется при помощи классов пакета `java.nio.charset`.	Работа с различными кодировками текста организуется при помощи классов пространства имен `System.Text`.
Интерфейс и классы пакета `java.util` `EventListener`, `EventListenerProxy`, `EventObject` используются для реализации образца «подписчик» в рамках спецификаций JavaBeans (см. предыдущую лекцию).	
Класс `java.util.Scanner` реализует простой лексический анализатор текста. Более гибкую работу с регулярными выражениями можно реализовать с помощью классов пакета `java.util.regex`.	В C# работа с регулярными выражениями может быть организована при помощи классов пространства имен `System.Text.RegularExpressions` в сборке System.
Пакет `java.util.concurrent` и его подпакет `locks` содержат набор классов, реализующих коллекции с эффективной синхронизацией работы нескольких потоков (например, разные потоки могут параллельно изменять значения по разным ключам отображения) и примитивы синхронизации потоков — барьеры, семафоры,	Аналогичные функции выполняют классы пространства имен `System.Threading`, расположенные как в сборке mscorlib, так и в System.

события, затворы (latches), блокировки типа «много читателей-один писатель» и пр.	
Пакет `java.util.concurrent.atomic` содержит классы, реализующие гарантированно атомарные действия с данными различных типов.	
Пакет `java.io` содержит класс `File`, представляющий файлы и операции над ними, а также большое количество подклассов абстрактных классов `Reader` и `InputStream`, предназначенных для потокового ввода данных, и `Writer` и `OutputStream`, предназначенных для потокового вывода данных.	Аналогичные классы содержатся в пространстве имен `System.IO`.
Пакет `java.nio` содержат классы для организации более эффективного асинхронного ввода-вывода.	Аналоги классов из `java.nio` находятся в пространстве имен `System.Runtime.Remoting.Channels` в сборках mscorlib и System.Runtime.Remoting.
Классы и интерфейсы, лежащие в основе компонентной модели JavaBeans, находятся в пакете `java.beans`.	Классы и интерфейсы, лежащие в основе компонентной модели графических элементов управления .NET, находятся в пространстве имен `System.ComponentModel`.
На основе этой модели реализованы библиотеки элементов управления графического пользовательского интерфейса (graphical user interface, GUI) Java.	Эти классы, в основном, расположены в сборке System.
Одна из этих библиотек (самая старая и не очень эффективная) размещается в пакете `java.awt`.	Библиотека элементов GUI находится в пространстве имен `System.Windows.Forms` в рамках сборки `System.Windows.Forms`.
Другая, более новая и демонстрирующая б***о***льшую производительность — в пакете `javax.swing`.	Библиотека классов общего назначения для работы с графикой находится в пространстве `System.Drawing` в сборке `System.Drawing`.
Одной из наиболее эффективных библиотек графических элементов управления на Java на данный момент считается библиотека SWT (Standard Widget Toolkit [15]), на основе которой разрабатывается расширяемая среда разработки приложений Eclipse [16].	
Интерфейсы и классы для разработки сетевого ПО и организации связи между приложениями, работающими на разных машинах, находятся в пакетах `java.net`, `javax.net`, `java.rmi`, `javax.rmi`.	Аналогичные классы и интерфейсы находятся в пространствах имен `System.Net` и `System.Runtime.Remoting` в рамках сборок mscorlib и System.
Пакеты `java.security`, `javax.crypto` и `javax.security` определяют основные интерфейсы и классы для поддержки обеспечения безопасных соединений, шифрования, использования различных протоколов безопасности и различных моделей управления ключами и сертификатами.	Аналогичные классы и интерфейсы находятся в пространстве имен `System.Security` в сборках mscorlib, System и System.Security.
Пакеты `java.sql` и `javax.sql` содержат основные интерфейсы и классы для организации работы с базами данных,	Аналогичные библиотеки в .NET находятся в пространстве имен `System.Data` в сборке System.Data.

образующие так называемый интерфейс связи с базами данных JDBC (Java DataBase Connectivity).	Определенные там интерфейсы являются основой ADO.NET.
Пакет `javax.naming` содержит стандартный интерфейс служб директорий, называемый JNDI (Java Naming and Directory Interface) (см. следующие лекции).	Интерфейс и реализация аналогичной службы директорий ActiveDirectory on Microsoft находятся в пространстве имен `System.DirectoryServices` в сборках System.DirectoryServices и System.DirectoryServices.Protocols.

Литература к Лекции 11

[1] J. Gosling, B. Joy, G. Steele, and G. Bracha. Java Language Specification, 3-rd edition. Addison-Wesley Professional, 2005.
Доступна как http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf.

[2] C# Language Specification. Working Draft 2.7. ECMA, June 2004.
Доступна как http://download.microsoft.com/download/8/1/6/81682478-4018-48fe-9e5e-f87a44af3db9/standard.pdf.

[3] C# Langauge Specification 2.0, March 2005 Draft.
Доступна как http://download.microsoft.com/download/8/1/6/81682478-4018-48fe-9e5e-f87a44af3db9/CSharp%202.0%20Specification.doc.

[4] Б. Лисков, Дж. Гатег. Использование абстракций и спецификаций при разработке программ. М.: Мир, 1989.

[5] Б. Майер. Объектно-ориентированное программирование. Концепции разработки. М.: Русская редакция, 2004.

[6] Документация по JNI http://java.sun.com/j2se/1.5.0/docs/guide/jni/index.html.

[7] S. Liang. Java Native Interface: Programmer's Guide and Specification. Addison-Wesley Professional, 1999.

[8] Страница технологии JavaBeans http://java.sun.com/products/javabeans/index.jsp.

[9] JavaBeans Specification 1.01. Доступна через страницу http://java.sun.com/products/javabeans/docs/spec.html.

[10] Документация по библиотекам J2SE http://java.sun.com/j2se/1.5.0/docs/api/index.html.

[11] Страница разработчиков .NET http://www.msdn.microsoft.com/netframework/.

[12] Страница библиотеки Jakarta Commons http://jakarta.apache.org/commons/index.html.

[13] Страница Apache Jakarta Project http://jakarta.apache.org/.

[14] Страница библиотеки Joda http://www.joda.org/.

[15] Страница библиотеки SWT http://www.eclipse.org/swt/.

[16] Страница проекта Eclipse http://www.eclipse.org/.

Лекция 12. Компонентные технологии и разработка распределенного ПО

Аннотация

Рассматриваются основные понятия компонентных технологий разработки ПО и понятие компонента. Рассказывается об общих принципах разработки распределенного ПО и об организации взаимодействия его компонентов в рамках удаленного вызова процедур и транзакций.

Ключевые слова

Программный компонент, интерфейс, программный контракт, компонентная модель, компонентная среда, базовые службы компонентной среды, распределенное ПО, прозрачность, открытость, масштабируемость, безопасность, синхронное и асинхронное взаимодействие, удаленный вызов процедур, транзакция.

Текст лекции

Основные понятия компонентных технологий

Понятие *программного компонента* (*software component*) является одним из ключевых в современной инженерии ПО. Этим термином обозначают несколько различных вещей, часто не уточняя подразумеваемого в каждом конкретном случае смысла.

- Если речь идет об архитектуре ПО (или ведет ее архитектор ПО), под компонентом имеется в виду то же, что часто называется *программным модулем*. Это достаточно произвольный и абстрактный элемент структуры системы, определенным образом выделенный среди окружения, решающий некоторые подзадачи в рамках общих задач системы и взаимодействующий с окружением через определенный интерфейс. В этом курсе для этого понятия употребляется термин *архитектурный компонент* или *компонент архитектуры*.
- На диаграммах компонентов в языке UML часто изображаются компоненты, являющиеся единицами сборки и конфигурационного управления, — файлы с кодом на каком-то языке, бинарные файлы, какие-либо документы, входящие в состав системы. Иногда там же появляются компоненты, представляющие собой единицы развертывания системы, — это компоненты уже в третьем, следующем смысле.
- Компоненты развертывания являются блоками, из которых строится компонентное программное обеспечение. Эти же компоненты имеются в виду, когда говорят о компонентных технологиях, компонентной или компонентно-ориентированной (component based) разработке ПО, компонентах JavaBeans, EJB, CORBA, ActiveX, VBA, COM, DCOM, .Net, Web-службы (web services), а также о компонентном подходе, упоминаемом в названии данного курса. Согласно [1], такой ***компонент*** представляет собой структурную единицу программной системы, обладающую четко определенным интерфейсом, который полностью описывает ее зависимости от окружения. Такой компонент может быть независимо поставлен или не поставлен, добавлен в состав некоторой системы или удален из нее, в том числе, может включаться в состав систем других поставщиков.

Различие между этими понятиями, хотя и достаточно тонкое, все же может привести к серьезному непониманию текста или собеседника в некоторых ситуациях.

В определении архитектурного компонента или модуля делается акцент на выделение структурных элементов системы в целом и декомпозицию решаемых ею задач на более мелкие подзадачи. Представление такого компонента в виде единиц хранения информации (файлов, баз данных и пр.) не имеет никакого значения. Его интерфейс хотя и определен, может быть уточнен и расширен в процессе разработки.

Понятие компонента сборки и конфигурационного управления связано со структурными элементами системы, с которыми приходится иметь дело инструментам, осуществляющим сборку и конфигурационное управление, а также людям, работающим с этими инструментами. Для этих

видов компонентов интерфейсы взаимодействия с другими такими же элементами системы могут быть не определены.

В данной лекции и в большинстве следующих мы будем иметь дело с компонентами в третьем смысле. Это понятие имеет несколько аспектов.

- Компонент в этом смысле — *выделенная структурная единица с четко определенным интерфейсом.* Он имеет более строгие требования к четкости определения интерфейса, чем архитектурный компонент. Абсолютно все его зависимости от окружения должны быть описаны в рамках этого интерфейса. Один компонент может также иметь несколько интерфейсов, играя несколько разных ролей в системе.
  При описании интерфейса компонента важна не только сигнатура операций, которые можно выполнять с его помощью. Становится важным и то, какие другие компоненты он может задействовать при работе, а также каким ограничениям должны удовлетворять входные данные операций и какие свойства выполняются для результатов их работы.

  Эти ограничения являются так называемым ***интерфейсным контрактом*** или ***программным контрактом*** компонента. Интерфейс компонента включает набор операций, которые можно вызвать у любого компонента, реализующего данный интерфейс, и набор операций, которые этот компонент может вызвать в ответ у других компонентов. Интерфейсный контракт для каждой операции самого компонента (или используемой им) определяет ***предусловие*** и ***постусловие*** ее вызова. Предусловие операции должно быть выполнено при ее вызове, иначе корректность результатов не гарантируется. Если эта операция вызывается у компонента, то обязанность позаботиться о выполнении предусловия лежит на клиенте, вызывающем операцию. Если же эта операция вызывается компонентом у другого компонента, он сам обязуется выполнить это предусловие. С постусловием все наоборот — постусловие вызванной у компонента операции должно быть выполнено после ее вызова, и это — обязанность компонента. Постусловие операции определяет, какие ее результаты считаются корректными. В отношении вызываемых компонентом операций выполнение их постусловий должно гарантироваться теми компонентами, у которых они вызываются, а вызывающий компонент может на них опираться в своей работе.

  Пример. Рассмотрим сайт Интернет-магазина. В рамках этого приложения может работать компонент, в чьи обязанности входит вывод списка товаров заданной категории. Одна из его операций принимает на вход название категории, а выдает HTML-страничку в заданном формате, содержащую список всех имеющихся на складе товаров этой категории. Предусловие может состоять в том, что заданная строка действительно является названием категории. Постусловие требует, чтобы результат операции был правильно построенной HTML-страницей, чтобы ее основное содержимое было таблицей со списком товаров именно указанной категории, название каждого из которых представляло бы собой ссылку, по которой можно попасть на его описание, а в остальном — чтобы эта страница была построена в соответствии с принятым проектом сайта.
  Более аккуратно построенный компонент не требовал бы ничего в качестве предусловия (т.е. оно было бы выполнено при любом значении параметра), а в случае некорректного названия категории в качестве результата выдавал бы HTML-страницу с сообщением о неправильном названии категории товаров.
  При реализации интерфейса предусловия операций могут ослабляться, а постусловия — только усиливаться. Это значит, что, реализуя данную операцию, некоторый компонент может реализовать ее для более широкого множества входных данных, чем это требуется предусловием, а также может выполнить в результате более строгие ограничения, чем это требуется постусловием. Однако внешним компонентам нельзя опираться на это, пока они работают с исходным интерфейсом, — реализация может поменяться. Точно так же, если интерфейс компонента требует наличия в системе других компонентов с определенным набором операций, это не означает, что данная реализация этого интерфейса действительно вызывает эти операции.

- Компонент в этом смысле — *единица развертывания.* Он может быть присоединен к остальной системе, когда она уже некоторое время работает, и должен после этого выполнять все свои функции, если в исходной системе уже были все компоненты, от которых он зависит. Он может быть удален из нее. Естественно, после этого могут перестать работать те компоненты, которые зависят от него. Он может быть встроен в программные продукты третьих партий и распространяться вместе с ними. В то же время никакая его часть не обладает этими свойствами.
  В идеале такой компонент может быть добавлен или полностью замещен другой реализацией тех же интерфейсов прямо в ходе работы системы, без ее остановки. Хотя и не все разновидности компонентов обладают этим свойством, его наличие признается крайне желательным.
  Все это означает, что такой компонент должен быть работоспособен в любой среде, где имеются необходимые для его работы другие компоненты. Это требует наличия определенной инфраструктуры, которая позволяет компонентам находить друг друга и взаимодействовать по определенным правилам.
- Набор правил определения интерфейсов компонентов и их реализаций, а также правил, по которым компоненты работают в системе и взаимодействуют друг с другом, принято объединять под именем ***компонентной модели (component model)*** [2]. В компонентную модель входят правила, регламентирующие ***жизненный цикл компонента***, т.е. то, через какие состояния он проходит при своем существовании в рамках некоторой системы (незагружен, загружен и пассивен, активен, находится в кэше и пр.) и как выполняются переходы между этими состояниями.
  Существуют несколько компонентных моделей. Правильно взаимодействовать друг с другом могут только компоненты, построенные в рамках одной модели, поскольку компонентная модель определяет «язык», на котором компоненты могут общаться друг с другом.

  Помимо компонентной модели, для работы компонентов необходим некоторый набор *базовых служб (basic services)*. Например, компоненты должны уметь находить друг друга в среде, которая, возможно, распределена на несколько машин. Компоненты должны уметь передавать друг другу данные, опять же, может быть, при помощи сетевых взаимодействий, но реализации отдельных компонентов сами по себе не должны зависеть от вида используемой связи и от расположения их партнеров по взаимодействию. Набор таких базовых, необходимых для функционирования большинства компонентов служб, вместе с поддерживаемой с их помощью компонентной моделью называется ***компонентной средой*** (или ***компонентным каркасом***, ***component framework***). Примеры известных компонентных сред — различные реализации J2EE, .NET, CORBA. Сами по себе J2EE, .NET и CORBA являются спецификациями компонентных моделей и набора базовых служб, которые должны поддерживаться их реализациями.
  Компоненты, которые работают в компонентных средах, по-разному реализующих одну и ту же компонентную модель и одни и те же спецификации базовых служб, должны быть в состоянии свободно взаимодействовать. На практике этого, к сожалению, не всегда удается достичь, но любое препятствие к такому взаимодействию рассматривается как серьезная, подлежащая скорейшему разрешению проблема.
  Соотношение между компонентами, их интерфейсами, компонентной моделью и компонентной средой можно изобразить так, как это сделано на Рис. 65.
- Компоненты отличаются от классов объектно-ориентированных языков.
  - Класс определяет не только набор реализуемых интерфейсов, но и саму их реализацию, поскольку он содержит код определяемых операций. Контракт компонента, чаще всего, не фиксирует реализацию его интерфейсов.
  - Класс написан на определенном языке программирования. Компонент же не привязан к определенному языку, если, конечно, его компонентная модель этого не требует, — компонентная модель является для компонентов тем же, чем для классов является язык программирования.

- Обычно компонент является более крупной структурной единицей, чем класс. Реализация компонента часто состоит из нескольких тесно связанных друг с другом классов.

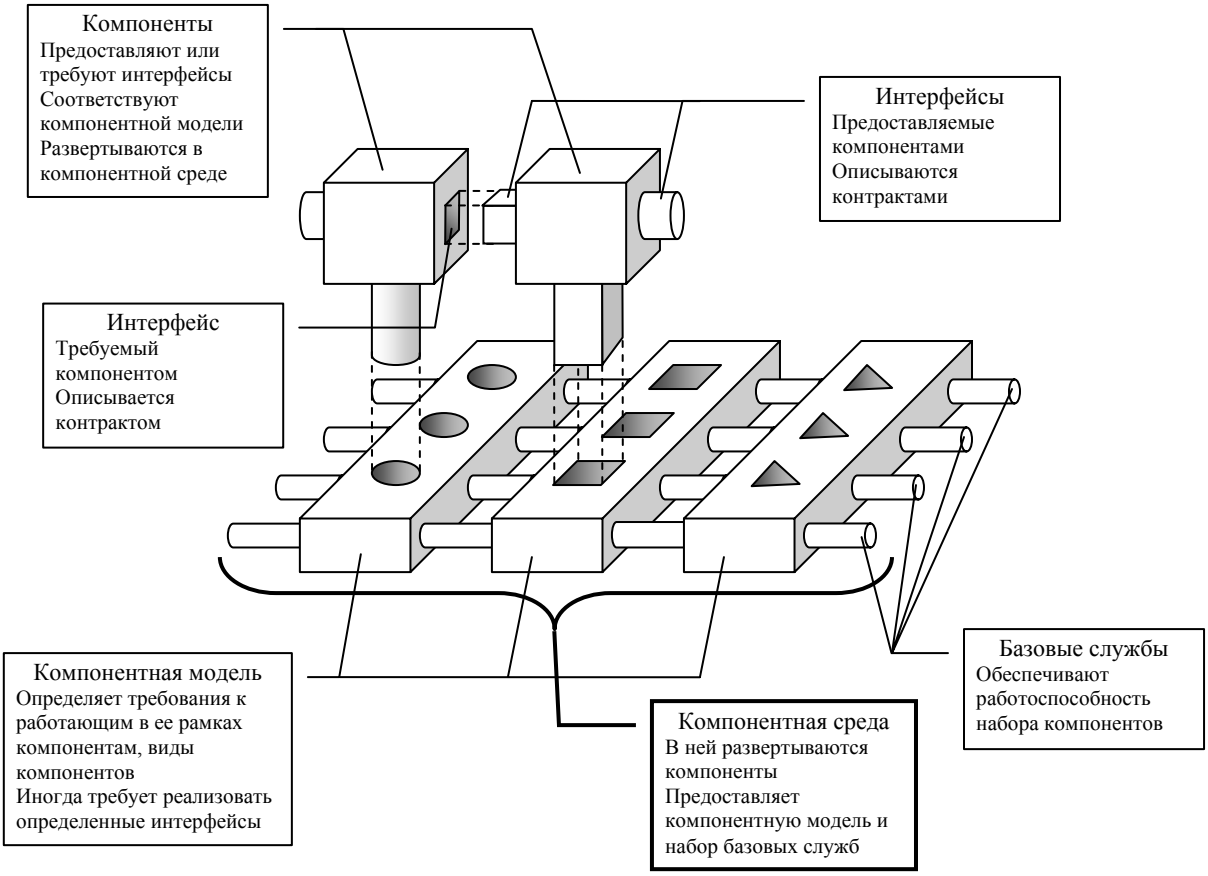


Рисунок 65. Основные элементы компонентного программного обеспечения.

- Понятие компонента является более узким, чем понятие *программного модуля*. Основное содержание понятия модуля — наличие четко описанного интерфейса между ним и окружением. Понятие компонента добавляет атомарность развертывания, а также возможность поставки или удаления компонента отдельно от всей остальной системы.
- Возможность включения и исключения компонента из системы делает его отдельным элементом продаваемого ПО. Компоненты, хотя и не являются законченными продуктами, могут разрабатываться и продаваться отдельно, если они следуют правилам определенной компонентной модели и реализуют достаточно важные для покупателей функции, которые те хотели бы иметь в рамках своей программной системы.
  Надо отметить, что, хотя рынок ПО существует достаточно давно, а компонентные технологии разрабатываются уже около 20 лет, рынок компонентов развивается довольно медленно. Поставщиками компонентов становятся лишь отдельные компании, тесно связанные с разработчиками конкретных компонентных сред, а не широкое сообщество индивидуальных и корпоративных разработчиков, как это представляли себе создатели компонентных технологий при их зарождении.
  По-видимому, один из основных факторов, мешающих развитию этого рынка, — это «гонка технологий» между поставщиками основных компонентных сред. Ее следствием является отсутствие стабильности в их развитии. Новые версии появляются слишком часто, и достаточно часто при выходе новых версий изменяются элементы компонентной модели. Так что при разработке компонентов для следующей версии приходится следовать уже несколько другим правилам, а старые компоненты с трудом могут быть переиспользованы.

Общие принципы построения распределенных систем

В дальнейшем мы будем рассматривать компонентные технологии в связи с разработкой распределенных программных систем. Прежде чем углубляться в их изучение, полезно разобраться в общих принципах построения таких систем, без привязки к компонентному подходу. Тогда многие из решений, применяемых в рамках таких технологий, становятся гораздо более понятными.

Построение распределенных систем высокого качества является одной из наиболее сложных задач разработки ПО. Технологии типа J2EE и .NET создаются как раз для того, чтобы сделать разработку широко встречающихся видов распределенных систем — так называемых бизнес-приложений, поддерживающих решение бизнес-задач некоторой организации, — достаточно простой и доступной практически любому программисту. Основная задача, которую пытаются решить с помощью распределенных систем — обеспечение максимально простого доступа к возможно большему количеству ресурсов как можно большему числу пользователей. Наиболее важными свойствами такой системы являются *прозрачность, открытость, масштабируемость* и *безопасность*.

- ***Прозрачность (transparency)***.
  Прозрачностью называется способность системы скрыть от пользователя физическое распределение ресурсов, а также аспекты их перераспределения и перемещения между различными машинами в ходе работы, репликацию (т.е. дублирование) ресурсов, трудности, возникающие при одновременной работе нескольких пользователей с одним ресурсом, ошибки при доступе к ресурсам и в работе самих ресурсов.
  Технология разработки распределенного ПО тоже может обладать прозрачностью настолько, насколько она позволяет разработчику забыть о том, что создаваемая система распределена, и насколько легко в ходе разработки можно отделить аспекты построения системы, связанные с ее распределенностью, от решения задач предметной области или бизнеса, в рамках которых системе предстоит работать.
  Степень прозрачности может быть различной, поскольку скрывать все эффекты, возникающие при работе распределенной системы, неразумно. Кроме того, прозрачность системы и ее производительность обычно находятся в обратной зависимости — например, при попытке преодолеть отказы в соединении с сервером большинство Web-браузеров пытается установить это соединение несколько раз, а для пользователя это выглядит как сильно замедленная реакция системы на его действия.
- ***Открытость (openness)***.
  Открытость системы определяется как полнота и ясность описания интерфейсов работы с ней и служб, которые она предоставляет через эти интерфейсы. Такое описание должно включать в себя все, что необходимо знать для того, чтобы пользоваться этими службами, независимо от реализации данной системы и платформы, на которой она развернута. Один из основных элементов описания службы — ее контракт.
  Открытость системы важна как для обеспечения ее переносимости, так и для облегчения использования системы и возможности построения других систем на ее основе.
  Распределенные системы обычно строятся с использованием служб, предоставляемых другими системами, и в то же время сами часто являются составными элементами или поставщиками служб для других систем.
  Именно поэтому использование компонентных технологий при разработке практически полезного распределенного ПО неизбежно.
- ***Масштабируемость (scalability)***.
  Масштабируемость системы — это зависимость изменения ее характеристик от количества ее пользователей и подключенных ресурсов, а также от степени географической распределенности системы. В число значимых характеристик при этом попадают функциональность, производительность, стоимость, трудозатраты на разработку, на внесение изменений, на сопровождение, на администрирование, удобство работы с системой. Для некоторых из них наилучшая возможная масштабируемость обеспечивается линейной зависимостью, для других хорошая масштабируемость означает, что показатель

не меняется вообще при изменении масштабов системы или изменяется незначительно. Система хорошо масштабируема по производительности, если параметры задач, решаемых ею за одно и то же время, можно увеличивать достаточно быстро (лучше — линейно или еще быстрее, но это возможно не для всех задач) при возрастании количества имеющихся ресурсов, в частности, отдельных машин. Однако, очень плохо, если внесение изменений в систему становится все более трудоемким при ее росте, даже если этот рост линейный, — желательно, чтобы трудоемкость внесения одного изменения почти не возрастала. Для функциональности же, опять, чем быстрее растет число доступных функций при росте числа вовлеченных в систему элементов, тем лучше.
Большую роль играет *административная масштабируемость* системы — зависимость удобства работы с ней от числа административно независимых организаций, вовлеченных в ее обслуживание.
При реализации очень больших систем (поддерживающих работу тысяч и более пользователей, включающих сотни и более машин) хорошая масштабируемость может быть достигнута только с помощью децентрализации основных служб системы и управляющих ею алгоритмов. Вариантами такого подхода являются следующие.

- Децентрализация обработки запросов за счет использования для этого нескольких машин.
- Децентрализация данных за счет использования нескольких хранилищ данных или нескольких копий одного хранилища.
- Децентрализация алгоритмов работы за счет использования для алгоритмов:
  - не требующих полной информации о состоянии системы;
  - способных продолжать работу при сбое одного или нескольких ресурсов системы;
  - не предполагающих единого хода времени на всех машинах, входящих в систему.
- Использование, где это возможно, *асинхронной связи* — передачи сообщений без приостановки работы до прихода ответа.
- Использование комбинированных систем организации взаимодействия, основанных на следующих схемах.
  - Иерархическая организация систем, хорошо масштабирующая задачи поиска информации и ресурсов.
  - *Репликация* — построение копий данных и их распределение по системе для балансировки нагрузки на разные ее элементы. Частным случаем репликации является *кэширование*, при котором результаты наиболее часто используемых запросов запоминаются и хранятся как можно ближе к клиенту, чтобы переиспользовать их при повторении запросов.
  - Взаимодействие точка-точка (peer-to-peer, P2P) обеспечивает независимость взаимодействующих машин от других машин в системе.

- ***Безопасность (safety)***.
Так как распределенные системы вовлекают в свою работу множество пользователей, машин и географически разделенных элементов, вопросы их безопасности получают гораздо большее значение, чем при работе обычных приложений, сосредоточенных на одной физической машине. Это связано как с невозможностью надежно контролировать доступ к различным элементам такой системы, так и с ее доступностью для гораздо более широкого и разнообразного по своему поведению сообщества пользователей.
Понятие безопасности включает следующие характеристики.
  - *Сохранность и целостность данных.*
  При обеспечении групповой работы многих пользователей с одними и теми же данными нужно обеспечивать их сохранность (т.е. предотвращать исчезновение данных, введенных одним из пользователей) и в тоже время целостность, т.е. непротиворечивость, выполнение всех присущих данным ограничений.
  Это непростая задача, которая не имеет решения, удовлетворяющего все стороны во

всех ситуациях, — при одновременном изменении одного и того же элемента данных разными пользователями итоговый результат должен быть непротиворечив и поэтому часто может совпадать только с вводом одного из них. Как будет обработана такая ситуация и возможно ли ее возникновение вообще, зависит от дополнительных требований к системе, от принятых протоколов работы, от того, какие риски — потерять данные одного из пользователей или значительно усложнить работу пользователей с системой — будут сочтены более важными.

- *Защищенность данных и коммуникаций.*
  При работе с коммерческими системами, содержащими большие объемы персональной и бизнес-информации, а также с системами обслуживания пользователей государственных ведомств очень важна защищенность как информации, постоянно хранящейся в системе, так и информации одного сеанса работы. Для распределенных систем обеспечить защищенность гораздо сложнее, поскольку нельзя физически изолировать все элементы системы и разрешить доступ к ней только проверенным и обладающим необходимыми знаниями и умениями людям.
- *Отказоустойчивость и способность к восстановлению после ошибок.*
  Одним из достоинств распределенных систем является возможность построения более надежно работающей системы из не вполне надежных компонентов. Однако для того, чтобы это достоинство стало реальным, необходимо тщательное проектирование систем с тем, чтобы избежать зависимости работоспособности системы в целом от ее отдельных элементов. Иначе достоинство превращается в недостаток, поскольку в распределенной системе элементов больше и выше вероятность того, что хотя бы один элемент выйдет из строя и хотя бы один ресурс окажется недоступным.
  Еще важнее для распределенных систем уметь восстанавливаться после сбоев. Уровни этого восстановления могут быть различными. Обычно данные одного короткого сеанса работы считается возможным не восстанавливать, поскольку такие данные часто малозначимы или легко восстанавливаются (иначе стоит серьезно рассмотреть необходимость восстановления сеансов). Но так называемые постоянно хранимые (persistent) данные чаще всего требуется восстанавливать в их последнем непротиворечивом состоянии.

Перед разработчиками систем, удовлетворяющих перечисленным свойствам, встает огромное количество проблем. Решать их все сразу просто невозможно в силу ограниченности человеческих способностей. Чтобы хоть как-то структурировать эти проблемы, их разделяют по следующим аспектам [3].

- **Связь.**
  Организация связи и передачи данных между элементами системы.
  В связи с этим аспектом возникают следующие задачи.
  - Какие протоколы использовать для передачи данных.
  - Как реализовать обращения к процедурам и методам объектов одних процессов из других.
  - Какой способ передачи данных выбрать — ***синхронный*** или ***асинхронный***. В первом случае сторона, инициировавшая передачу, приостанавливает свою работу до прихода ответа другой стороны на переданное сообщение. Во втором случае первая сторона имеет возможность продолжить работу, пока данные передаются и обрабатываются другой стороной.
  - Нужно ли, и если нужно, то как, организовать хранение (асинхронных) сообщений в то время, когда и отправитель и получатель сообщения могут быть неактивны.
  - Как организовать передачу непрерывных потоков данных, представляющих собой аудио-, видеоданные или смешанные потоки данных. Этот вопрос имеет большое значение, поскольку заметные человеку прерывания в передаче таких данных приводят к значительному падению качества предоставляемых услуг.

- **Именование.**
  Поддержка идентификации и поиска отдельных ресурсов внутри системы.
  - По каким правилам присваивать имена и идентификаторы различным ресурсам.
  - Как организовать поиск ресурсов в системе по идентификаторам и атрибутам, описывающим какие-нибудь свойства ресурсов.
  - Как размещать и находить мобильные ресурсы, изменяющие свое физическое положение в ходе работы.
  - Как организовывать и поддерживать в рабочем состоянии сложные ссылочные структуры, необходимые для описания имеющихся в распределенной системе ресурсов. Как, например, находить и удалять ресурсы, ставшие никому не доступными.
- **Процессы.**
  Организация работ в рамках процессов и потоков.
  - Как разделить работы в системе по отдельным процессам и машинам.
  - Нужно ли определять различные ролей процессов в системе, например, клиентские и серверные, и как организовывать их работу.
  - Как организовать работу исполняемых ***агентов*** — процессов, способных перемещаться между машинами и выполнять свои задачи в любой подходящей среде.
- **Синхронизация.**
  Синхронизация параллельно выполняемых потоков работ.
  - Как синхронизовать действия отдельных процессов и потоков, работающих в системе, для получения нужных результатов.
  - Как организовать работу многих процессов на разных машинах в том случае, если в системе нельзя непротиворечиво определить глобальное время.
  - Как организовать выполнение транзакций — таких наборов действий, которые надо либо все выполнить, либо не выполнить ни одного из них.
- **Целостность.**
  Поддержка целостности данных и непротиворечивости вносимых изменений.
  - Каким образом можно обеспечивать целостность данных.
  - Какие *модели непротиворечивости* нужно поддерживать. Модель непротиворечивости определяет, на основе каких требований формируются результаты выполняемых одновременно изменений и что доступно клиентам, выполнявшим эти изменения.
  - Какие протоколы обеспечения непротиворечивости, создания и записи транзакций, создания и согласования реплик и кэшей использовать для выполнения требований этих моделей.
- **Отказоустойчивость.**
  Организация отказоустойчивой работы.
  - Как организовать отказоустойчивую работу одного процесса.
  - Как обеспечить надежную связь между элементами системы.
  - Какие протоколы использовать для реализации надежной двусторонней связи или надежных групповых рассылок.
  - Какие протоколы использовать для записи промежуточных состояний и восстановления данных и работы системы после сбоев.
- **Защита.**
  Организация защищенности данных и коммуникаций.
  - Как организовать защиту системы в целом.
    При этом большее значение, чем технические аспекты, имеют организационные и психологические факторы — проблемы определения процедур проведения работ,

обеспечивающих нужный уровень защищенности, и проблемы соблюдения людьми этих процедур.

- Как организовать защиту данных от несанкционированного доступа.
- Как обеспечить защиту каналов связи от двух видов атак — воспрепятствовать несанкционированному доступу к передаваемой информации и не дать подменить информацию в канале.
- Какие протоколы аутентификации пользователей, подтверждения идентичности и авторства использовать.

Из перечисленных тем отдельного рассмотрения заслуживают вопросы организации передачи сообщений и транзакций, тем более что все рассматриваемые далее технологии используют эти механизмы. Более того, практически любая распределенная система сейчас строится на основе ***программного обеспечения промежуточного уровня*** (***middleware***, это программное обеспечение, которое предназначено для облегчения интеграции ПО, размещенного на нескольких машинах, в единую распределенную систему и поддержки работы такой системы), содержащего ту или иную их реализацию.

Синхронное и асинхронное взаимодействие

При описании взаимодействия между элементами программных систем инициатор взаимодействия, т.е. компонент, посылающий запрос на обработку, обычно называется ***клиентом***, а отвечающий компонент, тот, что обрабатывает запрос — ***сервером***. «Клиент» и «сервер» в этом контексте обозначают роли в рамках данного взаимодействия. В большинстве случаев один и тот же компонент может выступать в разных ролях — то клиента, то сервера — в различных взаимодействиях. Лишь в небольшом классе систем роли клиента и сервера закрепляются за компонентами на все время их существования.

Синхронным (synchronous) называется такое взаимодействие между компонентами, при котором клиент, отослав запрос, блокируется и может продолжать работу только после получения ответа от сервера. По этой причине такой вид взаимодействия называют иногда ***блокирующим (blocking)***.

Обычное обращение к функции или методу объекта с помощью передачи управления по стеку вызовов является примером синхронного взаимодействия.

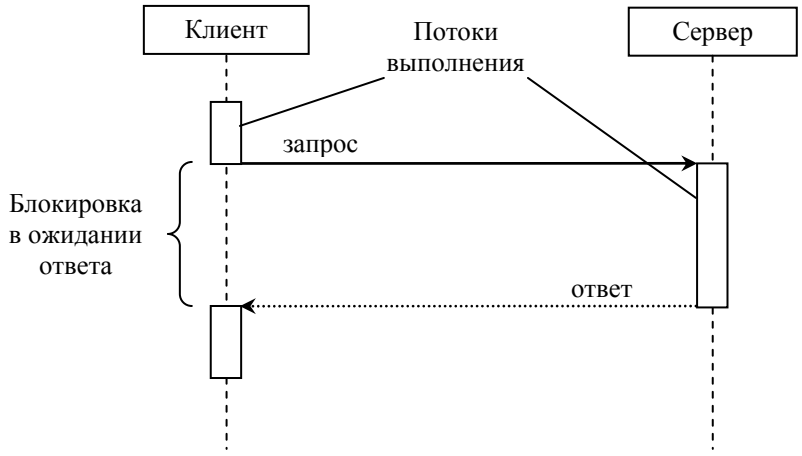


Рисунок 66. Синхронное взаимодействие.

Синхронное взаимодействие достаточно просто организовать, и оно гораздо проще для понимания. Человеческое сознание обладает единственным «потоком управления», представленным в виде фокуса внимания, и поэтому человеку проще понимать процессы, которые разворачиваются последовательно, поскольку не нужно постоянно переключать внимание на происходящие одновременно различные события. Код программы клиентского компонента, описывающей синхронное взаимодействие, устроен проще — его часть, отвечающая за обработку ответа сервера, находится непосредственно после части, в которой формируется запрос. В силу

своей простоты синхронные взаимодействия в большинстве систем используются гораздо чаще асинхронных.

Вместе с тем синхронное взаимодействие ведет к значительным затратам времени на ожидание ответа. Это время часто можно использовать более полезным образом: ожидая ответа на один запрос, клиент мог бы заняться другой работой, выполнить другие запросы, которые не зависят от еще не пришедшего результата. Поскольку все распределенные системы состоят из достаточно большого числа уровней, через которые проходят практически все взаимодействия, суммарное падение производительности, связанное с синхронностью взаимодействий, оказывается очень большим.

Наиболее распространенным и исторически первым достаточно универсальным способом реализации синхронного взаимодействия в распределенных системах является ***удаленный вызов процедур*** (***Remote Procedure Call, RPC***, вообще-то, по смыслу правильнее было бы сказать «дистанционный вызов процедур», но по историческим причинам закрепилась имеющаяся терминология). Его модификация для объектно-ориентированной среды называется ***удаленным вызовом методов (Remote Method Invocation, RMI)***. Удаленный вызов процедур определяет как способ организации взаимодействия между компонентами, так и методику разработки этих компонентов.

На первом шаге разработки определяется интерфейс процедур, которые будут использоваться для удаленного вызова. Это делается при помощи *языка определения интерфейсов (Interface Definition Language, IDL)*, в качестве которого может выступать специализированный язык или обычный язык программирования, с ограничениями, определяющимися возможностью передачи вызовов на удаленную машину.

Определение процедуры для удаленных вызовов компилируется компилятором IDL в описание этой процедуры на языках программирования, на которых будут разрабатываться клиент и сервер (например, заголовочные файлы на C/C++), и два дополнительных компонента — ***клиентскую*** и ***серверную заглушки (client stub*** и ***server stub)***.

Клиентская заглушка представляет собой компонент, размещаемый на той же машине, где находится компонент-клиент. Удаленный вызов процедуры клиентом реализуется как обычный, локальный вызов определенной функции в клиентской заглушке. При обработке этого вызова клиентская заглушка выполняет следующие действия.

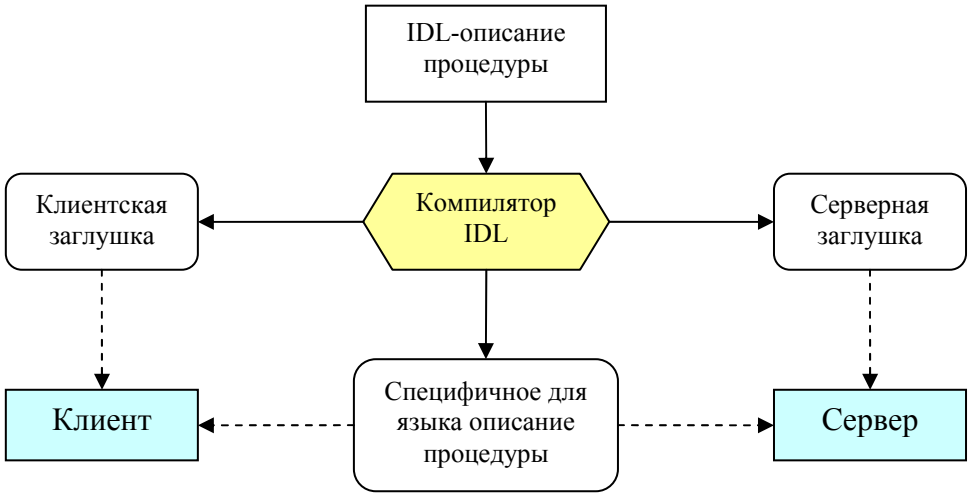


Рисунок 67. Схема разработки компонентов, взаимодействующих с помощью RPC.

1. Определяется физическое местонахождение в системе сервера, для которого предназначен данный вызов. Это шаг называется ***привязкой (binding)*** к серверу. Его результатом является адрес машины, на которую нужно передать вызов.
2. Вызов процедуры и ее аргументы упаковываются в сообщение в некотором формате, понятном серверной заглушке (см. далее). Этот шаг называется ***маршалингом (marshaling)***.

3. Полученное сообщение преобразуется в поток байтов (это *сериализация, serialization*) и отсылается с помощью какого-либо протокола, транспортного или более высокого уровня, на машину, на которой помещен серверный компонент.
4. После получения от сервера ответа, он распаковывается из сетевого сообщения и возвращается клиенту в качестве результата работы процедуры.

В результате для клиента удаленный вызов процедуры выглядит как обращение к обычной функции.

Серверная заглушка располагается на той же машине, где находится компонент-сервер. Она выполняет операции, обратные к действиям клиентской заглушки — принимает сообщение, содержащее аргументы вызова, распаковывает эти аргументы при помощи *десериализации (deserialization)* и *демаршалинга (unmarshaling)*, вызывает локально соответствующую функцию серверного компонента, получает ее результат, упаковывает его и посылает по сети на клиентскую машину.

Таким образом обеспечивается отсутствие видимых серверу различий между удаленным вызовом некоторой его функции и ее же локальным вызовом.

Определив интерфейс процедур, вызываемых удаленно, мы можем перейти к разработке сервера, реализующего эти процедуры, и клиента, использующего их для решения своих задач.

При удаленном вызове процедуры клиентом его обращение оформляется так же, как вызов локальной функции и обрабатывается клиентской заглушкой. Клиентская заглушка определяет адрес машины, на которой находится сервер, упаковывает данные вызова в сообщение и отправляет его на серверную машину. На серверной машине серверная заглушка, получив сообщение, распаковывает его, извлекает аргументы вызова, обращается к серверу с таким вызовом локально, дожидается от него результата, упаковывает результат в сообщение и отправляет его обратно на клиентскую машину. Получив ответное сообщение, клиентская заглушка распаковывает его и передает полученный ответ клиенту.

Эта же техника может быть использована и для реализации взаимодействия компонентов, работающих в рамках различных процессов на одной машине.

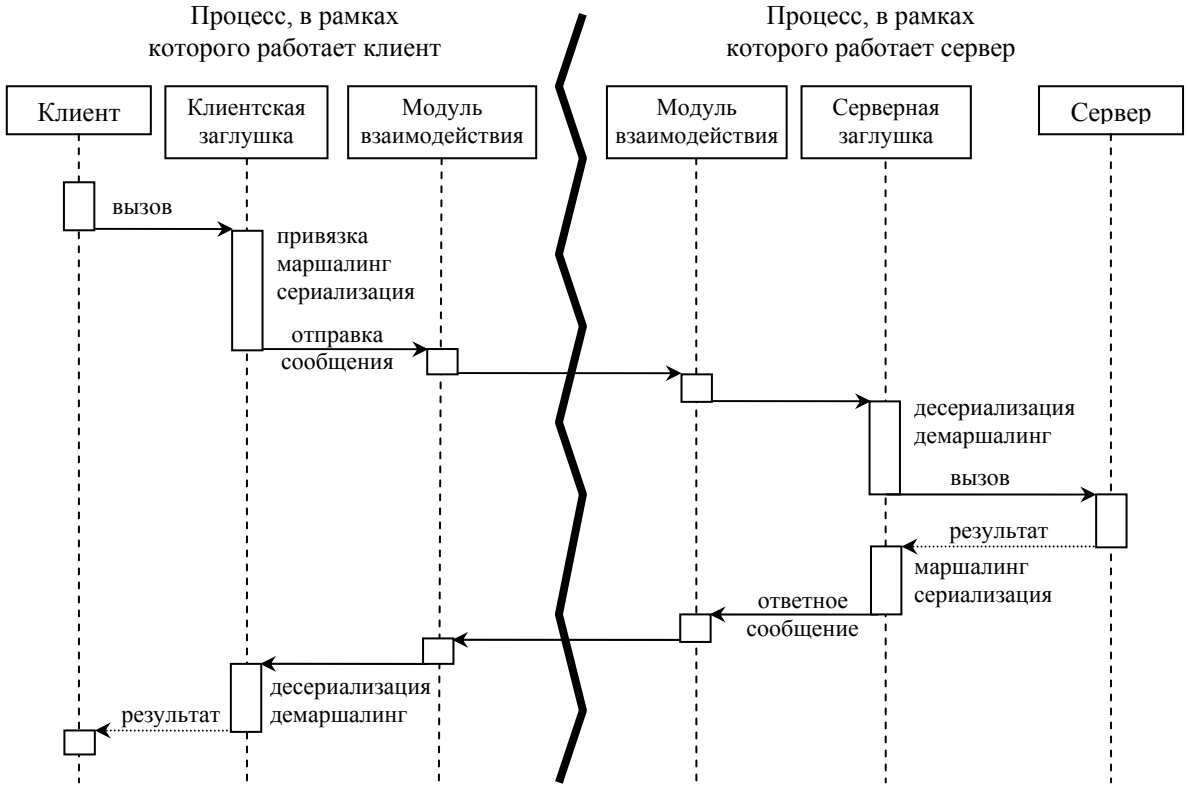


Рисунок 68. Схема реализации удаленного вызова процедуры.

При организации ***удаленного вызова методов*** в объектно-ориентированной среде применяются такие же механизмы. Отличия в его реализации связаны со следующими аспектами.

- Один объект-сервер может предоставлять несколько методов для удаленного обращения к ним.
  Для такого объекта генерируются клиентские заглушки, имеющие в своем интерфейсе все эти методы. Кроме того, информация о том, какой именно метод вызывается, должна упаковываться вместе с аргументами вызова и использоваться серверной заглушкой для обращения именно к этому методу.
  Серверная заглушка в контексте RMI иногда называется ***скелетоном (skeleton)*** или ***каркасом***.
- В качестве аргументов удаленного вызова могут выступать объекты.
  Заметим, что передача указателей в аргументах удаленного вызова процедур практически всегда запрещена — указатели привязаны к памяти данного процесса и не могут быть переданы в другой процесс.
  При передаче объектов возможны два подхода, и оба они используются на практике.
  - Идентичность передаваемого объекта может не иметь значения, например, если сервер использует его только как хранилище данных и получит тот же результат при работе с правильно построенной его копией. В этом случае определяются методы для сериализации и десериализации данных объекта, которые позволяют сохранить их в виде потока байтов и восстановить объект с теми же данными на другой стороне.
  - Идентичность передаваемого объекта может быть важна, например, если сервер вызывает в нем методы, работа которых зависит от того, что это за объект. При этом используется особого рода ссылка на этот объект, позволяющая обратиться к нему из другого процесса или с другой машины, т.е. тоже с помощью удаленного вызова методов.

В рамках ***асинхронного (asynchronous)*** или ***неблокирующего (non blocking)*** взаимодействия клиент после отправки запроса серверу может продолжать работу, даже если ответ на запрос еще не пришел.

Примером асинхронного взаимодействия является электронная почта. Другой пример — распространение сообщений о новостях различных видов в соответствии с имеющимся на текущий момент реестром подписчиков, где каждый подписчик определяет темы, которые его интересуют.

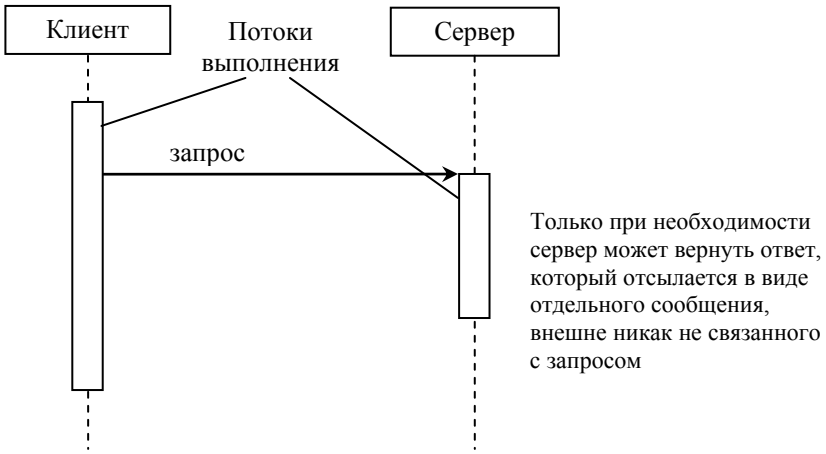


Рисунок 69. Асинхронное взаимодействие.

Асинхронное взаимодействие позволяет получить более высокую производительность системы за счет использования времени между отправкой запроса и получением ответа на него для выполнения других задач. Другое важное преимущество асинхронного взаимодействия — меньшая зависимость клиента от сервера, возможность продолжать работу, даже если машина, на которой находится сервер, стала недоступной. Это свойство используется для организации

надежной связи между компонентами, работающей, даже если и клиент, и сервер не все время находятся в рабочем состоянии.

В то же время асинхронные взаимодействия более сложно использовать. Поскольку при таком взаимодействии нужно писать специфический код для получения и обработки результатов запросов, системы, основанные на асинхронных взаимодействиях между своими компонентами, значительно труднее разрабатывать и сопровождать.

Чаще всего асинхронное взаимодействие реализуется при помощи очередей сообщений. При отправке сообщения клиент помещает его во входную очередь сервера, а сам продолжает работу. После того, как сервер обработает все предшествующие сообщения в очереди, он выбирает это сообщение для обработки, удаляя его из очереди. После обработки, если необходим ответ, сервер создает сообщение, содержащее результаты обработки, и кладет его во входную очередь клиента или в свою выходную.

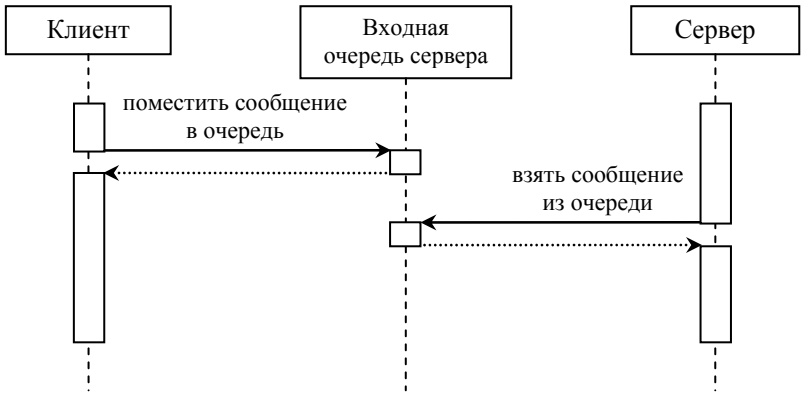


Рисунок 70. Реализация асинхронного взаимодействия при помощи очередей сообщений.

Очереди сообщений могут быть сконфигурированы самыми разными способами. У компонента может иметься одна входная очередь, а может — и несколько, для сообщений от разных источников или имеющих разный смысл. Кроме того, компонент может иметь выходную очередь, или несколько, вместо того, чтобы класть сообщения во входные очереди других компонентов. Очереди сообщений могут храниться независимо как от компонентов, которые кладут туда сообщения, так и от тех, которые забирают их оттуда. Сообщения в очередях могут иметь приоритеты, а сама очередь — реализовывать различные политики поддержания или изменения приоритетов сообщений в ходе работы.

Транзакции

Понятие транзакции пришло в инженерию ПО из бизнеса и используется чаще всего (но все же не всегда) для реализации функциональности, связанной с обеспечением различного рода сделок.

Примером, поясняющим необходимость использования транзакций, является перевод денег с одного банковского счета на другой. При переводе соответствующая сумма должна быть снята с первого счета и добавиться к уже имеющимся деньгам на втором. Если между первой и второй операцией произойдет сбой, например, пропадет связь между банками, деньги исчезнут с первого счета и не появятся на втором, что явно не устроит их владельца. Перестановка операций местами не помогает — при сбое между ними ровно такая же сумма возникнет из ничего на втором счете. В этом случае недоволен будет банк, поскольку он должен будет выплатить эти деньги владельцу счета, хотя сам их не получал.

Выход из этой ситуации один — сделать так, чтобы либо обе эти операции выполнялись, либо ни одна из них не выполнялась. Такое свойство обеспечивается их объединением в одну транзакцию.

Транзакции представляют собой группы действий, обладающие следующим набором свойств.

- *Атомарность (atomicity)*. Для окружения транзакция неделима — она либо выполняется целиком, либо ни одно из ее действий транзакции не выполняется. Другие процессы не имеют доступа к промежуточным результатам транзакции.

- *Непротиворечивость (consistency)*. Транзакция не нарушает инвариантов и ограничений целостности данных системы.
- *Изолированность (isolation)*. Одновременно происходящие транзакции не влияют друг на друга. Это означает, что несколько транзакций, выполнявшихся параллельно, производят такой суммарный эффект, как будто они выполнялись в некоторой последовательности. Сама эта последовательность определяется внутренними механизмами реализации транзакций. Это свойство также называют *сериализуемостью* транзакций, поскольку любой сценарий их выполнения эквивалентен некоторой их последовательности или серии.
- *Долговечность (durability)*. После завершения транзакции сделанные ею изменения становятся постоянными и доступными для выполняемых в дальнейшем операций. Если транзакция завершилась, никакие сбои не могут отменить результаты ее работы.

По первым буквам английских терминов для этих свойств, их часто называют ACID.

Свойствами ACID во всей полноте обладают так называемые *плоские транзакции (flat transactions)*, самый распространенный вариант транзакций. Иногда требуется гораздо более сложное поведение, в рамках которого нужно уметь выполнять или отменять только часть операций в составе транзакции; бывают случаи, когда процессам, не участвующим в транзакции, нужно уметь получить ее промежуточные результаты. Сокрытие промежуточных результатов часто накладывает слишком сильные ограничения на работу системы, если транзакция продолжается заметное время (а иногда их выполнение требует нескольких месяцев!). Для решения таких задач применяются механизмы, допускающие вложенность транзакций друг в друга, длинные транзакции, позволяющие получать доступ к своим промежуточным результатам, и пр.

Одним из широко распространенных видов программного обеспечения промежуточного уровня являются *мониторы транзакций (transaction monitors)*, обеспечивающие выполнение удаленных вызовов процедур с поддержкой транзакций. Такие транзакции часто называют ***распределенными***, поскольку участвующие в них процессы могут работать на разных машинах.

Рисунок 71. Схема реализации поддержки распределенных транзакций.

Для организации таких транзакций необходим *координатор*, который получает информацию обо всех участвующих в транзакции действиях и обеспечивает ее атомарность и изолированность от других процессов. Обычно транзакции реализуются при помощи примитивов, позволяющих

начать транзакцию, *завершить ее успешно (commit)*, с сохранением всех сделанных изменений, и *откатить транзакцию (rollback)*, отменив все выполненные в ее рамках действия.

Примитив «начать транзакцию» сообщает координатору о необходимости создать новую транзакцию, зарегистрировать начавший ее объект как участника и передать ему идентификатор транзакции. При передаче управления (в том числе с помощью удаленного вызова метода) участник транзакции передает вместе с обычными данными ее идентификатор. Компонент, операция которого была вызвана в рамках транзакции, сообщает координатору идентификатор транзакции с тем, чтобы координатор зарегистрировал и его как участника этой же транзакции.

Если один из участников не может выполнить свою операцию, выполняется откат транзакции. При этом координатор рассылает всем зарегистрированным участникам сообщения о необходимости отменить выполненные ими ранее действия.

Если вызывается примитив «завершить транзакцию», координатор выполняет некоторый протокол подтверждения, чтобы убедиться, что все участники выполнили свои действия успешно и можно открыть результаты транзакции для внешнего мира. Наиболее широко используется ***протокол двухфазного подтверждения (Two-phase Commit Protocol, 2PC)*** [3,4], который состоит в следующем.

1. Координатор посылает каждому компоненту-участнику транзакции запрос о подтверждении успешности его действий.
2. Если данный компонент выполнил свою часть операций успешно, он возвращает координатору подтверждение.
   Иначе — он посылает сообщение об ошибке.
3. Координатор собирает подтверждения всех участников и, если все зарегистрированные участники транзакции присылают подтверждения успешности, рассылает им сообщение о подтверждении транзакции в целом. Если хотя бы один участник прислал сообщение об ошибке или не ответил в рамках заданного времени, координатор рассылает сообщение о необходимости отменить транзакцию.
4. Каждый участник, получив сообщение о подтверждении транзакции в целом, сохраняет локальные изменения, сделанные в рамках транзакции.
   Если же он получить сообщение об отмене транзакции, он отменяет локальные изменения.

Аналог протокола двухфазного подтверждения используется, например, в компонентной модели JavaBeans для уведомления об изменениях свойств компонента, которые некоторые из оповещаемых о них компонентов-подписчиков могут отменить [5,6]. При этом до внесения изменений о них надо оповестить с помощью метода `vetoableChange()` интерфейса `java.beans.VetoableChangeListener`. Если хотя бы один из подписчиков требует отменить изменение с помощью создания исключения типа `java.beans.PropertyVetoException`, его надо откатить, сообщив об этом остальным подписчикам. Если же все согласны, то после внесения изменений о них, как об уже сделанных, оповещают с помощью метода `propertyChange()` интерфейса `java.beans.PropertyChangeListener`.

Литература к Лекции 12

[1] C. Szyperski. Component Software Beyond Object-Oriented Programming. Boston, MA: Addison-Wesley and ACM Press, 1998.

[2] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, K. Wallnau. Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition/ Technical Report CMU/SEI-2000-TR-008.
Доступен как http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr008.pdf.

[3] Э. Таненбаум, М. ван Стеен. Распределенные системы. Принципы и парадигмы. СПб.: Питер, 2003.

[4] G. Alonso, F. Casati, H. Kuno, V. Machiraju. Web Services. Concepts, Architectures and Applications. Springer-Verlag, 2004.

[5] JavaBeans Specification 1.01. Доступна через страницу http://java.sun.com/products/javabeans/docs/spec.html.

[6] Документация по библиотекам J2SE http://java.sun.com/j2se/1.5.0/docs/api/index.html.
[7] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. Pattern-Oriented Software Architecture. Volume 2. Patterns for Concurrent and Networked Objects. Wiley, 2000.

Лекция 13. Компонентные технологии разработки Web-приложений

Аннотация

Рассматриваются основные элементы компонентных сред Java 2 Enterprise Edition и .NET. Показывается, как в рамках этих технологий решаются основные задачи построения распределенных Web-приложений.

Ключевые слова

Web-приложения, расширяемый язык разметки XML, XSLT, схема документа XML, Web-контейнер, EJB-контейнер, Web-компоненты J2EE, компоненты EJB, дескрипторы развертывания, Java RMI, JMS, JNDI, JTA, защита на основе ролей, DOM, SAX, StAX, ADO.NET, ASP.NET, конфигурационные файлы .NET, .NET Remoting, Active Directory, зоны приложений.

Текст лекции

Web-приложения

После обзора общих концепций, связанных с компонентными технологиями и распределенным программным обеспечением, отметим дополнительные общие черты таких технологий в их сегодняшнем виде.

Программное обеспечение в современном мире становится все сложнее и приобретает все больше функций. Коммерческие компании и государственные организации стремятся автоматизировать все больше своих процессов, как внутренних, так и тех, что связаны с общением с внешним миром. При этом, однако, разработка таких приложений, их внедрение и поддержка становятся все дороже.

Есть, тем не менее, фактор, который помогает значительно снизить расходы — широчайшее распространение Интернет. Если ваше программное обеспечение использует для связи между своими элементами базовые протоколы Интернет (TCP/IP и HTTP) и предоставляет пользовательский интерфейс с помощью HTML, который можно просматривать в любом браузере, то практически каждый его потенциальный пользователь не имеет технических препятствий для обращения к этому ПО. Не нужно распространять специальные клиентские компоненты, ставить клиентам специальное оборудование, не нужно тратить много времени и средств на обучение пользователей работе со специфическим интерфейсом, настройке связи с серверами и т.д. Интернет предоставляет готовую инфраструктуру для создания крупномасштабных программных систем, в рамках которых десятки тысяч компонентов могли бы работать совместно и миллионы пользователей могли бы получать их услуги.

Поэтому вполне логично, что ***Web-приложения***, т.е. программные системы, использующие для связи протоколы Интернет, а в качестве пользовательского интерфейса — HTML страницы, стали одним из самых востребованных видов ПО. Однако, чтобы сделать потенциальные выгоды от использования Интернет реальными, необходимы технологии разработки Web-приложений, которые позволяли бы строить их на компонентной основе, минимизируя затраты на интеграцию отдельных компонентов, их развертывание и поддержку в рабочем состоянии.

Другим важным фактором является распространение ***расширяемого языка разметки (Extensible Markup Language, XML)*** как практически универсального формата данных. XML предоставляет стандартную лексическую форму для представления текстовой информации различной структуры и стандартные же способы описания этой структуры. Многие аспекты создания и работы Web-приложений связаны с обменом разнообразно структурированными данными между отдельными компонентами или представлением информации об организации, свойствах и конфигурации системы, имеющей гибкую структуризацию. Использование XML позволяет решить часть возникающих здесь проблем.

Поскольку все современные технологии разработки Web-приложений так или иначе используют XML, следующий раздел посвящен основным элементам этого языка.

Расширяемый язык разметки XML

Данный раздел содержит краткий обзор основных конструкций XML, для более глубоко изучения этого языка и связанных с ним технологий см. [1-7].

XML [3-5] является *языком разметки*: различные элементы данных в рамках XML-документов выделяются *тегами* — каждый элемент начинается с открывающего тега `<tag>` и заканчивается закрывающим `</tag>`. Здесь `tag` — идентификатор тега, который обычно является английским словом или набором слов, разделяемых знаками '-', которое(-ые) описывают назначение этого элемента данных. Элементы данных могут быть вложены друг в друга, образуя дерево документа. Кроме того, каждый элемент может иметь набор значений атрибутов, которые представляют собой строки, числа или логические значения. Значения атрибутов для данного элемента помещаются внутри его открывающего тега. Элемент данных, не имеющий вложенных подэлементов, может быть оформлен в виде конструкции `<tag … />`, т.е. не иметь отдельного закрывающего тега.

Ниже приведен пример описания информации о книге в виде XML.

```
<book
    title = "Pattern-Oriented Software Architecture, Volume 1: A System of Patterns"
    ISBN = "047195869"
    year = 1996
    hardcover = true
    pages = 476
    language = "English">
  <author>Frank Buschmann</author>
  <author>Regine Meunier</author>
  <author>Hans Rohnert</author>
  <author>Peter Sommerlad</author>
  <author>Michael Stal</author>
  <publisher
      title = "John Wiley & Sons"
      address = "605 Third Avenue, New York, NY 10158-0012, USA" />
</book>
```

В этом примере тег `<book>`, представляющий описание книги, имеет вложенные теги `<author>` и `<publisher>`, представляющие ее авторов (таких тегов может быть несколько) и издателя. Он также имеет атрибуты `title`, `ISBN`, `year`, `hardcover`, `pages` и `language` (название книги, ее международный стандартный номер, т.е. International Standard Book Number или ISBN, плюс год издания, наличие твердой обложки, число страниц и язык). Тег `<publisher>`, в свою очередь, имеет атрибуты `title` и `address` (название и юридический адрес издательской организации).

Элементы XML-документа, называемые также *сущностями*, являются в некотором смысле аналогами значений структурных типов в .NET, а значения их атрибутов — аналогами соответствующих значений полей. При этом теги играют роль самих типов, а атрибуты и вложенные теги — роль их полей, имеющих, соответственно, примитивные и структурные типы. Расширяемым XML назван потому, что можно задать специальную структуру тегов и их атрибутов для некоторого вида документов. Эта структура описывается в отдельном документе, называемом *схемой*, который сам написан на специальном подмножестве XML, *DTD (Document Type Declaration, декларация типа документа)* [3-5] или *XMLSchema* [6].

XML-документ всегда начинается заголовком, описывающим версию XML, которой соответствует документ, и используемую кодировку. По умолчанию используется кодировка UTF-8.

Затем чаще всего идет описание типа документа, указывающее схему, которой он соответствует, и тег корневого элемента, содержащего все остальные элементы данного документа. Схема может задаваться в формате DTD или XMLSchema. Второй, хотя и является более новым, пока еще используется реже, потому что достаточно много документов определяется с помощью DTD и очень многие инструменты для обработки XML могут пользоваться этим форматом. Используемая схема определяется сразу двумя способами — при помощи строки, которая может служить ключом для поиска схемы на данной машине, и при помощи унифицированного идентификатора документа (Unified Resource Identifier, URI), содержащего ее описание и используемого в том случае, если ее не удалось найти локально.

Ниже приводится пример заголовка и описания типа документа для дескриптора развертывания EJB компонентов (см. подробности далее).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 8.1
EJB 2.1//EN" "http://www.sun.com/software/appserver/dtds/sun-ejb-jar_2_1-1.dtd">
<sun-ejb-jar>
…
</sun-ejb-jar>
```

Другой пример показывает заголовок документа DocBook — основанного на XML формата для технической документации, которая может быть автоматически преобразована в HTML, PDF и другие документы с определенными для них правилами верстки.

```
<?xml version="1.0" encoding="windows-1251"?>
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.3//EN"
"http://www.oasis-open.org/docbook/xml/4.3/docbookx.dtd">
<article>
…
</article>
```

Помимо элементов данных и заголовка с описанием типа документа, XML-документ может содержать комментарии, помещаемые в теги `<!-- … -->`, *инструкции обработки* вида `<? processor-name … ?>` (здесь `processor-name` — идентификатор обработчика, которому предназначена инструкция) и *секции символьных данных* CDATA, которые начинаются набором символов `<![CDATA[`, а заканчиваются с помощью `]]>`. Внутри секций символьных данных могут быть любые символы, за исключением закрывающей комбинации. В остальных местах некоторые специальные символы должны быть представлены комбинациями символов в соответствии с Таблицей 11.

Символ	Представление в XML
<	`<`
>	`>`
&	`&`
"	`&quot;`
'	`&apos;`

Таблица 11. Представления специальных символов в XML.

XML содержит много других конструкций, помимо уже перечисленных, но их рассмотрение выходит за рамки данного курса. Читатель, желающий узнать больше об этом языке и связанных с ним технологиях, может обратиться к [1-7].

Платформа Java 2 Enterprise Edition

Платформа J2EE предназначена в первую очередь для разработки распределенных Web-приложений и поддерживает следующие 4 вида компонентов [8].

- ***Enterprise JavaBeans (EJB).***
  Компоненты EJB предназначены для реализации на их основе бизнес-логики приложения и операций над данными. Любые компоненты, разработанные на Java, принято называть бинами (bean, боб или фасолина, в разговорном языке имеет также значения головы и монеты). Компоненты Enterprise JavaBean отличаются от «обычных» тем, что работают в рамках ***EJB-контейнера***, который является для них компонентной средой. Он поддерживает следующие базовые службы при работе с компонентами EJB.
  - Автоматическую поддержку обращений к компонентам, размещенным на разных машинах.
  - Автоматическую поддержку транзакций.
  - Автоматическую синхронизацию состояния баз данных и соответствующих компонентов EJB в обе стороны.

- Автоматическую поддержку защищенности за счет аутентификации пользователей, проверки прав пользователей или компонентов на выполнение выполняемых ими операций и авторизации производимых действий.
- Автоматическое управление жизненным циклом компонента (последовательностью переходов между состояниями типа «отсутствует»-«инициализирован»-«активен») и набором компонентов как ресурсами: удаление компонентов, ставших ненужными; загрузку новых компонентов; балансировку нагрузки между имеющимися компонентами; использование пула готовых к работе, но не инициализированных компонентов, чтобы не тратить время на их удаление и создание, и пр.

В целом EJB-контейнер представляет собой пример ***объектного монитора транзакций (object transaction monitor)*** — ПО промежуточного уровня, поддерживающего в рамках объектно-ориентированной парадигмы удаленные вызовы методов и распределенные транзакции.

- ***Web-компоненты (Web components).***
  Эти компоненты служат для предоставления интерфейса к корпоративным программным системам поверх широко используемых протоколов Интернет, а именно, HTTP. Предоставляемые интерфейсы могут быть как интерфейсами для людей (WebUI), так и специализированными программными интерфейсами, работающими подобно удаленному вызову методов, но поверх HTTP.
  В группу Web-компонентов входят *фильтры (filters)*, *обработчики Web-событий (web event listeners)*, ***сервлеты (servlets)*** и ***серверные страницы Java (JavaServer Pages, JSP)***.
  Компонентной средой для работы Web-компонентов служит ***Web-контейнер***, поставляемый в рамках любой реализации платформы J2EE. Web-контейнер реализует такие службы, как управление жизненным циклом компонентов и набором компонентов как ресурсом, распараллеливание независимых работ, выполнение удаленных обращений к компонентам, поддержка защищенности с помощью проверки прав компонентов и пользователей на выполнение различных операций.
- Обычные приложения на Java.
  J2EE является расширением J2SE и поэтому все Java приложения могут работать и в этой среде. Однако, в дополнение к обычным возможностям J2SE, эти приложения могут использовать в своей работе Web-компоненты и EJB, как напрямую, так и удаленно, связываясь с ними по HTTP.
- Аплеты (applets).
  Это небольшие компоненты, имеющие графический интерфейс пользователя и предназначенные для работы внутри стандартного Web-браузера. Они используются в тех случаях, когда не хватает выразительных возможностей пользовательского интерфейса на базе HTML, и могут связываться с удаленными Web-компонентами, работающими на сервере, по HTTP.

Компонент любого из этих видов оформляется как небольшой набор классов и интерфейсов на Java, а также имеет *дескриптор развертывания (deployment descriptor)* — описание в определенном формате на основе XML конфигурации компонента в рамках контейнера, в который он помещается. Приложение в целом также имеет дескриптор развертывания. Дескрипторы развертывания играют важную роль, позволяя менять некоторые параметры функционирования компонента и привязывать их к параметрам среды, в рамках которой компонент работает, не затрагивая его код.

Платформа J2EE приспособлена для разработки многоуровневых Web-приложений. При работе с такими приложениями пользователь формирует свои запросы, заполняя HTML-формы в браузере, который упаковывает их в HTTP-сообщения и пересылает Web-серверу. Web-сервер передает эти сообщения Web-компонентам, выделяющим из них исходные запросы пользователя и передающим их для обработки компонентам EJB. Результаты работы EJB компонентов превращаются Web-компонентами в динамически генерируемые HTML-страницы, и отправляются

обратно пользователю, представая перед ним в окне браузера. Аплеты используются там, где нужен более функциональный интерфейс, чем стандартные формы и страницы HTML.

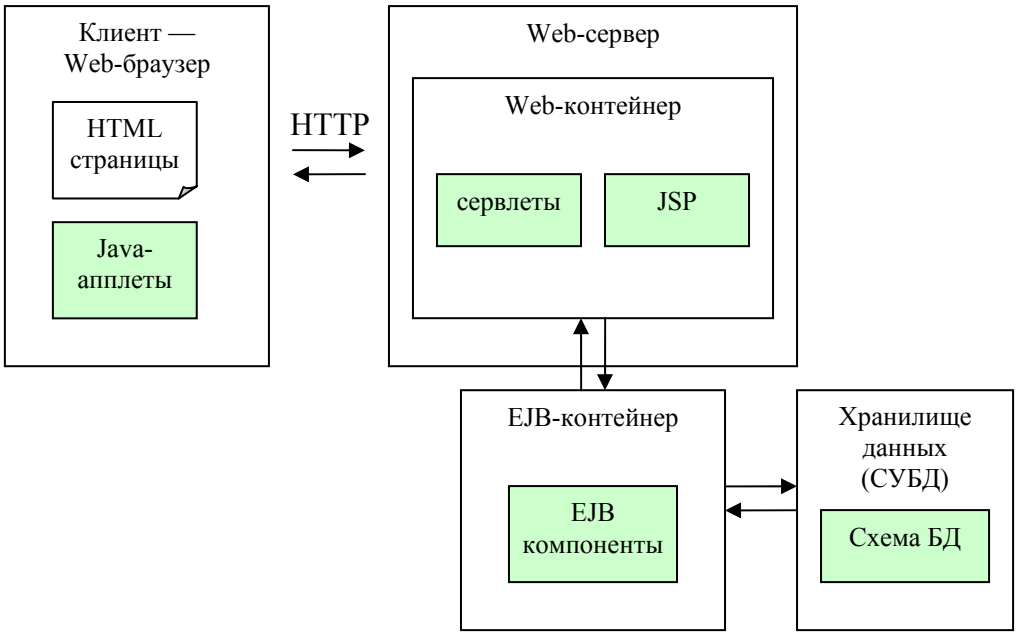


Рисунок 72. Типовая архитектура J2EE приложения.
Выделены компоненты, разрабатываемые вручную.

Таким образом, приложения на базе J2EE строятся с использованием трех основных архитектурных стилей.

- *Многоуровневая система.*
  Самые крупные подсистемы организованы как уровни, решающие различные задачи.
  - Интерфейс взаимодействия с внешней средой, включая пользователей, реализуется при помощи Web-компонентов.
  - Уровень бизнес-логики и модели данных реализуется при помощи EJB компонентов.
  - Уровень управления ресурсами строится на основе коммерческих систем управления базами данных (СУБД). Можно также подключать другие виды ресурсов, для которых имеется реализация интерфейса поставщика служб J2EE (J2EE service provider interface, J2EE SPI).
- *Независимые компоненты.*
  Первые два уровня построены из отдельных компонентов, каждый из которых имеет собственную область ответственности, но может привлекать для решения частных задач другие компоненты.
- *Данные-представление-обработка (MVC).*
  Работа компонентов в рамках обработки группы тесно связанных запросов организуется по образцу MVC. При этом сервлеты и обработчики Web-событий служат обработчиками, компоненты JSP — представлением, а компоненты EJB — моделью данных.

Рассмотрим теперь, как решаются общие задачи построения распределенных систем [9] на базе платформы J2EE.

Связь

Связь между компонентами, работающими в разных процессах и на разных машинах, обеспечивается в J2EE, в основном, двумя способами: синхронная связь — с помощью реализации удаленного вызова методов на Java (*Java RMI*), асинхронная — с помощью *службы сообщений Java (Java message service, JMS)*.

Java RMI [10] является примером реализации общей техники удаленного вызова методов. Базовые интерфейсы для реализации удаленного вызова методов на Java находятся в пакете `java.rmi` стандартной библиотеки.

Набор методов некоторого класса, доступных для удаленных вызовов, выделяется в специальный интерфейс, называемый ***удаленным интерфейсом (remote interface)*** и наследующий `java.rmi.Remote`. Сам класс, методы объектов которого можно вызвать удаленно, должен реализовывать этот интерфейс. Этот же интерфейс реализуется автоматически создаваемой клиентской заглушкой. Поэтому объекты-клиенты работают только с объектом этого интерфейса, а не с объектом класса, реализующего декларированные в таком интерфейсе операции.

Кроме того, класс, реализующий удаленный интерфейс, должен наследовать классу `java.rmi.server.RemoteObject`. Этот класс содержит реализации методов `hashCode()`, `equals()` и `toString()`, которые учитывают возможность размещения его объектов в процессах, отличных от того, где они вызываются. Обычно наследуется не сам этот класс, а его подклассы `java.rmi.server.UnicastRemoteObject` или `java.rmi.activation.Activatable`. Первый реализует общую функциональность объектов, которые можно вызвать удаленно поверх транспортного протокола TCP, пока работает содержащий их процесс, включая занесение информации о таких объектах в реестр RMI (собственная служба именования в рамках Java RMI). Второй класс служит для реализации ***активизируемых объектов (activatable objects)***, которые создаются системой «по требованию» — как только кто-нибудь вызывает в них какой-нибудь метод. Ссылки на такие объекты могут сохраняться, а обратиться к ним можно спустя долгое время, даже после перезагрузки системы.

Каждый ***удаленный метод (remote method)***, т.е. метод, который можно вызвать из другого процесса, должен декларировать в качестве класса возможных исключений `java.rmi.RemoteException` или его базовые типы `java.io.IOException` или `java.lang.Exception`. Этот класс сам служит базовым для классов исключений, представляющих ошибки связи, ошибки маршалинга параметров или результатов и ошибки протоколов, реализующих RMI.

Объекты, реализующие один из удаленных интерфейсов, могут быть переданы в качестве параметров или результатов удаленных методов по ссылке как объекты этого интерфейса. При этом в другой процесс передается байт-код клиентской заглушки, связанной с таким объектом, и объекты этого процесса получают возможность вызывать в нем методы.

Остальные объекты передаются как параметры или результаты удаленных вызовов с помощью сериализации их данных и построения копии такого объекта в другом процессе. Так же передаются и создаваемые удаленным методом исключения. Для этого им необходимо реализовывать интерфейс `java.io.Serializable` или же быть значениями примитивных типов.

Простой пример Java классов, взаимодействующих по RMI, приведен ниже.

Код удаленного интерфейса.

```
package examples.rmi;

public interface Hello extends java.rmi.Remote
{
  public String hello() throws java.rmi.RemoteException;
}
```

Код реализации удаленного интерфейса.

```
package examples.rmi;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements Hello
{
  public static final String ServerHost = "hostname";
  public static final String ServerURL = "rmi://" + ServerHost + ":2001/SERVER";
```

```
  public static final int RegistryPort = 2000;

  public HelloImpl () throws java.rmi.RemoteException { }

  public String hello () throws java.rmi.RemoteException
  { return "Hello!"; }

  public static void main (String[] args)
  {
    try
    {
      Hello stub = new HelloImpl();
      Registry registry = LocateRegistry.getRegistry(RegistryPort);
      registry.rebind(ServerURL, stub);
    }
    catch (Exception e)
    {
      System.out.println("server creation exception");
      e.printStackTrace();
    }
  }
}
```

Код класса-клиента.

```
package examples.rmi;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class HelloClient
{
  public HelloClient () { }

  public static void main (String[] args)
  {
    try
    {
      Registry registry = LocateRegistry.getRegistry
                          (HelloImpl.ServerHost, HelloImpl.RegistryPort);
      Hello stub = (Hello) registry.lookup(HelloImpl.ServerURL);
      System.out.println("response: " + stub.hello());
    }
    catch (Exception e)
    {
      System.err.println("Client exception: " + e.toString());
      e.printStackTrace();
    }
  }
}
```

Для того чтобы запустить этот пример, нужно выполнить следующие действия.

1. Скомпилировать все классы и интерфейсы. В классе-реализации серверного компонента константа `ServerHost` должна быть инициализирована именем машины, на которой будет работать сервер.
2. Создать и скомпилировать заглушки при помощи компилятора Java RMI, запустив его в корневой директории для кода на Java.
   ```
   rmic examples.rmi.HelloImpl
   ```
3. Запустить реестр RMI на той машине, на которой будет выполняться серверный компонент `HelloImpl`. Реестр используется для регистрации серверного объекта и последующего поиска его клиентами и в данном примере принимает сообщения по порту 2000.
   ```
   start rmiregistry 2000 &
   ```
4. Запустить сам серверный компонент.
   ```
   java examples.rmi.HelloImpl
   ```

5. Запустить клиентский компонент на той же машине или на другой.
   `java examples.rmi.HelloClient`
   Если никаких ошибок не сделано, и порты 2000 и 2001 на серверной машине свободны, клиент выдаст сообщение.
   `response: Hello!`

Поскольку базовые интерфейсы компонентов EJB наследуют `java.rmi.Remote`, такие компоненты автоматически предоставляют возможность обращаться к некоторым из своих методов удаленно.

Служба сообщений Java (JMS) [11] представляет собой спецификацию интерфейсов для поддержки передачи сообщений (в частности, асинхронных) между компонентами в рамках платформы J2EE. Ее базовые библиотеки — пакет `javax.jms` — не входят в состав J2SE. Та или иная реализация JMS должна входить в любую реализацию платформы J2EE.

Основные элементы JMS следующие.

- *Сообщение*. Все объекты-сообщения реализуют интерфейс `javax.jms.Message`. Сообщение имеет тело, которое может быть отображением (map) ключей в значения, текстом, объектом, набором байт или потоком значений примитивных типов Java. Каждый из видов сообщений представлен особым подинтерфейсом общего интерфейса `Message`. Сообщение может иметь набор заголовков (headers), большинство из которых определяются автоматически.
  Кроме того, сообщение может иметь набор свойств, которые позволяют определять дополнительные заголовки, специфичные для данного приложения.
- Клиент может передать сообщение, установив *соединение* с провайдером JMS. Соединения представляются с помощью объектов интерфейса `javax.jms.Connection`, а получить соединение можно с помощью фабрики соединений (объект `javax.jms.ConnectionFactory`), найдя ее при помощи службы именования. Передать сообщение можно и воспользовавшись *объектом-адресатом* (объект `javax.jms.Destination`), который также можно получить через службу именования.
- JMS поддерживает как связь *точка-точка (peer-to-peer, P2P)*, так и посылку и прием сообщений по схеме *подписки/публикации*. Основные интерфейсы JMS (соединение, фабрика соединений, адресат, сессия и пр.) имеют специфические подинтерфейсы для каждой из этих двух моделей связи. В клиентских программах рекомендуется всегда использовать общие интерфейсы.

Именование

Поиск ресурсов по именам или идентификаторам и набору их свойств в рамках J2EE и J2SE осуществляется при помощи интерфейса JNDI (Java Naming and Directory Interface, Java интерфейс служб имен и каталогов) [12].

Интерфейсы и классы JNDI находятся в пакете `javax.namimg` и его подпакетах `javax.naming.directory, javax.naming.event, javax.naming.ldap`.

Основные сущности служб именования и каталогов, хранящие привязку ресурсов к именам и наборам атрибутов, называются *контекстами*. Они представляются объектами интерфейса `javax.naming.Context`, в частности, реализующих его классов `javax.naming.InitialContext, javax.naming.directory.InitialDirContext, javax.naming.ldap.InitialLdapContext`.

Методы этого интерфейса служат для привязки объекта к имени (`void bind(String, Object), void rebind(String, Object)` — связать данное имя с данным объектом, даже если оно уже имеется в этом контексте), поиска объекта по имени (`Object lookup (String)`), разрыва связи между именем и объектом (`void unbind(String)`) и пр.

В дополнение к этим методам классы `InitialDirContext` и `InitialLdapContext` реализуют интерфейс контекста службы каталогов `DirContext`. Контекст службы каталогов имеет методы `void bind(String, Object, Attributes)` для привязки набора атрибутов к объекту, `Attributes getAttributes(String)` — для получения набора атрибутов объекта по указанному имени и

`NamingEnumeration<SearchResult> search(String, Attributes)` — для поиска объектов по указанному набору атрибутов в контексте с указанным именем.

Класс `InitialLdapContext` реализует общий протокол работы со службами каталогов — *простой протокол доступа к службам каталогов (Lightweight Directory Access Protocol, LDAP)*.

При загрузке виртуальной машины механизм инициализации JNDI конструирует начальный контекст по JNDI свойствам, задаваемым во всех файлах с именем `jndi.properties`, которые находятся в директориях, перечисленных в classpath.

Стандартный набор JNDI свойств, которые могут быть установлены для Java приложения или аплета, включает следующие:

- `java.naming.factory.initial` (соответствует константе `Context.INITIAL_CONTEXT_FACTORY`) — имя класса фабрики для создания начальных контекстов, обязательно должно быть установлено;
- `java.naming.provider.url` (соответствует константе `Context.PROVIDER_URL`) — URL сервера каталогов или имен;
- `java.naming.dns.url` (соответствует константе `Context.DNS_URL`) — URL для определения DNS узла, используемого для получения адреса JNDI URL;
- `java.naming.applet` (соответствует константе `Context.APPLET`) — объект-апплет, используемый для получения JNDI свойств;
- `java.naming.language` (соответствует константе `Context.LANGUAGE`) — список, через запятую, предпочтительных языков для использования в данной службе (пример: `en-US, fr, ja-JP-kanji`). Языки описываются в соответствии с RFC 1766 [13].

Ниже приведен пример использования JNDI для распечатки содержимого директории c:/tmp. Для работы с файловой системой через JNDI используется реализация службы именования на основе файловой системы от Sun [14].

```
package examples.jndi;

import java.util.Properties;

import javax.naming.Binding;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;

public class JNDIExample
{
  public static void main (String[] args)
  {
    Properties env = new Properties();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
    env.put(Context.PROVIDER_URL, "file://c:/tmp");

    try
    {
      Context cntx = new InitialContext(env);
      NamingEnumeration list = cntx.listBindings("");

      while(list.hasMore())
      {
        Binding bnd = (Binding)list.next();
        System.out.println(bnd.getName());
      }
    }
    catch (NamingException e) { e.printStackTrace(); }
  }
}
```

Процессы и синхронизация

Разбиение J2EE приложения на набор взаимодействующих процессов и потоков и управление ими осуществляется их Web- и EJB-контейнерами автоматически. На их работу можно влиять с помощью конфигурирования J2EE-сервера в целом и конкретных приложений.

Все методы вспомогательных классов, которые используются Web-компонентами или компонентами EJB, нужно объявлять синхронизированными (**`synchronized`**).

Компоненты J2EE-приложений, работающие в рамках контейнеров, могут создавать собственные отдельные потоки, но делать это нужно с большой осторожностью, поскольку этими потоками контейнер управлять не сможет и они могут помешать работе других компонентов.

Целостность

Целостность и непротиворечивость данных при работе J2EE приложений поддерживается с помощью механизма распределенных транзакций. Управление такими транзакциями может быть возложено на EJB-контейнер, что делается с помощью определения политик участия методов EJB-компонентов в транзакциях в их дескрипторах развертывания, или может осуществляться вручную. В обоих случаях используются механизмы, реализующие *интерфейсы управления транзакциями Java (Java Transaction API, JTA)*.

Базовые интерфейсы JTA находятся в пакетах `javax.transaction.` и `javax.transaction.xa`. Это, прежде всего, интерфейсы менеджера транзакций `TransactionManager`, самих транзакций `Transaction` и `UserTransaction` и интерфейс синхронизации `Synchronization`, позволяющий получать уведомление о начале завершения и конце завершения транзакций.

Методы интерфейса `TransactionManager` позволяют запустить транзакцию, завершить ее успешно или откатить, а также получить объект, представляющий текущую транзакцию и имеющий тип `Transaction`. Методы интерфейса `Transaction` позволяют завершить или откатить транзакцию, представляемую объектом такого интерфейса, зарегистрировать объекты для синхронизации при завершении транзакции, а также добавить некоторые ресурсы в число участников данной транзакции или удалить их из этого списка. Такие ресурсы представляются в виде объектов интерфейса `javax.transaction.xa.XAResource`. Интерфейс `UserTransaction` предназначен для управления пользовательскими транзакциями — он предоставляет немного меньше возможностей, чем `TransactionManager`.

В том случае, если управление транзакциями целиком поручается EJB-контейнеру (это так называемые *транзакции, управляемые контейнером, container managed transactions*), влиять на их ход можно, указывая в дескрипторах развертывания EJB-компонентов различные *транзакционные атрибуты (transaction attributes)* для их методов. Транзакционный атрибут может принимать одно из следующих значений.

- `Required`
  Метод, имеющий такой атрибут, всегда должен выполняться в контексте транзакции. Он будет работать в контексте той же транзакции, в которой работал вызвавший его метод, а если он был вызван вне контекста транзакции, с началом его работы будет запущена новая транзакция.
  Этот атрибут используется наиболее часто.
- `RequiresNew`
  Метод, имеющий такой атрибут, всегда будет запускать новую транзакцию в самом начале работы. При этом внешняя транзакция, если она была, будет временно приостановлена.
- `Mandatory`
  Метод, имеющий такой атрибут, должен вызываться только из транзакции, в контексте которой он и продолжит работать. При вызове такого метода извне транзакции будет создана исключительная ситуация типа `TransactionRequiredException`.
- `NotSupported`
  При вызове такого метода внешняя транзакция, если она есть, будет временно приостановлена. Если ее нет, новая транзакция не будет запущена.

- `Supports`
  Такой метод работает внутри транзакции, если его вызвали из ее контекста; если же он был вызван вне транзакции, новая транзакция не запускается.
- `Never`
  При вызове такого метода из транзакции создается исключительная ситуация типа `RemoteException`. Он может работать, только будучи вызван извне транзакции.

Откатить автоматически управляемую транзакцию можно, создав исключительную ситуацию типа `javax.ejb.EJBException` или вызвав метод `setRollbackOnly()` интерфейса `javax.ejb.EJBContext`.

Отказоустойчивость

Отказоустойчивость J2EE приложений не обеспечивается дополнительными средствами, такими, как репликация, надежная передача сообщений и пр. Если в них возникает необходимость, разработчик должен сам реализовывать эти механизмы либо пользоваться готовыми решениями за рамками платформы J2EE.

Защита

Защищенность J2EE приложения поддерживается несколькими способами.

- С помощью определения методов *аутентификации*, т.е. определения идентичности пользователей. Эти методы определяются в дескрипторе развертывания приложения. Можно использовать следующие способы аутентификации.
  - Отсутствие аутентификации.
  - С помощью базового механизма протокола HTTP. При попытке обращения к ресурсу по протоколу HTTP будет запрошено имя пользователя и пароль, которые будут проверены Web-сервером. Этот способ не слишком хорошо защищен, поскольку реквизиты пользователя пересылаются по сети в незашифрованном виде.
  - С помощью дайджеста (digest). Этот метод работает так же, как базовый механизм аутентификации по HTTP, но имя и пароль пользователя пересылаются в зашифрованном виде. Такой способ используется достаточно редко.
  - С помощью специальной формы. При этом определяется страница, на которой расположена форма аутентификации (обычно это те же поля для ввода имени пользователя и пароля, но, может быть, и каких-то других его атрибутов), и страница, на которой находится сообщение, выдаваемое при неудачной аутентификации.
  - С использованием сертификата клиента. Этот метод требует использовать протокол HTTPS. Клиент должен предоставить свой сертификат или открытый ключ, удовлетворяющий стандарту X.509 на инфраструктуру открытых ключей. Можно использовать и взаимную аутентификацию — в этом случае и клиент, и сервер предоставляют свои сертификаты.
- С помощью соединений по протоколу HTTP поверх уровня защищенных сокетов (Secure Socket Layer, SSL, на HTTP поверх SSL часто ссылаются с помощью отдельной аббревиатуры HTTPS).
  Можно потребовать использовать только такие соединения, указав атрибуты CONFIDENTIAL и/или INTEGRAL в дескрипторе развертывания приложения. Первый атрибут означает, что данные, передаваемые между клиентом и приложением, будут зашифрованы, так что их тяжело будет прочитать третьей стороне. Второй атрибут означает, что эти данные будут сопровождаться дополнительной информацией, гарантирующей их целостность, т.е. то, что они не были подменены где-то между участвующими в связи сторонами.
- С помощью механизма описания *ролей*, определения доступности различных методов Web-компонентов и EJB-компонентов для разных ролей, а также задания политик переноса или создания ролей при совместной работе нескольких методов. Роли, политики их переноса и

правила доступа различных ролей к методам описываются в дескрипторах развертывания компонентов.
При развертывании приложения зарегистрированные на J2EE-сервере пользователи и группы пользователей могут быть отображены на различные роли, определенные в приложении.

- С помощью определения ограничений доступа к наборам ресурсов, задаваемых в виде списков унифицированных идентификаторов ресурсов (URI) или шаблонов URI. Эти ограничения описываются в дескрипторе развертывания приложения и определяют роли и разрешенные им виды прямого доступа (не через обращение к другим компонентам) к данному набору URI.
- С помощью программного определения ролей и пользователей, от имени которых работает текущий поток, из кода самих компонентов.
  Это можно делать при помощи методов `isUserInRole()` и `getUserPrincipal()` интерфейса `HTTPServletRequest`, используемого для представления запросов к Web-компонентам, и аналогичных методов `isCallerInRole()` и `getCallerPrincipal()` интерфейса `EJBContext`, используемого для описания контекста выполнения методов EJB-компонентов.

Работа с XML

Поскольку сейчас очень часто информация хранится и передается в виде XML-документов, для разработки Web-приложений большое значение имеют средства работы с XML. Основными элементами, необходимыми для облегчения работы с XML-документами, являются их разбор и внутреннее представление XML-данных.

Библиотеки для работы с XML-документами находятся в пакетах `javax.xml` , `org.w3c.dom` и `org.xml.sax`, а также вложенных в них пакетах. В этих пакетах определяются следующие интерфейсы.

- Общий интерфейс обработчиков XML находится в пакете `javax.xml.parsers`. Такие обработчики могут быть основаны на *простом интерфейсе работы с XML (Simple API for XML, SAX)* [15] или на *объектной модели документов (Document Object Model, DOM)* [16]. Оба этих подхода основаны на стандартах W3C.
- Простой интерфейс для работы с XML (SAX) [15] определяется в пакете `org.xml.sax`. Это интерфейс, основанный на событиях, — XML-парсер, реализующий его, последовательно разбирает XML-данные и генерирует очередное событие в зависимости от вида обнаруженной конструкции. Для работы с XML-документами необходимо написать ряд обработчиков таких событий, являющихся реализациями интерфейса `org.xml.sax.ContentHandler`.
- Объектная модель документов (DOM) [16] представляет собой интерфейс работы с XML-документом, представленным в виде дерева его элементов. Java-представление этого интерфейса описано в пакете `org.w3c.dom`. Одной из его реализаций является JDOM [17], а dom4j [18] предоставляет несколько упрощенную и более удобную с точки зрения Java, но не вполне соответствующую стандарту DOM реализацию.
- Обработка XML-документов может быть построена на базе *расширяемого языка трансформаций на основе таблиц стилей (Extensible Stylesheet Language Transformations, XLST)* [19]. При этом процесс обработки документов описывается в виде XSLT-программы, которая затем подается на вход интерпретатору XSLT (XSLT-процессору) вместе с обрабатываемым XML-документом. Интерфейсы для работы с XSLT определены в пакете `javax.xml.transform`. Широко используемыми XSLT-процессорами являются Saxon [20] и Xalan [21].
- В новую версию J2EE 5.0, ожидаемую в 2006 году, должны войти интерфейсы для *потоковой обработки XML-документов (Streaming API for XML, StAX)*. В этом подходе XML-документ рассматривается как поток различных конструкций, которые становятся

доступными по запросу (pull-модель), в отличие от работы на основе событий в SAX, когда каждая конструкция порождает событие, которое нужно обработать (push-модель). Обработка XML-документов в стиле StAX гораздо более гибкая, чем в SAX, и вполне сравнима с ней по удобству. В настоящее время уже доступны спецификации интерфейсов StAX [22] и несколько их реализаций.

Платформа .NET

Среда .NET предназначена для более широкого использования, чем платформа J2EE. Однако ее функциональность в части, предназначенной для разработки распределенных Web-приложений, очень похожа на J2EE.

В рамках .NET имеются аналоги основных видов компонентов J2EE. Web-компонентам соответствуют компоненты, построенные по технологии ASP.NET, а компонентам EJB, связывающим приложение с СУБД, — компоненты ADO.NET. Компонентная среда .NET обычно рассматриваются как однородная. Однако существующие небольшие отличия в правилах, управляющих созданием и работой компонентов ASP.NET и остальных, позволяют предположить, что в рамках .NET присутствует аналог Web-контейнера, отдельная компонентная среда для ASP.NET, и отдельная — для остальных видов компонентов. Тем не менее, даже если это так, эти среды тесно связаны и, как и контейнеры J2EE, позволяют взаимодействовать компонентам, размещенным в разных средах. Компонентная среда для ASP.NET, в отличие от Web-контейнера в J2EE, поддерживает автоматические распределенные транзакции.

Тем самым, типовая архитектура Web-приложений на основе .NET может быть изображена так, как это сделано на Рис. 73.

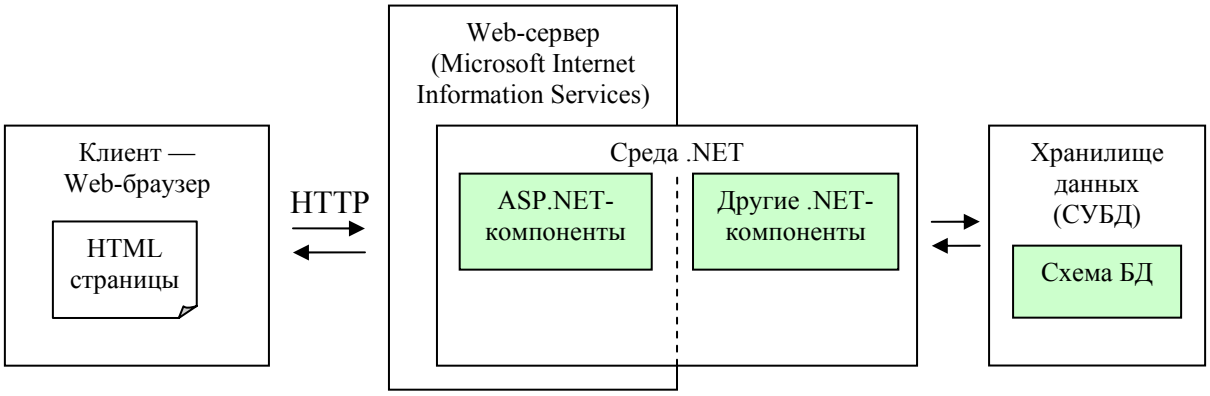


Рисунок 73. Типовая архитектура Web-приложения на основе .NET.

В целом, Web-приложения на основе .NET используют тот же набор архитектурных стилей, что и аналогичные J2EE-приложения.

Обычно .NET-компоненты представляют собой набор .NET-классов и *конфигурационных файлов*, играющих роль дескрипторов развертывания и также представленных в некотором формате на основе XML. Для приложений в целом тоже пишутся особые конфигурационные файлы.

Связь

Связь между компонентами в рамках .NET осуществляется при помощи механизма Remoting, реализующего как RMI, так и асинхронную передачу сообщений.

Классы и интерфейсы, служащие основой механизма Remoting, находятся в пространстве имен `System.Rutime.Remoting` и его подпространствах, в сборках mscorlib и System.Rutime.Remoting.

В рамках Remoting объекты, которые могут участвовать в качестве целей удаленного вызова, его параметров и результатов, делятся на ***передаваемые по значению (marshal-by-value)*** и ***передаваемые по ссылке (marshal-by-reference)***.

Тип объекта, передаваемого по значению, должен реализовывать интерфейс `System.Runtime.Serialization.ISerializable` или должен быть помечен атрибутом `System.SerializableAttribute`. В последнем случае .NET автоматически создает код, необходимый для сериализации или десериализации данных объекта. Исключения, которые могут быть созданы методом, вызываемым удаленно, также должны быть передаваемы по значению.

Объекты, передаваемые по ссылке, должны наследовать классу `System.MarshalByRefObject`. При передаче такого объекта как аргумента или результата в другом процессе (или зоне приложения, см. следующий раздел) создается клиентская заглушка, связанная с этим объектом и называемая в .NET ***посредником (proxy)***.

Более тонкую настройку удаленных вызовов можно делать при помощи наследников класса `System.ContextBoundObject`. Наследование этого класса говорит о том, что удаленные вызовы в объекте этого класса должны происходить в рамках некоторого контекста, являющегося частью его зоны приложения. Примером такого контекста служит контекст транзакции. При вызовах таких объектов .NET проводит дополнительные проверки, связанные с разницей контекстов между вызывающим объектом и вызывающим.

Ниже находится пример взаимодействующих по механизму Remoting классов, имеющий ту же функциональность, что и пример, приведенный для Java RMI.

Код класса, реализующего метод для удаленных обращений.

```
using System;

namespace Examples.Remoting
{
  public class Hello : MarshalByRefObject
  {
    public String HelloMethod() { return "Hello!"; }
  }
}
```

Код сервера, ожидающего обращения клиентов.

```
using System;
using System.Runtime.Remoting;

namespace Examples.Remoting
{
  public class HelloImpl
  {
    public static void Main()
    {
      RemotingConfiguration.Configure("RemotingServer.exe.config");
      Console.ReadLine();
    }
  }
}
```

Код клиентского класса.

```
using System;
using System.Runtime.Remoting;

namespace Examples.Remoting
{
  public class HelloClient
  {
    public static void Main()
    {
      RemotingConfiguration.Configure("RemotingClient.exe.config");
      Hello stub = new Hello();
      Console.WriteLine("response: " + stub.HelloMethod());
    }
  }
}
```

Кроме классов, должны быть написаны конфигурационные файлы серверного и клиентского приложений. Конфигурационный файл сервера выглядит так.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown
          mode = "Singleton"
          type = "Examples.Remoting.Hello, Hello"
          objectUri = "MessageServer"
        />
      </service>
      <channels>
        <channel ref = "http" port = "8989" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Конфигурационный файл клиента.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown
          type = "Examples.Remoting.Hello, Hello"
          url = "http://hostname:8989/MessageServer"
        />
        </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Чтобы запустить этот пример, нужно выполнить следующие шаги.

- Скомпилировать все классы (предполагается, что классы находятся в файлах с именами Hello.cs, Server.cs и Client.cs).
  ```
  csc.exe /noconfig /t:library Hello.cs
  csc.exe /noconfig /r:Hello.dll Server.cs
  csc.exe /noconfig /r:Hello.dll Client.cs
  ```
  В результате первой команды будет получена динамическая библиотека, содержащая определение класса `Hello`. В двух других она используется, чтобы предоставить ссылку на этот тип.
- Запустить серверный компонент
  ```
  Server.exe
  ```
- Перенести на другую машину или в отдельную директорию на той же машине файлы Client.exe, Hello.dll, RemotingClient.exe.config, заменив `hostname` в URL в конфигурационном файле клиента на имя машины, где работает сервер, и запустить клиентский компонент
  ```
  Client.exe
  ```
  Если никаких ошибок не сделано, и порт 8989 на серверной машине доступен, клиент выдаст сообщение.
  ```
  response: Hello!
  ```

С помощью конфигурационных файлов можно определить политику активации серверного объекта (использовать один объект для всех вызовов, создавать для каждого вызова свой объект, создавать объект при его создании клиентом), транспортный протокол для передачи сообщений (HTTP или TCP), и пр.

Асинхронное взаимодействие в рамках .NET может быть реализовано с помощью имеющихся в библиотеке .NET асинхронных делегатов, на основе асинхронных удаленных вызовов по Remoting или с помощью обмена сообщениями на базе библиотек `System.Messaging`.

Взаимодействие между компонентами ASP.NET использует похожие, но скрытые от разработчика компонентов механизмы.

Именование

В примере на взаимодействие по Remoting использовалась локальная служба именования, встроенная в среду .NET — поэтому никаких специальных действий по регистрации и поиску компонентов не производилось. Эта служба позволяет не заботиться об этих вопросах в том случае, если физическое положение компонентов, с которыми необходимо установить связь, известно и постоянно.

Если же это не так, необходимо иметь полноценную службу именования и/или службу каталогов. В этом качестве в .NET используется Active Directory. Поскольку эта технология появилась раньше, чем среда .NET, в рамках .NET была создана библиотека адаптеров, позволяющих использовать функции Active Directory. Элементы этой библиотеки находятся в пространстве имен `System.DirectoryServices` и его подпространствах, в сборках System.DirectoryServices и System.DirectoryServices.Protocols.

Для работы с записями Active Directory используются объекты класса `System.DirectoryServices.DirectoryEntry`. С помощью конструкторов и методов этого класса можно создавать и изменять регистрационные записи службы каталогов. Для поиска зарегистрированных объектов по идентификаторам или свойствам используются методы класса `System.DirectoryServices.DirectorySearcher`.

Active Directory поддерживает обращения к своим записям по нескольким разным протокола, включая протокол LDAP.

Процессы и синхронизация

Помимо процессов и потоков, среда .NET поддерживает так называемые *зоны приложений (application domains)*, которые служат агрегатами ресурсов, как и процессы, но, в отличие от них, управляются с помощью более эффективных механизмов. В рамках одного процесса может быть создано несколько зон приложений. Передача объектов и ресурсов между зонами приложений невозможна без использования специальных механизмов, таких как Remoting. Потоки же в .NET могут пересекать границы зон приложений, если обладают соответствующими правами.

Зоны приложений служат дополнительным элементом защиты .NET-приложений от непреднамеренного взаимного влияния и позволяют сохранить работоспособность процесса при возникновении проблем в одном из его приложений.

Помимо автоматически создаваемых потоков и зон приложений, разработчик может создавать свои собственные потоки и зоны приложений. Вопросы синхронизации потоков и передачи данных между зонами приложений в Web-приложениях могут решаться при помощи стандартных механизмов .NET — конструкций и библиотек синхронизационных примитивов, а также библиотечного класса `System.AppDomain`, чьи методы позволяют выполнять различные операции с зонами приложений.

Целостность

Целостность данных в .NET поддерживается, как и в J2EE, в основном, за счет механизма транзакций. Распределенные транзакции в .NET реализованы на базе *сервера транзакций Microsoft (Microsoft Transaction Server, MTS)*, который появился как часть компонентной среды COM+. Интерфейсы для работы с его функциями собраны в пространстве имен `System.EnterpriseServices`, в сборке с таким же именем.

Автоматические транзакции поддерживаются при помощи указания у классов транзакционных атрибутов, имеющих тип `System.EnterpriseServices.TransactionAttribute`. Такой атрибут может принимать значения `Required`, `RequiresNew`, `NotSupported`, `Supported` и `Disabled`. Первые

три имеют то же значение, что и в J2EE. Атрибут `Supported` действует аналогично `Supports` в J2EE. Атрибут `Disabled` обозначает, что транзакционный контекст вызвавшего метода будет проигнорирован.

Значение атрибутов по умолчанию для методов компонентов ASP.NET и методов обычных классов, используемых в распределенных транзакциях, различны — у первых это `Disabled`, у вторых — `Required`.

Чтобы определить класс, чьи методы могут участвовать в транзакциях, нужно унаследовать его от класса `System.EnterpriseServices.ServicedComponent`, определить транзакционный атрибут, а также прикрепить к сборке, содержащей этот класс, сертификат и зарегистрировать сборку в реестре COM+.

Для завершения или отката автоматической транзакции используются следующие конструкции.

- Атрибут `System.EnterpriseServices.AutoCompleteAttribute` у участвующего в транзакции метода говорит о том, что в случае нормального завершения работы метода и отсутствия проблем у других ее участников транзакция будет завершена успешно. Если же в результате работы метода будет создано исключение, транзакция будет отменена.
- Методы `SetComplete()` и `SetAbort()` класса `System.EnterpriseServices.ContextUtil` могут использоваться для успешного завершения или отката автоматически созданной транзакции.

Создание и управление транзакциями «в ручном режиме» может быть осуществлено для компонентов ADO.NET при помощи методов класса `System.Data.Common.DbConnection` и его наследников и класса `System.Data.Common.DbTransaction`. Для управления транзакциями при передаче сообщений используется класс `System.Messaging.MessageQueueTransaction`.

Отказоустойчивость

Так же, как и для J2EE, отказоустойчивость .NET-приложений должна обеспечиваться либо за счет использования дополнительных продуктов, либо за счет специфического проектирования приложения.

Защита

Защищенность .NET-приложений поддерживается примерно теми же методами, что и защищенность J2EE-приложений.

Здесь также имеется несколько техник аутентификации, возможность определения ролей, обеспеченных набором прав доступа к различным элементам системы, а также возможность использования различных протоколов шифрования и защищенной передачи данных, управления ключами и подтверждения целостности данных. В рамках .NET используются также многоуровневые *политики защиты*, которые определяют набор прав, предоставляемых коду из разных источников.

Программные интерфейсы к различным механизмам управления защищенностью приложений и ресурсов реализуются классами и интерфейсами пространства имен `System.Security`, находящимися в сборках mscorlib, System и System.Security.

Работа с XML

В целом техника работы с XML-документами в .NET опирается на реализацию объектной модели документов XML (DOM) и на механизм разбора, аналогичный StAX, реализуемый классом `System.Xml.XmlReader`. Классы, реализующие различные парсеры XML, различные варианты представления XML-документов, а также их трансформацию на основе XSLT-описаний, находятся в пространстве имен `System.Xml`, разбросанному по сборкам System.Data, System.Data.SqlXml и System.Xml.

Одной из особенностей работы с XML в .NET является встроенная возможность работы с XML-данными в рамках механизмов ADO.NET (в основном предназначенных для работы с реляционными СУБД) с помощью класса `System.Xml.XmlDataDocument`.

Литература к Лекции 13

[1] Web-сайт консорциума World Wide Web http://www.w3.org/.
[2] http://www.xml.com/.
[3] XML 1.1, 2004. Доступен через http://www.w3.org/TR/xml11/.
[4] Annotated XML 1.0, 1998. Доступен через http://www.xml.com/axml/axml.html.
[5] Расширяемый язык разметки (XML) 1.0 (русский перевод первой версии стандарта). Доступен через http://www.rol.ru/news/it/helpdesk/xml01.htm.
[6] Материалы по XMLSchema http://www.w3.org/XML/Schema.
[7] Namespaces in XML, 1999. Доступен через http://www.w3.org/TR/REC-xml-names/.
[8] Java Platform Enterprise Edition Specifications, version 1.4. http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf.
[9] Э. Таненбаум, М. ван Стеен. Распределенные системы. Принципы и парадигмы. СПб.: Питер, 2003.
[10] Документация по Java RMI http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html
[11] Документация по JMS API http://java.sun.com/products/jms/docs.html
[12] Документация по JNDI http://java.sun.com/j2se/1.5.0/docs/guide/jndi/index.html
[13] RFC 1766, доступен по ссылке http://rfc.net/rfc1766.html
[14] http://java.sun.com/products/jndi/serviceproviders.html
[15] Web-страница стандарта SAX http://www.saxproject.org/
[16] Web-страница стандарта DOM http://www.w3.org/DOM/
[17] Web-страница проекта JDOM http://www.jdom.org/
[18] Web-страница проекта dom4j http://www.dom4j.org/
[19] Стандарт XSLT. Доступен через http://www.w3.org/TR/xslt
[20] Web-страница проекта Saxon http://www.saxonica.com/
[21] Web-страница проекта Xalan http://xml.apache.org/xalan-j/
[22] Streaming API for XML http://www.jcp.org/en/jsr/detail?id=173
[23] B. McLaughlin. Java and XML, Second Edition. O'Reilly, 2001.
[24] Документация по платформе J2EE http://java.sun.com/j2ee/1.4/docs/index.html
[25] Документация по платформе .NET — находится в разделе .NET Development MSDN, http://msdn.microsoft.com/library/default.asp
[26] П. Аллен, Дж. Бамбара, М. Ашнаульт, Зияд Дин, Т. Гарбен, Ш. Смит. J2EE. Разработка бизнес-приложений. СПб.: ДиаСофт, 2002.

Лекция 14. Разработка различных уровней Web-приложений в J2EE и .NET

Аннотация

Рассматриваются используемые в рамках Java Enterprise Edition и .NET техники разработки компонентов Web-приложений, связывающих приложение с базой данных и представляющих собой элементы пользовательского интерфейса.

Ключевые слова

Компонент EJB, компонент данных, сеансовый компонент, компонент, управляемый сообщениями, протокол HTTP, сервлет Java, серверная страница Java (JSP), Web-форма .NET.

Текст лекции

Общая архитектура Web-приложений

В данной лекции мы рассмотрим техники разработки компонентов Web-приложений на основе платформ J2EE и .NET. Общая архитектура такого приложения может быть представлена схемой, изображенной на Рис. 74. Обе платформы предоставляют специальную поддержку для разработки компонентов на двух уровнях: уровне интерфейса пользователя (WebUI) и уровне связи с данными.

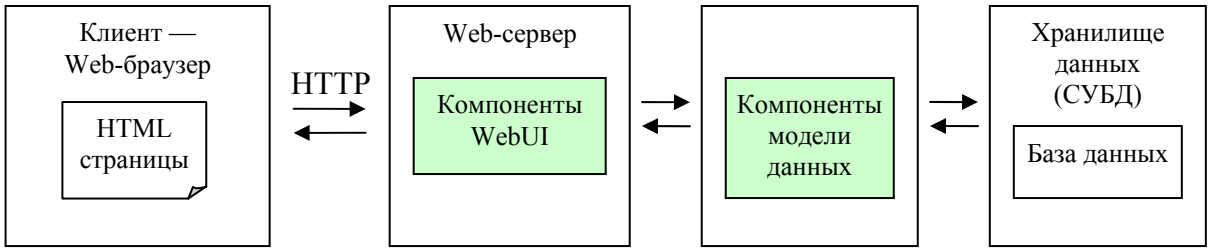


Рисунок 74. Общая схема архитектуры Web-приложений J2EE и .NET.

Пользовательский интерфейс Web-приложений основан на генерации динамических страниц HTML, содержащих данные, которые запрашивает пользователь. Уровень модели данных предоставляет приложению возможность работать с данными, обычно хранящимися в виде набора таблиц и связей между ними, как с набором связанных объектов.

Основные отличия между техниками разработки компонентов этих двух уровней, используемыми в рамках J2EE и .NET, можно сформулировать следующим образом.

- В J2EE компоненты EJB предназначены не только для представления данных приложения в виде объектов, но и для реализации его бизнес-логики, т.е. объектов предметной области и основных способов работы с ними.
  В .NET нет специально выделенного вида компонентов, предназначенного для реализации бизнес-логики — она может реализовываться с помощью обычных классов, что часто удобнее. Это положение должно измениться с выходом EJB 3.0.
- EJB компоненты являются согласованным с объектно-ориентированным подходом представлением данных приложения. Работа с ними организуется так же, как с объектами обычных классов (с точностью до некоторых деталей).
  В .NET-приложениях все предлагаемые способы представления данных являются объектными обертками вокруг реляционного представления — в любом случае приходится работать с данными как с набором таблиц. В .NET нет автоматической поддержки их преобразования в систему взаимосвязанных объектов и обратно.

Уровень бизнес-логики и модели данных в J2EE

В рамках приложений, построенных по технологии J2EE связь с базой данных и бизнес-логику, скрытую от пользователя, принято реализовывать с помощью компонентов Enterprise JavaBeans. На момент написания этой лекции последней версией технологии EJB является версия 2.1, в первой половине 2006 года должны появиться инструменты для работы с EJB 3.0 (в рамках J2EE 5.0).

Возможны и другие способы реализации этих функций. Например, бизнес-логика может быть реализована непосредственно в методах объектов пользовательского интерфейса, а обмен данными с базой данных — через интерфейс JDBC. При этом, однако, теряется возможность переиспользования функций бизнес-логики в разных приложениях на основе единой базы данных, а также становится невозможным использование автоматических транзакций при работе с данными. Транзакции в этом случае нужно организовывать с помощью явных обращений к JTA (см. предыдущую лекцию).

Компонент Enterprise JavaBeans (EJB) является компонентом, представляющим в J2EE-приложении элемент данных или внутренней, невидимой для пользователя логики приложения. Для компонентов EJB определен жизненный цикл в рамках рабочего процесса приложения — набор состояний, через которые проходит один экземпляр такого компонента. Компоненты EJB работают внутри EJB-контейнера, являющегося для них компонентной средой. Функции EJB-контейнера следующие.

- Управление набором имеющихся EJB-компонентов, например, поддержкой пула компонентов для обеспечения большей производительности, а также жизненным циклом каждого отдельного компонента, в частности, его инициализацией и уничтожением.
- Передача вызовов между EJB-компонентами, а также их удаленных вызовов. Несколько EJB-контейнеров, работающих на разных машинах, обеспечивают взаимодействие наборов компонентов, управляемых каждым из них.
- Поддержка параллельной обработки запросов.
- Поддержка связи между EJB-компонентами и базой данных приложения и синхронизация их данных.
- Поддержка целостности данных приложения с помощью механизма транзакций.
- Защита приложения с помощью механизма ролей: передача прав ролей при вызовах между компонентами и проверка допустимости обращения в рамках роли к методам компонентов.

Для разработки набора компонентов EJB нужно, во-первых, для каждого компонента создать один или несколько классов и интерфейсов Java, обеспечивающих реализацию самой функциональности компонента и определение интерфейсов для удаленных обращений к нему, и, во-вторых, написать дескриптор развертывания — XML-файл, описывающий следующее.

- Набор EJB-компонентов приложения.
- Совокупность элементов кода на Java, образующих один компонент.
- Связь свойств компонента с полями таблиц БД и связями между таблицами.
- Набор ролей, правила доступа различных ролей к методам компонентов, правила передачи ролей при вызовах одними компонентами других.
- Политику компонентов и их методов по отношению к транзакциям.
- Набор ресурсов, которыми компоненты могут пользоваться в своей работе.

Правила создания EJB компонента зависят от его вида. Различают три таких вида EJB-компонентов.

- ***Компоненты данных (сущностные, entity beans)***.
  Представляют данные приложения и основные методы работы с ними.
- ***Сеансовые компоненты (session beans)***.
  Представляют независимую от пользовательского интерфейса и конкретных типов данных логику работы приложения, называемую иногда бизнес-логикой.

- ***Компоненты, управляемые сообщениями (message driven beans)***.
  Тоже предназначены тоже для реализации бизнес-логики. Но, если сеансовые компоненты предоставляют интерфейс для синхронных вызовов, компоненты, управляемые сообщениями, предоставляют асинхронный интерфейс. Клиент, вызывающий метод в сеансовом компоненте, ждет, пока вызванный компонент не завершит свою работу. Компоненту же, управляемому сообщениями, можно отослать сообщение и продолжать работу сразу после окончания его передачи, не дожидаясь окончания его обработки.

Далее описываются основные правила построения EJB компонентов разных видов. Более детальное описание этих правил можно найти в [1,2].

Компоненты данных и сеансовые компоненты

Компонент данных или сеансовый компонент могут состоять из следующих элементов: пара интерфейсов для работы с самим компонентом — *удаленный интерфейс* и *локальный интерфейс*; пара интерфейсов для поиска и создания компонентов — *удаленный исходный интерфейс* и *локальный исходный интерфейс*; *класс компонента*, реализующий методы работы с ним; и, для компонентов данных, — *класс первичного ключа*. Обязательно должен быть декларирован класс компонента и один из интерфейсов — удаленный или локальный. Для компонентов данных обязательно должен быть определен класс первичного ключа.

- *Удаленный интерфейс (remote interface)*.
  Этот интерфейс декларирует методы компонента, к которым можно обращаться удаленно, т.е. из компонентов, работающих в рамках другого процесса или на другой машине. Удаленный интерфейс должен наследовать интерфейс `javax.ejb.EJBObject` (в свою очередь, наследующий `java.rmi.Remote`).
  Для компонента данных он определяет набор свойств (в смысле JavaBeans, т.е. пар методов `Type getName()`/`void setName(Type)`), служащих для работы с отдельными полями данных или компонентами, связанными с этим компонентом по ссылкам. Это могут быть и часто используемые дополнительные операции, как-то выражающиеся через операции с отдельными полями данных, в том числе и вычислимые свойства. Например, для книги в базе данных приложения хранится набор ссылок на данные об ее авторах, а число авторов может быть часто используемым свойством книги, вычисляемым по этому набору ссылок.
  Для сеансового компонента методы удаленного интерфейса служат для реализации некоторых операций бизнес-логики или предметной области, вовлекающих несколько компонентов данных.
- *Локальный интерфейс (local interface)*.
  По назначению этот интерфейс полностью аналогичен удаленному, но декларирует методы компонента, которые можно вызывать только в рамках того же процесса. Этот интерфейс служит для увеличения производительности приложений, в которых взаимодействия между компонентами происходят в основном в рамках одного процесса. При этом они могут использовать локальные интерфейсы друг друга, не привлекая сложный механизм реализации удаленных вызовов методов. Однако, при использовании локального интерфейс компонента нужно обеспечить развертывание этого компонента в рамках того же EJB-контейнера, что и вызывающий его компонент.
  Локальный интерфейс должен наследовать интерфейсу `javax.ejb.EJBLocalObject`.
- *Удаленный исходный интерфейс (remote home interface).*
  Исходные интерфейсы служат для поиска и создания компонентов. Такой интерфейс может декларировать метод поиска компонента данных по значению его первичного ключа `findByPrimaryKey(…)` и метод создания такого компонента с указанным значением первичного ключа `create(…)`. Могут быть также определены методы, создающие компонент по набору его данных или возвращающие коллекцию компонентов, данные которых соответствуют аргументам такого метода. Например, метод, создающий компонент, который представляет книгу с данным названием, `createByTitle(String title)`, или метод, находящий все книги с данным набором авторов `Collection`

`findByAuthors(Collection authors)`.
Удаленный исходный интерфейс предназначен для создания и поиска компонентов извне того процесса, в котором они работают. Его методы возвращают ссылку на удаленный интерфейс компонента или коллекцию таких ссылок.
Такой интерфейс должен наследовать интерфейсу `javax.ejb.EJBHome` (являющемуся наследником `java.rmi.Remote`).

- *Локальный исходный интерфейс (local home interface)*.
Имеет то же общее назначение, что и удаленный исходный интерфейс, но служит для работы с компонентами в рамках одного процесса. Соответственно, его методы поиска или создания возвращают ссылку на локальный интерфейс компонента или коллекцию таких ссылок.
Должен наследовать интерфейсу `javax.ejb.EJBLocalHome`.
- *Класс компонента (bean class)*.
Этот класс реализует методы удаленного и локального интерфейсов (но не должен реализовывать сами эти интерфейсы!). Он определяет основную функциональность компонента.
Для компонентов данных такой класс должен быть абстрактным классом, реализующим интерфейс `javax.ejb.EntityBean`. Свойства, соответствующие полям хранимых данных или ссылкам на другие компоненты данных, должны быть определены в виде абстрактных пар методов `getName()`/`setName()`. В этом случае EJB-контейнер может взять на себя управление синхронизацией их значений с базой данных. Вычислимые свойства, значения которых не хранятся в базе данных, реализуются в виде пар неабстрактных методов.
Для сеансовых компонентов класс компонента должен быть неабстрактным классом, реализующим интерфейс `javax.ejb.SessionBean` и все методы удаленного и локального интерфейсов.
Кроме того, класс компонента может (а иногда и должен) реализовывать некоторые методы, которые вызываются EJB-контейнером при переходе между различными этапами жизненного цикла компонента.

  Например, при инициализации экземпляра компонента всегда вызывается метод `ejbCreate()`. Для компонентов данных он принимает на вход и возвращает значение типа первичного ключа компонента. Если первичный ключ — составной, он должен принимать на вход значения отдельных его элементов. Такой метод для компонента данных должен возвращать **`null`** и всегда должен быть реализован в классе компонента. Для сеансовых компонентов он имеет тип результата `void`, а на вход принимает какие-то параметры, служащие для инициализации экземпляра компонента. Для каждого метода исходных интерфейсов с именем `createSomeSuffix(…)` в классе компонента должен быть реализован метод `ejbCreateSomeSuffix(…)` с теми же типами параметров. Для компонентов данных все такие методы возвращают значение типа первичного ключа, для сеансовых — **`void`**.

  Другие методы жизненного цикла компонента, которые можно перегружать в классе компонента, декларированы в базовых интерфейсах компонентов соответствующего вида (`javax.ejb.EntityBean` или `javax.ejb.SessionBean`). Это, например, `ejbActivate()` и `ejbPassivate()`, вызываемые при активизации и деактивизации экземпляра компонента; `ejbRemove()`, вызываемый перед удалением экземпляра компонента из памяти; для компонентов данных — `ejbStore()` и `ejbLoad()`, вызываемые при сохранении данных экземпляра в базу приложения или при их загрузке оттуда.
Схема жизненного цикла компонента данных показана на Рис. 75.

  Сеансовые компоненты могут поддерживать состояние сеанса, обеспечивая пользователю возможность получения результатов очередного запроса с учетом предшествовавших запросов в рамках данного сеанса, или не поддерживать. Во втором случае компонент реализует обработку запросов в виде чистых функций.
Жизненный цикл сеансового компонента различается в зависимости от того, поддерживает ли компонент состояние сеанса или нет.

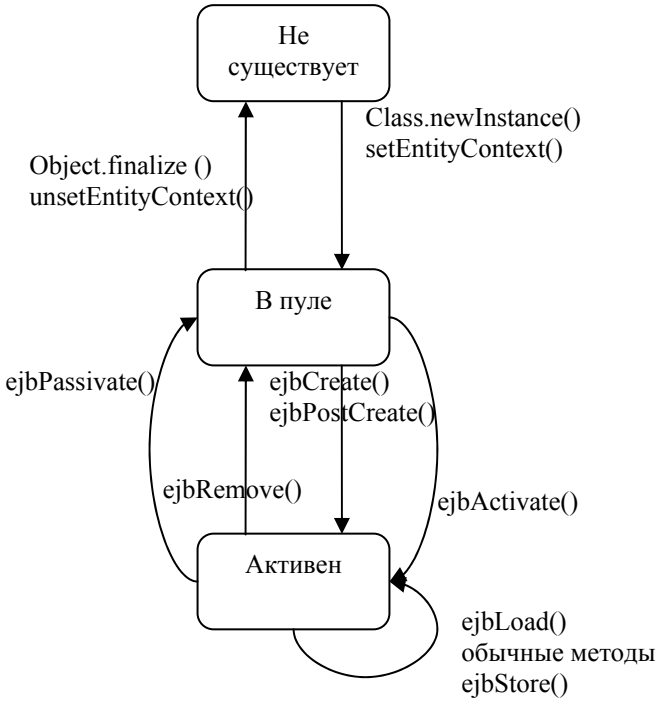


Рисунок 75. Жизненный цикл EJB компонента данных.

Схема жизненного цикла сеансового компонента с состоянием показана на Рис. 76. Отличие от жизненного цикла компонента данных единственное — метод `ejbCreate()` сразу переводит компонент в активное состояние.

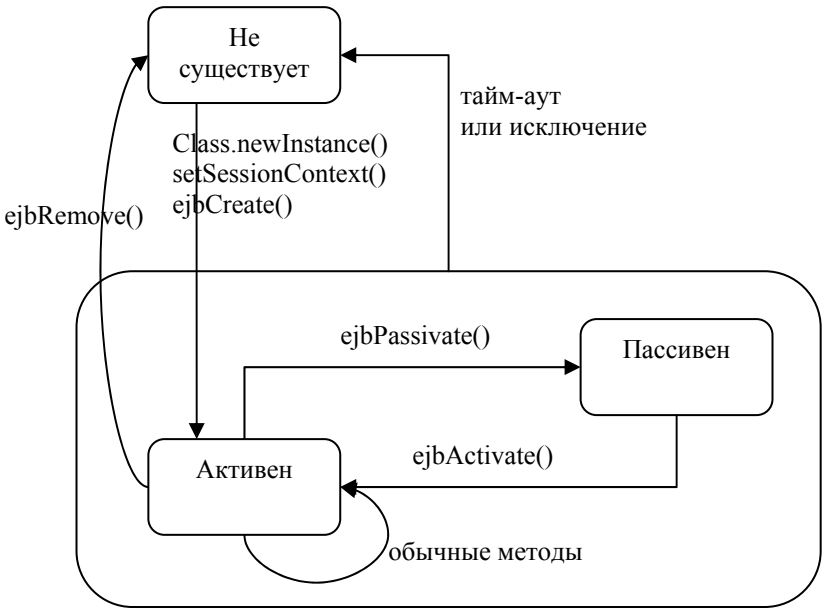


Рисунок 76. Жизненный цикл сеансового компонента с состоянием.

Жизненный цикл сеансового компонента без состояния гораздо проще. Его схема представлена на Рис. 77.

- *Класс первичного ключа (primary key class).*
  Декларируется только для компонентов данных, если в этом качестве нельзя использовать подходящий библиотечный класс.
  Определяет набор данных, которые образуют первичный ключ записи базы данных, соответствующей одному экземпляру компонента.
  Чаще всего это библиотечный класс, например, `String` или `Integer`. Пользовательский класс необходим, если первичный ключ составной, т.е. состоит из нескольких значений

простых типов данных. В таком классе должен быть определен конструктор без параметров и правильно перегружены методы `equals()` и `hashCode()`, чтобы EJB-контейнер мог корректно управлять коллекциями экземпляров компонентов с такими первичными ключами. Такой класс также должен реализовывать интерфейс `java.io.Serializable`.

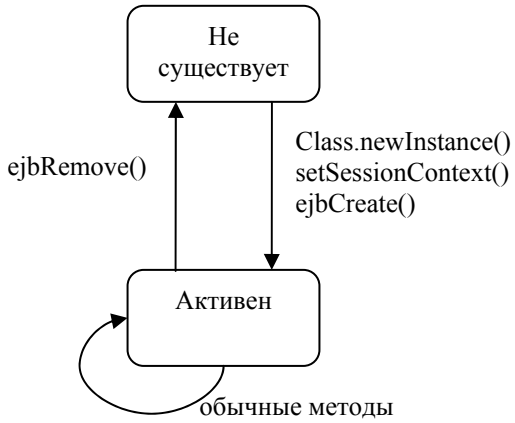


Рисунок 77. Жизненный цикл сеансового компонента без состояния.

Ниже приведены примеры декларации класса компонента и интерфейсов для компонентов данных, соответствующих простой схеме из двух таблиц, которая изображена на Рис. 78.

В рамках этой схемы, состоящей из таблиц, где хранятся данные книг и организаций-издателей, каждая книга связана с одним и только одним издателем, а каждый издатель может иметь ссылки на некоторое множество книг (возможно, пустое). Каждая таблица имеет поле `ID`, являющееся первичным ключом. Таблица `Book` имеет поле `PublisherID`, содержащее значение ключа записи об издателе данной книги.

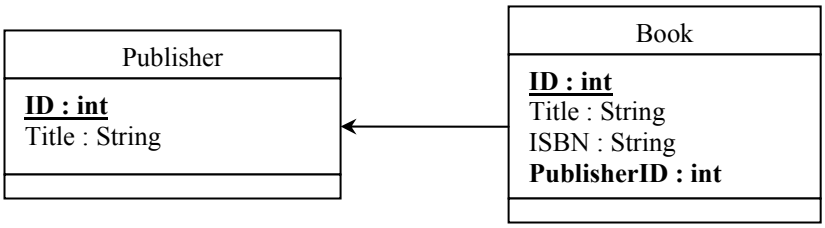


Рисунок 78. Пример схемы БД.

Примеры кода удаленных интерфейсов для компонентов, представляющих данные о книгах и издателях в рамках EJB-приложения.

```
package ru.msu.cmc.prtech.examples;

import java.rmi.RemoteException;
import java.util.Collection;
import javax.ejb.EJBObject;

public interface PublisherRemote extends EJBObject
{
  public String getTitle ()             throws RemoteException;
  public void   setTitle (String title) throws RemoteException;

  public Collection getBooks ()throws RemoteException;
  public void       setBooks (Collection books)
    throws RemoteException;

  public void addBook    (String title, String isbn)
    throws RemoteException;
  public void removeBook (String title, String isbn)
    throws RemoteException;
}
```

```
package ru.msu.cmc.prtech.examples;

import java.rmi.RemoteException;

import javax.ejb.EJBObject;

public interface BookRemote extends EJBObject
{
  public String getTitle ()             throws RemoteException;
  public void   setTitle (String title) throws RemoteException;

  public String getISBN ()            throws RemoteException;
  public void   setISBN (String isbn) throws RemoteException;

  public PublisherRemote getPublisher () throws RemoteException;
  public void            setPublisher (PublisherRemote publisher)
    throws RemoteException;
}
```

Примеры кода удаленных исходных интерфейсов.

```
package ru.msu.cmc.prtech.examples;

import java.rmi.RemoteException;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;

public interface PublisherHomeRemote extends EJBHome
{
  public PublisherRemote create (Integer id)
    throws CreateException, RemoteException;

  public PublisherRemote findByPK (Integer id)
    throws FinderException, RemoteException;
}
```

```
package ru.msu.cmc.prtech.examples;

import java.rmi.RemoteException;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;

public interface BookHomeRemote extends EJBHome
{
  public BookRemote create (Integer id)
    throws CreateException, RemoteException;

  public BookRemote createBook (String title, String isbn)
    throws CreateException, RemoteException;

  public BookRemote findByPK (Integer id)
    throws FinderException, RemoteException;
}
```

Примеры кода классов компонентов. Показано, как реализовывать дополнительные, не поддерживаемые контейнером автоматически, методы работы со связями между данными об издателях и книгах и дополнительные методы создания компонентов.

```
package ru.msu.cmc.prtech.examples;

import java.util.Collection;
import java.util.Iterator;
```

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public abstract class PublisherBean implements EntityBean
{
  public Integer ejbCreate (Integer pk)
  {
    setId(pk);
    return null;
  }

  public void ejbPostCreate (Integer pk) { }

  public abstract Integer getId ();
  public abstract void    setId (Integer pk);

  public abstract String getTitle ();
  public abstract void   setTitle (String title);

  public abstract Collection getBooks ();
  public abstract void       setBooks (Collection books);

  public void addBook (String title, String isbn)
  {
    try
    {
      InitialContext context = new InitialContext();
      BookHomeRemote bookHome =
         (BookHomeRemote)context.lookup("BookHomeRemote");
      BookRemote book = bookHome.createBook(title, isbn);

      Collection books = getBooks();
      books.add(book);
    }
    catch (NamingException e) { e.printStackTrace(); }
    catch (CreateException e) { e.printStackTrace(); }
    catch (RemoteException e) { e.printStackTrace(); }
  }

  public void removeBook (String title, String isbn)
  {
    Collection books = getBooks();
    Iterator it = books.iterator();

    try
    {
      while(it.hasNext())
      {
        BookRemote book = (BookRemote)it.next();
        if(   book.getTitle().equals(title)
           && book.getISBN().equals(isbn)
          )
        {
          it.remove();
          break;
        }
      }
    }
    catch (RemoteException e) { e.printStackTrace(); }
  }
}
```

```
package ru.msu.cmc.prtech.examples;

import javax.ejb.EntityBean;

public abstract class BookBean implements EntityBean
{
  public Integer ejbCreate (Integer pk)
  {
    setId(pk);
    return null;
  }

  public void ejbPostCreate (Integer pk) { }

  public Integer ejbCreateBook (String title, String isbn)
  {
    setTitle(title);
    setISBN(isbn);
    return null;
  }

  public void ejbPostCreateBook (String title, String isbn) { }

  public abstract Integer getId ();
  public abstract void    setId (Integer pk);

  public abstract String getTitle ();
  public abstract void   setTitle (String title);

  public abstract String getISBN ();
  public abstract void   setISBN (String isbn);

  public abstract PublisherRemote getOrganization ();
  public abstract void            setOrganization (PublisherRemote pr);
}
```

Компоненты, управляемые сообщениями

Компоненты, управляемые сообщениями, не доступны для удаленных вызовов, и поэтому не имеют удаленных и исходных интерфейсов. Для создания такого компонента нужно определить только его класс. Обращения к компоненту организуются в виде посылки сообщений к объекту этого класса как к реализующему интерфейс `javax.jms.MessageListener`. Вместе с этим интерфейсом класс компонента EJB, управляемого сообщениями, обязан реализовывать интерфейс `javax.ejb.MessageDrivenBean`.

Первый интерфейс требует определения метода **`void`** `onMessage(javax.jms.Message)`, который разбирает содержимое пришедшего сообщения и определяет способ его обработки. Кроме того, нужно определить методы **`void`** `ejbCreate()` для создания компонента и **`void`** `ejbRemove()` для освобождения ресурсов при его удалении.

Жизненный цикл компонента, управляемого сообщениями выглядит в целом так же, как и жизненный цикл сеансового компонента без состояния (Рис. 77). Вся необходимая информация передается такому компоненту в виде данных обрабатываемого им сообщения.

Пример реализации класса компонента, управляемого сообщениями, приведен ниже. Данный компонент получает идентификатор издателя, название и ISBN книги и добавляет такую книгу к книгам, изданным данным издателем.

```
package ru.msu.cmc.prtech.examples;

import javax.ejb.EJBException;
import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.jms.MapMessage;
import javax.jms.Message;
```

```
import javax.jms.MessageListener;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class TransferProcessorBean
  implements MessageDrivenBean, MessageListener
{
  Context context;

  public void setMessageDrivenContext (MessageDrivenContext mdc)
      throws EJBException
  {
    try { context = new InitialContext(); }
    catch (NamingException e) { throw new EJBException(e); }
  }

  public void ejbCreate() { }

  public void onMessage (Message msg)
  {
    MapMessage message = (MapMessage)msg;
    try
    {
      Integer publisherPK = (Integer)message.getObject("Publisher");
      String  title       = (String)message.getObject("Title");
      String  isbn        = (String)message.getObject("ISBN");

      PublisherHomeRemote publisherHome =
        (PublisherHomeRemote)context.lookup("PublisherHomeRemote ");

      PublisherRemote publisher = publisherHome.findByPK(publisherPK);

      publisher.addBook(title, isbn);
    }
    catch (Exception e) { throw new EJBException(e); }
  }

  public void ejbRemove () throws EJBException
  {
    try { context.close(); }
    catch (NamingException e) { }
  }
}
```

Дескрипторы развертывания компонентов EJB

Помимо декларации интерфейсов и классов компонентов, для построения EJB компонента необходимо написать дескриптор развертывания — XML файл в специальном формате, определяющий набор компонентов приложения и их основные свойства.

Чаще всего дескрипторы развертывания не пишут вручную, их готовят с помощью специализированных инструментов для развертывания J2EE приложений или сред разработки (например, такими возможностями обладает среда NetBeans 4.0 [3]). Здесь мы опишем только часть содержимого дескрипторов развертывания. Полное описание используемых в них тегов и их назначения см. в [2].

Дескриптор развертывания упаковывается вместе с байт-кодом классов компонентов приложения в JAR-архив. При развертывании такой архив помещают в выделенную директорию, в которой сервером J2EE ищет развертываемые приложения. После этого сервер сам осуществляет запуск приложения и поиск кода компонентов, необходимых для обработки поступающих запросов, на основании информации, предоставленной дескриптором.

Заголовок дескриптора развертывания для набора EJB компонентов версии 2.1 выглядит следующим образом.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN" http://java.sun.com/j2ee/dtds/ejb-jar_2_1.dtd">
```

Дескриптор развертывания содержит следующие теги.

- `<ejb-jar>`
  Обязательный элемент.
  Это корневой тег дескриптора развертывания набора EJB компонентов, содержащий все остальные теги.
- `<enterprise-beans>`
  Обязательный элемент, должен появиться внутри `<ejb-jar>` ровно один раз.
  Содержит набор описаний отдельных EJB компонентов в виде элементов `<entity>`, `<session>`, `<message-driven>`.
- `<entity>` и `<session>`
  Эти теги вложены в тег `<enterprise-beans>` и служат для описания, соответственно, компонентов данных и сеансовых компонентов. Они могут содержать следующие вложенные теги.
  - `<ejb-name>`
    Требуется ровно один.
    Задает имя компонента.
  - `<home>`
    Необязателен, начиная с EJB 2.0. В EJB 1.1 требуется ровно один.
    Указывает полное имя удаленного исходного интерфейса.
  - `<remote>`
    Необязателен, начиная с EJB 2.0. В EJB 1.1 требуется ровно один.
    Указывает полное имя удаленного интерфейса.
  - `<local-home>`
    Необязателен.
    Указывает полное имя локального исходного интерфейса.
  - `<local>`
    Необязателен.
    Указывает полное имя локального интерфейса.
  - `<ejb-class>`
    Требуется ровно один.
    Указывает полное имя класса компонента.
  - `<primkey-field>`
    Необязателен, используется в описании компонентов данных.
    Указывает имя поля, являющегося первичным ключом (если он состоит только из одного поля и синхронизацией компонента с базой данных управляет контейнер).
  - `<prim-key-class>`
    Требуется ровно один, используется в описании компонентов данных.
    Указывает имя класса первичного ключа. Можно отложить точное определение класса первичного ключа до развертывания, тогда в этом поле указывается `java.lang.Object`.
  - `<persistence-type>`
    Требуется ровно один, используется в описании компонентов данных.
    Имеет значения `Bean` или `Container`, в зависимости от того, управляется ли синхронизация компонента с базой данных самим компонентом или контейнером.
  - `<cmp-version>`
    Необязателен.
    Указывает версию спецификаций EJB, в соответствии с которой разработан компонент, что определяет способ управления этим компонентом. Может иметь значения 2.x и 1.x.

- `<abstract-schema-name>`
  Необязателен.
  Задает уникальный идентификатор компонента для использования в запросах на языке EJB QL, который используется для описания запросов к схеме данных при реализации компонента данных, самостоятельно управляющего связью с СУБД.
- `<cmp-field>`
  Один или более, используется в описании компонентов данных.
  Каждый такой элемент описывает одно поле данных, синхронизация которого с СУБД управляется EJB контейнером. Он может содержать тег `<description>` с описанием поля и должен содержать тег `<field-name>` с именем поля. В EJB 2.0 это имя совпадает с именем абстрактного свойства (для которого в классе компонента декларированы методы `getName()` и `setName()`), а в EJB 1.1 — с именем одного из полей класса компонента.
- `<security-role-ref>`
  Один или более, необязателен.
  Указывает роли, используемые данным компонентом. Они при работе приложения служат для авторизации доступа — сопоставляются с ролями, которым разрешен доступ к тому или иному методу.
  Может содержать тег `<description>` (необязателен) и теги `<role-name>` (обязателен), `<role-link>` (необязателен, служит для сопоставления указанного имени роли с логической ролью, описанной в `<security-role>` раздела `<assembly-descriptor>`).
- `<security-identity>`
  Необязателен.
  Определяет, какую логическую роль будет играть данный компонент при обращениях к другим компонентам. Для этого может быть использован вложенный тег `<run-as><role-name>...</role-name></run-as>` для указания имени конкретной логической роли, или `<use-caller-identity/>` для указания того, что нужно использовать роль вызывающего клиента.
- `<session-type>`
  Требуется ровно один, используется в описании сеансовых компонентов.
  Имеет значения `Stateful` или `Stateless`, в зависимости от того, использует ли данный компонент состояние сеанса.
- `<transaction-type>`
  Требуется ровно один, используется в описании сеансовых компонентов.
  Имеет значения `Container` или `Bean`, в зависимости от того, управляет ли транзакциями данного компонента контейнер или он сам. В первом случае соответствующие транзакции должны быть описаны в разделе `<assembly-descriptor>`, см. далее.
- `<query>`
  Один или более, необязателен.
  Используется для описания запросов, с чьей помощью реализуются некоторые методы компонентов данных, которые сами управляют связью с базой данных. Запросы описываются на языке EJB QL [X] и привязываются к методам компонента с помощью тегов `<query-method>`. Сам код запроса описывается внутри элемента CDATA во вложенном теге `<ejb-ql>`.
  Например

```
<query>
  <query-method>
    <method-name>findByName</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    <!CDATA[
      SELECT OBJECT(c) FROM Client c WHERE c.name = ?1
```

```
        ]]
      </ejb-ql>
    </query>
```

- `<relationships>`
  Необязателен. Вложен в `<ejb-jar>`.
  Описывает набор отношений между компонентами, которые соответствуют связям в схеме базы данных и автоматически поддерживаются контейнером. Каждое отношение описывается с помощью вложенного тега `<ejb-relation>`.
  - `<ejb-relation>`
    Описывает одно отношение и может иметь следующие элементы.
    - `<ejb-relation-name>`
      Необязателен, только один. Задает имя отношения.
    - `<ejb-relationship-role>`
      Обязательно два. Описывает одну роль в рамах отношения. В описании роли должны присутствовать следующие данные.
      - Имя роли — во вложенном теге `<ejb-relationship-role-name>`.
      - Множественность — сколько экземпляров компонента могут играть такую роль в раках данного отношения с одним экземпляром в другой роли. Описывается в теге `<multiplicity>` и может иметь значения `One` или `Many`.
      - Имя компонента, экземпляры которого играют данную роль в этом отношении. Определяется в теге `<relationship-role-source>` внутри тега `<ejb-name>` в виде имени, которое присвоено компоненту в рамках данного дескриптора.
      - Имя поля, которое хранит ссылку или коллекцию ссылок, поддерживающие это отношение в рамках экземпляра компонента. Определяется в теге `<cmr-field>`, во вложенном теге `<cmr-field-name>`, и для него в классе компонента должно быть определено абстрактное свойство с тем же именем.

- `<assembly-descriptor>`
  Этот обязательный тег внутри `<ejb-jar>` содержит дополнительные указания для сборки компонентов, в частности следующие.
  - `<container-transaction>`
    Один или более, необязателен.
    Содержит необязательный элемент `<description>`, а также приведенные ниже.
    Для компонента данных должно быть по одному такому элементу на каждый метод удаленного интерфейсов. Сеансовые компоненты, транзакциями которых управляет EJB-контейнер, также должны подчиняться этому правилу.
    - `<method>`
      Один или более.
      Содержит тег `<ejb-name>`, указывающий имя компонента, и `<method-name>`, указывающий имя метода или знак *, который обозначает применение указанного атрибута ко всем методам.
      Может также включать элементы `<description>`, `<method-params>` и `<method-intf>`, который может иметь значения `Remote`, `Home`, `Local`, `Local-Home`, в зависимости от того, в каком интерфейсе этот метод декларирован — для поддержки возможности декларировать методы с одним именем и набором параметром в разных интерфейсах.
    - `<trans-attribute>`
      Ровно один.
      Определяет атрибут транзакции, управляющий политикой включения в транзакции или создания новых транзакций.
      Для компонентов данных атрибуты транзакции должны быть определены для всех методов удаленного интерфейса и методов, декларированных в исходном

интерфейсе, для сеансовых компонентов — для всех методов удаленного интерфейса.
Может иметь значения `NotSupported`, `Supports`, `Required`, `RequiresNew`, `Mandatory`, `Never`. Об их смысле рассказывалось в предыдущей лекции.

- `<security-role>`
  Один или более, необязателен.
  Определяет роли, служащие для контроля доступа к методам компонентов. В таком элементе должен содержаться тег `<role-name>`, задающий имя роли.
- `<method-permission>`
  Один или более, необязателен.
  Указывает правила доступа ролей, определенных в тегах `<security-role>`, к методам компонентов.
  Содержит необязательный тег `<description>`, один или несколько тегов `<role-name>` и один или несколько тегов `<method>` (см. выше), кроме того, в нем может присутствовать тег `<unchecked/>`, который обозначает отсутствие проверки прав доступа во время работы, даже если они описаны.
  Каждый тег `<method>` содержит тег `<ejb-name>`, указывающий имя компонента, и `<method-name>`, указывающий имя метода или знак *, который обозначает применение указанного атрибута ко всем методам.
- `<exclude-list>`
  Необязателен.
  Содержит один или несколько тегов `<method>` (см. выше), определяющих методы, которые не должны вызываться при работе приложения. Каждый вызов такого метода создает исключительную ситуацию.

Уровень модели данных в .NET

В среде .NET нет средств, полностью аналогичных тем, которые предоставляются в J2EE для разработки компонентов EJB. Уровень бизнес-логики в .NET-приложениях предлагается реализовывать с помощью обычных классов, что значительно проще, чем реализовывать специальные наборы классов и интерфейсов для компонентов EJB. С другой стороны, связь с базой данных в .NET не реализуется в виде аналогичного EJB объектного интерфейса, если, конечно, не разрабатывать его целиком самостоятельно (или с использованием библиотек компонентов от третьих партий). Вместо этого предлагается для связи с базой данных использовать набор компонентов ADO.NET [4], представляющих собой объектные обертки реляционной структуры данных.

Классы ADO.NET располагаются в сборке System.Data (дополнительные классы можно найти в System.Data.OracleClient и System.Data.SqlXml) и пространстве имен `System.Data`, вместе с вложенными в него пространствами.

Основным классом, с помощью которого представляются данные из базы данных, является `System.Data.DataSet`. Он представляет набор таблиц, связанных между собой некоторыми связями и выполняющими определенные ограничения. Каждая таблица представляется объектом класса `System.Data.DataTable`, каждая связь — объектом класса `System.Data.Relation`, каждое ограничение — объектом класса `System.Data.Constraint`. Структура таблиц описывается с помощью их полей (представляемых объектами `System.Data.DataColumn`). Содержимое одной таблицы представлено как набор объектов-записей, имеющих тип `System.Data.DataRow`.

Из перечисленных классов только `DataSet` и `DataTable` являются сериализуемыми, т.е. только их объекты могут быть переданы в другой процесс или на другую машину.

Объект класса `DataSet` может представлять собой и набор данных документа XML. Получить такой объект можно с помощью класса `System.Xml.XmlDataDocument`.

Само взаимодействие с источником данных происходит с помощью объектов классов `DataAdapter`, `DataReader`, `DbConnection`, `DbTransaction`, `DbCommand` и производных от них, специфичных для того или иного вида источников данных (в рамках поставляемых в составе

среды Visual Studio .NET библиотек имеются специфичные классы для работы с источниками ODBC, OleDB, MS SQL Server, Oracle). Все перечисленные классы находятся в пространстве имен `System.Data.Common`, а их производные для данного вида источников данных — в соответствующем этому виду источников подпространстве `System.Data`.

Объекты классов `DataAdapter` и `DataReader` служат для чтения и записи данных в виде объектов `DataSet`. Остальные классы используются для определения соединений, организации транзакций, определения и выполнения SQL-команд по чтению или изменению данных.

Ниже приводится простой пример работы с данными с помощью библиотек ADO.NET.

```
DbConnection connection = new SqlConnection("Data Source=localhost;" +
    "Integrated Security=SSPI;Initial Catalog=DBCatalog");

DbCommand command = new SqlCommand("SELECT ID, Title, ISBN FROM Book",
    connection);

DataAdapter adapter = new SqlDataAdapter();
Adapter.SelectCommand = command;

connection.Open();

DataSet dataset = new DataSet();
adapter.Fill(dataset, "Book");

connection.Close();
```

Протокол HTTP

Прежде, чем перейти к построению интерфейса пользователя в Web-приложениях на основе J2EE и .NET, стоит рассмотреть основные элементы протокола HTTP, используемого для передачи данных между серверами и клиентами в таких приложениях. Поскольку основная функциональность компонентов интерфейса Web-приложений связана с обработкой и созданием сообщений HTTP, знание элементов этого протокола необходимо для понимания технологий разработки приложений такого рода.

HTTP (Hypertext Transfer Protocol, протокол передачи гипертекста) представляет собой протокол прикладного уровня, использующий для пересылки данных протокол транспортного уровня. Достаточно подробное описание его можно найти в [5,6].

Сообщения HTTP бывают двух видов: ***запросы клиента*** и ***ответы сервера***.

Запрос HTTP состоит из идентификации метода запроса, универсального идентификатора запрашиваемого ресурса (Universal Resource Identifier, URI), указания версии протокола и, возможно, набора заголовков с дополнительной информацией, а также поля данных общего вида.

```
<Request> ::= GET <URI> CrLf
            | <Method> <URI> <HTTP-Version> CrLf
              <Req-Header>* ( <Data> )?
<Req-Header>  ::= <Field> : <Value> CrLf
CrLf      ::= '\r''\n'
```

Основные методы протокола HTTP следующие.

- `GET`
  Служит для получения любой информации по URI запроса, обычно — документа, хранящегося по указанному адресу или генерируемого по запросу с таким URI. Может иметь заголовок `If-Modified-Since`, который предписывает не посылать тело запрашиваемого ресурса, если он не изменялся с указанной даты.
- `POST`
  Служит для создания нового ресурса, связанного с указанным по URI. Чаще всего используется для аннотации ресурсов, добавления сообщений в группы новостей, дистанционной работы с базами данных. Реальная обработка такого запроса зависит от содержащегося в нем URI.

Остальные методы — `HEAD`, `PUT`, `DELETE`, `LINK`, `UNLINK` — используются гораздо реже.

Заголовки запроса служат для передачи дополнительной информации об этом запросе или о клиенте. Заголовок состоит из идентификатора поля и его значения, разделенных двоеточием, и бывает одного из следующих типов.

- `From`
  Содержит e-mail адрес пользователя, под чьим именем работает клиент.
  Пример: `From: webmaster@yandex.ru`
- `Accept`, `Accept-Encoding`, `Accept-Charset` и `Accept-Language`
  В таком заголовке через запятую перечисляются возможные форматы (соответственно, кодировки, используемые таблицы символов и языки) ответов на данный запрос.
  Пример: `Accept: text/plain, text/html, text/x-dvi; q=.8; mxb=100000; mxt=5.0`
- `User-Agent`
  Содержит название используемой клиентской программы.
- `Referer`
  Используется для указания адреса ресурса, с которого был получен данный запрос.
- `If-Modified-Since`
  Используется для отмены ответной пересылки документов, модифицированных не позднее указанной даты, с целью снижения нагрузки на сеть.
- `Authorization`
  Содержит авторизационную информацию, специфичную для используемых сервером протоколов авторизации.
- `ChargeTo`
  Содержит информацию о том, куда выставить счет за обработку запроса.
- `Pragma`
  Содержит дополнительные директивы для промежуточных серверов, например, прокси-серверов.

Пример запроса.

```
GET /locate?keywords=HTTP+description HTTP/1.1
Date: Mon, 15 Dec 2004 12:18:15 GMT
Accept: image/gif, image/jpg, */*
Accept-Charset: iso-8859-1, *, utf-8
Accept-Language: en
Connection: keep-Alive
User-Agent: Mozilla/4.7 [en] (Win98; u)
```

Группа кодов	**Код**	**Фраза-объяснение, следующая за кодом**	**Значение кода**
1xx			Информационное сообщение
2xx			Успешная обработка
	200	OK	Все нормально
	201	Created	Документ создан
3xx			Перенаправление запроса
	301	Moved Permanently	Ресурс перемещен
	302	Moved Temporarily	Ресурс перемещен временно
4xx			Ошибка клиента
	400	Bad Request	Некорректно составленный запрос
	401	Unauthorized	Нужна аутентификация клиента
	403	Forbidden	Доступ к ресурсу запрещен
	404	Not Found	Запрашиваемый ресурс отсутствует
5xx			Ошибка сервера
	500	Internal Server Error	Внутренняя ошибка сервера

Таблица 12. Некоторые коды статуса ответа HTTP и их объяснение.

Ответ сервера на HTTP-запрос состоит либо только из запрашиваемого клиентом документа, либо в дополнение к нему содержит код статуса ответа и, возможно, несколько заголовков ответа.

```
<Response>    ::= ( <Content> )?
                | <HTTP-Version> <Code> <Explanation> CrLf
                  <Resp-Header>* ( <Content> )?
<Resp-Header> ::= <Field> : <Value> CrLf
CrLf      ::= '\r''\n'
```

Некоторые коды статуса ответа поясняются в Таблице 12.

Возможны следующие заголовки ответа.

- `Allowed`
  Перечисляются через пробел доступные для пользователя методы запросов по данному URI.
- `Public`
  Перечисляет доступные всем методы запросов.
- `Content-Length`, `Content-Type`, `Content-Encoding` и `Content-Language`
  Задают размер содержимого в байтах (подразумевается, что содержимое имеет бинарный формат и не должно разбиваться на строки при чтении), его формат или MIME-тип, кодировку и язык.
- `Date`
  Дата создания содержащегося документа или объекта.
- `Last-Modified`
  Дата последнего изменения содержащегося объекта.
- `Expires`
  Дата, послед которой содержащийся объект считается устаревшим.
- `URI`
  URI содержащегося объекта.
- `Title`
  Заголовок содержащегося документа.
- `Server`
  Описывает серверную программу.
- `Retry-After`
  Определяет промежуток времени на обработку запроса, до прохождения которого не надо направлять запрос повторно, если ответа на него еще нет.

Пример HTTP-ответа.

```
HTTP/1.0 200 OK
Content-Length: 2109
Content-Type: text/html
Expires: 0
Last-Modified: Thu, 08 Feb 2001 09:23:17 GMT
Server: Apache/1.3.12
<HTML> <HEAD> <TITLE> … </TITLE> </HEAD>
<BODY> …
</BODY>
</HTML>
```

Уровень пользовательского интерфейса в J2EE

Компоненты пользовательского интерфейса в Web-приложениях, построенных как по технологии J2EE, так и по .NET, реализуют обработку HTTP-запросов, приходящих от браузера, и выдают в качестве результатов HTTP-ответы, содержащие сгенерированные HTML-документы с запрашиваемыми данными. Сами запросы автоматически строятся браузером на основе действий пользователя — в основном, переходов по ссылкам и действий с элементами управления в HTML-формах.

Если стандартных элементов управления HTML не хватает для реализации функций приложения или они становятся неудобными, используются специальные библиотеки элементов управления WebUI, предоставляющие более широкие возможности для пользователя и более удобные с точки зрения интеграции с остальными компонентами приложения.

В рамках J2EE версии 1.4 два основных вида компонентов WebUI — ***сервлеты (servlets)*** и ***серверные страницы Java (Java Server Pages, JSP)*** — отвечают, соответственно, за обработку действий пользователя и представление данных в ответе на его запросы. В следующей версии J2EE 5.0 будут также использоваться ***компоненты серверного интерфейса Java (Java Server Faces, JSF)*** — библиотека элементов управления WebUI.

Сервлеты представляют собой классы Java, реализующие обработку запросов HTTP и генерацию ответных сообщений в формате этого протокола. Страницы JSP являются упрощенным представлением сервлетов, основанным на описании генерируемого в качестве ответа HTML-документа при помощи смеси из его постоянных элементов и кода на Java, генерирующего его изменяемые части. При развертывании Web-приложения содержащиеся в нем страницы JSP транслируются в сервлеты и далее работают в таком виде. Описание генерируемых документов на смеси из HTML и Java делает страницы JSP более удобными для разработки и значительно менее объемными, чем получаемый из них и эквивалентный по функциональности класс-сервлет.

Сервлеты

Интерфейс Java-сервлетов определяется набором классов и интерфейсов, входящих в состав пакетов `javax.servlet и javax.servlet.http`, являющихся частью J2EE SDK. Первый пакет содержит классы, описывающие независимые от протокола сервлеты, второй — сервлеты, работающие с помощью протокола HTTP.

Основные классы и интерфейсы пакета `javax.servlet.http` следующие.

- `HttpServlet`
  Предназначен для реализации сервлетов, работающих с HTTP-сообщениями. Содержит защищенные методы, обрабатывающие отдельные методы HTTP-запросов, из которых наиболее важны **`void`** `doGet(HttpServletRequest, HttpServletResponse)`, определяющий обработку `GET`-запросов, и **`void`** `doPost(HttpServletRequest, HttpServletResponse)`, обрабатывающий `POST`-запросы. В обоих методах первый параметр содержит всю информацию о запросе, а второй — о генерируемом ответе.
- `HttpServletRequest` и `HttpServletResponse` — интерфейсы, содержащие методы для получения и установки (второй) заголовков и других атрибутов HTTP-запросов и ответов. Второй интерфейс также содержит метод, возвращающий поток вывода для построения содержимого ответа.
- `Cookie`
  Класс, представляющий закладки сервера, которые хранятся на клиентской машине для запоминания информации о данном пользователе.
- `HTTPSession`
  Интерфейс, предоставляющий методы для управления сеансом обмена HTTP-сообщениями. Информация о сеансе используется в том случае, если она должна быть доступна нескольким сервлетам.

При развертывании J2EE приложения, помимо самих классов сервлетов, надо создать их ***дескриптор развертывания***, который оформляется в виде XML-файла web.xml.

Web-приложение поставляется в виде архива .war, содержащего все его файлы. На самом деле это zip-архив, расширение .war нужно для того, чтобы Web-контейнер узнавал архивы развертываемых на нем Web-приложений. Содержащаяся в этом архиве структура директорий Web-приложения должна включать директорию WEB-INF, вложенную непосредственно в корневую директорию приложения. Директория WEB-INF содержит две поддиректории — classes для .class-файлов сервлетов, классов и интерфейсов EJB-компонентов и других Java-классов, и lib

для .jar и .zip файлов, содержащих используемые библиотеки. Файл web.xml также должен находится непосредственно в директории WEB_INF.

Заголовок дескриптора развертывания сервлета выглядит так.

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc...BEB Web Application 2.2.. EN"
http://java.sun.com/j2ee/dtds/web-app_2_2.dtd>
```

Содержимое дескриптора развертывания помещается внутри тега `<web-app>`. В нем указывается список сервлетов, входящих в приложение и отображение сервлетов в URL, запросы к которым они обрабатывают. Один сервлет описывается в следующем виде.

```
<servlet>
  <servlet-name>ServletName</servlet-name>
  <servlet-class>com.company.deprtment.app.ServletClassName</servlet-class>
  <description>…</description>
  <init-param>
    <param-name>ParameterName</param-name>
    <param-value>ParameterValue</param-value>
    <description>…</description>
  </init-param>
</servlet>
```

Значения параметров инициализации сервлета можно получить с помощью методов `String getInitParameter(String)` и `Enumeration getInitParametersNames()` связанного с сервлетом объекта класса `ServletContext`.

Отображение сервлета на URL описывается так.

```
<servlet-mapping>
  <servlet-name>ServletName</servlet-name>
  <url-pattern>URL</url-pattern>
</servlet-mapping>
```

Серверные страницы Java

Серверные страницы Java [7,8] представляют собой компоненты, разрабатываемые на смеси из HTML и Java и предназначенные для динамического создания HTML-документов, содержащих результаты обработки запросов пользователя. Таким образом, JSP обычно играют роль представления в образце «данные-представление-обработчик», принятом за основу архитектуры приложений J2EE. Результатом работы JSP является HTML-страничка, а вставки Java кода служат для построения некоторых ее элементов на основе результатов работы приложения.

При работе Web-приложения JSP компилируются в сервлеты специального вида. При этом основное содержание страницы JSP превращается в метод `doGet()`, в котором HTML-элементы записываются в поток содержимого ответа в неизменном виде, а элементы Java-кода преобразуются в код, записывающий некоторые данные в тот же поток на основании параметров запроса или данных приложения.

Для развертывания JSP-страниц необходимо их описание в дескрипторе развертывания приложения web.xml, которое устроено так же, как описание сервлетов. Сами JSP-страницы помещаются, вместе с HTML-файлами и другими файлами, используемыми приложением в корневую директорию этого приложения или ее поддиректории.

Основные интерфейсы и базовые классы JSP-страниц и их отдельных элементов находятся во входящих в J2EE SDK пакетах `javax.servlet.jsp, javax.servlet.jsp.el, javax.servlet.jsp.tagext`.

Элементами JSP-страниц могут быть обычные теги HTML, а также специальные элементы JSP — *директивы*, *теги* или *действия (tags, actions)* и *скриптовые элементы*.

JSP-директивы описывают свойства страницы в целом и служат для передачи информации механизму управления JSP-страницами.

Директивы имеют следующий общий синтаксис.

```
<%@ directive attribute1="value1" ... attributeN="valueN" %>.
```

Основные директивы JSP следующие.

- Директива **`page`** предоставляют общую информацию о данной странице и статически включаемых в нее файлах. Такая директива на странице может быть только одна. Она может иметь следующие атрибуты.
  - `import = "`имена включаемых классов и пакетов через запятую`"`
    Порождает соответствующую Java-директиву **`import`** в сгенерированном коде сервлета.
  - `contentType = "`MIME-тип[`;charset=`таблица символов]`"`
    Задает тип MIME для генерируемого документа. По умолчанию используется `text/html`. Эквивалентен скриплету
    `<% response.setContentType(`MIME-тип`); %>` (см. далее).
  - `isThreadSafe = "true|false"`
    Значение `true` позволяет использовать один экземпляр сервлета, полученного из странички, для обработки множественных запросов. При этом необходимо синхронизовать доступ к данным этого сервлета.
  - `session = "true|false"`
    Значение `true` предписывает привязать сервлет к имеющейся HTTP-сессии, значение `false` говорит, что сессии использоваться не будут и обращение к переменной `session` приведет к ошибке.
  - `autoFlush = "true|false"`
    Определяет необходимость сбрасывать буфер вывода при заполнении.
  - `buffer = "`размер в KB|`none"`
    Задает размер буфера для выходного потока сервлета.
  - `extends = "`наследуемый класс`"`
    Определяет класс, наследуемый сгенерированным из данной JSP сервлетом.
  - `errorPage = "`url странички с информацией об ошибках`"`
    Определяет страницу, которая используется для обработки исключений, не обрабатываемых в рамках данной.
  - `isErrorPage = "true|false"`
    Допускает или запрещает использование данной страницы в качестве страницы обработки ошибок.
  - `language = "java"`
    Определяет язык программирования, применяемый в скриптовых элементах данной страницы. Пока есть возможность использовать только Java. Впоследствии предполагается (аналогично .NET) разрешить использование других языков, код которых будет также транслироваться в байт-код, интерпретируемый JVM.
- Директива **`include`** обеспечивает статическое (в ходе трансляции JSP в сервлет) включение в страничку внешнего документа. Она имеет атрибут `file`, значением которого должна быть строка, задающая URL включаемого файла.
- Директива **`taglib`** указывает используемую в данной странице библиотеку пользовательских тегов. Она имеет два атрибута — `uri`, значением которого является URI библиотеки, и `prefix`, определяющий префикс тегов из данной библиотеки. Префикс употребляется в дальнейшем с тегами только данной библиотеки. Он не может быть пустым и не должен совпадать с одним из зарезервированных префиксов `jsp`, `jspx`, `java`, `javax`, `servlet`, `sun`, `sunw`.

Теги или *действия* определяют основные действия, выполняемые при обработке данных и построении результирующего документа.

Теги могут быть *стандартными*, использование которых возможно в любой странице без дополнительных объявлений, или *пользовательскими*, которые могут употребляться, только если

предварительно с помощью директивы `taglib` была подключена содержащая их библиотека. Любой тег имеет следующий синтаксис.

```
<tagprefix:tag attribute1="value1" … attributeN="valueN" />
```

Теги могут содержать вложенные теги, такие как `jsp:param`, `jsp:attribute`. В этом случае они выглядят следующим образом.

```
<tagprefix:tag attribute1="value1" … attributeN="valueN">
  … (вложенные теги)
</tagprefix:tag>
```

Стандартные теги имеют префикс `jsp`, а префикс пользовательских тегов определяется в директиве `taglib`, подключающей содержащую их библиотеку.

Имеется довольно много стандартных тегов. Основные из них следующие.

- `jsp:include`
  Определяет динамическое включение некоторой страницы или файла в данную страницу при обработке запроса. С помощью вложенных тегов `jsp:param` может указывать один или несколько пар параметр-значение в качестве параметров включаемой страницы.
  Имеет атрибуты `page`, определяющий URL включаемой страницы, и `flush`, имеющий значения `true` или `false` в зависимости от того, нужно ли сразу после включения сбросить буфер выходного потока в генерируемом ответе или нет.
- `jsp:useBean`
  Определяет используемый объект или компонент. Фактически такой тег эквивалентен декларации переменной определенного типа, инициализируемой определенным объектом и доступной в рамках некоторого контекста.
  Имеет следующие атрибуты.
  - `id = "имя объекта"`
    Задает имя объекта, которое будет использоваться в коде JSP. Должно быть уникально в пределах страницы.
  - `class = "класс объекта"`
    Задает класс этого объекта.
  - `scope = "page|request|session|application"`
    Задает область видимости декларируемого объекта.
  - `type = "тип используемой ссылки на объект"`
    Указанный тип должен быть предком класса объекта. Это тип декларируемой переменной, а класс объекта определяет истинный тип объекта, хранящегося в ней.
- `jsp:setProperty`, `jsp:getProperty`
  Устанавливает или получает значение свойства объекта.
  Атрибут `name` определяет имя объекта, чье свойство используется, а атрибут property — имя свойства.
  Тег `jsp:getProperty` записывает полученное значение свойства в результирующий документ в виде строки (результата вызова `toString()` для этого значения).
  Тег `jsp:setProperty` имеет также дополнительный атрибут — либо `value`, значение которого присваивается свойству, либо `param`, который указывает имя параметра запроса, значение которого записывается в свойство. Если в теге `jsp:setProperty` вместо имени свойства в атрибуте `property` указан символ `*`, то всем свойствам указанного объекта с именами, совпадающими с именами параметров запроса, будут присвоены значения соответствующих параметров.
- `jsp:forward`
  Этот тег употребляется для перенаправления запроса на обработку другой странице. URL этой страницы указывается в качестве значения атрибута `page`. В качестве этого URL может использоваться JSP-выражение (см. далее), вычисляемое на основе параметров запроса.

С помощью вложенных тегов `jsp:param` можно передать одно или несколько значений параметров странице, на которую переключается управление.

- `jsp:plugin`
  Этот тег вставляет аплет или компонент JavaBean на страницу. Параметры инициализации компонента могут быть заданы при помощи вложенного тега `jsp:params`. Кроме того, `jsp:plugin` имеет следующие атрибуты.
  - `type = "bean|applet"`
    Задает вид вставляемого компонента.
  - `code = "`имя файла класса компонента (включая расширение .class)`"`
  - `codebase = "`имя директории, в которой находится файл класса компонента`"`
    Если этот атрибут отсутствует, используется директория, содержащая данную JSP-страницу.
  - `name = "`имя используемого экземпляра компонента`"`
  - `archive = "`список разделенных запятыми путей архивных файлов, которые будут загружены перед загрузкой компонента`"`
    Эти архивы содержат дополнительные классы и библиотеки, необходимые для работы компонента.
  - `align = "bottom|top|middle|left|right"`
    Задает положение экземпляра компонента относительно базовой строки текста содержащего его HTML-документа.
  - `height = "`высота изображения объекта в точках`"`
  - `width = "`ширина изображения объекта в точках`"`
  - `hspace = "`ширина пустой рамки вокруг объекта в точках`"`
  - `vspace = "`высота пустой рамки вокруг объекта в точках`"`
  - `jreversion = "`версия JRE, необходимая для работы компонента`"`
    По умолчанию используется версия 1.1.

Пользовательские теги могут быть определены для выполнения самых разнообразных действий. Одна из наиболее широко используемых библиотек тегов core (подключаемая с помощью директивы `<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix = "c" %>`), содержит теги, представляющие все конструкции языка Java.

Например, с помощью тега `<c:set var="variable" value="value" />` можно присвоить значение `value` в переменную `variable`, с помощью тега `<c:if test="expression">…</c:if>` можно выполнить код JSP, расположенный внутри этого тега только при том условии, если `expression` имеет значение **`true`**, с помощью тега `<c:forEach var="variable" items="collection">…</c:forEach>` можно выполнить содержащийся в нем код для всех элементов коллекции `collection`, используя для обращения к текущему элементу переменную `variable`.

Таким образом, весь код, который может быть записан в виде скриптовых элементов (см. далее), можно записать и в виде тегов. Применение тегов при этом вносит некоторые неудобства для Java-программиста. Но оно же делает код JSP-страниц более однородным и позволяет обрабатывать его с помощью инструментов для разработки Web-сайтов, гораздо лучше приспособленных к работе с размеченным с помощью тегов текстом.

Скриптовые элементы могут иметь один из следующих трех видов.

- *JSP-объявления*, служащие для определения вспомогательных переменных и методов. Эти переменные становятся впоследствии полями сгенерированного по JSP сервлета, а методы — его методами. Синтаксис JSP-объявления следующий.
  `<%!` код на Java `%>`
- *Скриплеты*, которые служат для вставки произвольного кода, обрабатывающего данные запроса или генерирующего элементы ответа, в произвольное место. Они имеют

следующий синтаксис.
`<%` код на Java `%>`

- *JSP-выражения*, используемые для вставки в какое-то место в результирующем документе вычисляемых значений (они также могут использоваться в качестве значений атрибутов тегов). Их синтаксис может иметь три вида.
  `<% =` выражение на Java `%>`
  `${`выражение на Java`}`
  `#{`выражение на Java`}`

Различий между первым и вторым способом представления выражений практически нет. Выражение третьего типа вычисляется отложено — вычисление его значения происходит только тогда, когда это значение действительно понадобится.

Комментарии в коде JSP оформляются в виде содержимого тега `<%-- … --%>`. Встречающийся в них код не обрабатывается во время трансляции и не участвует в работе полученного сервлета. Элементы кода внутри HTML-комментариев `<!-- … -->` обрабатываются так же, как и в других местах — они генерируют содержимое комментариев в результирующем HTML-документе.

Помимо объявленных в объявлениях и тегах `jsp:useBean` переменных в скриптовых элементах могут использоваться неявно доступные объекты, связанные с результирующим сервлетом, обрабатываемым им запросом и генерируемым ответом, например, следующие.

- `request` — запрос клиента (тип `ServletRequest`).
- `param` — параметры запроса (тип `Map`).
- `response` — ответ сервера (тип `ServletResponse`).
- `out` — выходной поток сервлета (тип `PrintWriter`).
- `session` — сеанс (тип `HttpSession`).
- `application` —приложение (`ServletContext`).
- `config` — конфигурация сервлета (`ServletConfig`).
- `pageContext` — контекст страницы (`javax.servlet.jsp.PageContext`).
- `exception` — произошедшее исключение.

Ниже приведен пример JSP страницы, генерирующей таблицу балансов клиентов некоторой организации в долларовом и рублевом выражениях.

```
<%@ page import="java.util.Date, java.util.Iterator,
    com.company.Client" %>
<jsp:useBean id="clients" class="com.company.ClientList"
    scope="page" />
<jsp:useBean id="convertor" class="com.company.ExchangeRate"
    scope="page" />
<html>
<head>
<title>Table of clients</title>
</head>
<body>
<h3 align="center">Table of clients</h3>
Created on <%= new Date() %> <br><br>

<table width="98%"  border="1" cellspacing="1" cellpadding="1">
  <tr>
  <%!
    private double dollarsToRubles(double m)
    {
      return m*convertor.getDollarToRubleRate(new Date());
    }
  %>
    <th width="50%" scope="col">Client</th>
    <th width="25%" scope="col">Balance, dollars</th>
    <th width="25%" scope="col">Balance, rubles</th>
```

```
  </tr>
  <%
     Iterator it = clients.getNumberOfClients().iterator();
     while(it.hasNext())
     {
       Client client = (Client)it.next();
       %>
  <tr>
    <td> ${client.getFullName()} </td>
    <td> ${client.getBalance()} </td>
    <td> ${dollarsToRubles(client.getBalance())} </td>
  </tr>
  <%
     }
  %>
</table> <br><br>

<jsp: include page="footer.txt" flush= "true" />
</body>
</html>
```

Уровень пользовательского интерфейса в .NET

Разработка компонентов пользовательского интерфейса Web-приложений в рамках .NET выделена в виде отдельной технологии ASP.NET [9,10] и в целом очень похожа на разработку тех же компонентов в J2EE. В .NET имеются те же виды компонентов: элементы управления, представленные *серверными элементами управления HTML (HTML server controls)* и просто *серверными элементами управления (Web Server Controls)*, обработчики HTTP запросов (аналог сервлетов в Java), представленные интерфейсами `IHttpHandler` и `IHttpAsyncHandler`, и так называемые ***Web-формы (Web forms)***, аналог серверных страниц Java.

Элементы управления WebUI в .NET могут быть размещены на HTML-страницах, но выполняются на сервере. Библиотеки таких компонентов находятся в сборке System.Web и в пространстве имен `System.Web.UI`, вместе с его подпространствами. Их употребление в рамках HTML-документа оформляется в виде специальных тегов с атрибутом `runat`, имеющим значение `server`. Ниже приведен пример использования компонента `System.Web.UI.WebControls.Button` в коде Web-формы.

```
<%@ Page Language="C#" AutoEventWireup="True" %>

<html>
<head>
   <script language="C#" runat=server>
      void OnButtonClick(object sender, EventArgs e)
      {
         Message.Text="Hello World!!";
      }
   </script>
</head>
<body>
   <form runat="server">
      <h3>Button Example</h3>
      Click on the submit button.<br><br>

      <asp:Button id="MyButton"
           Text="Submit"
           OnClick="OnButtonClick"
           runat="server"/>

      <p>
      <asp:label id="Message" runat="server"/>
   </form>
</body>
</html>
```

Аналогом сервлетов в .NET являются объекты, реализующие интерфейсы `System.Web.IHttpHandler` и `System.Web.IHttpAsyncHandler`. Оба они являются частью *программного интерфейса Web-сервера* Microsoft *(Internet Server Application Program Interface, ISAPI)*. Первый интерфейс предназначен для синхронной обработки запросов, с блокированием на время обработки вызвавшего ее потока Web-сервера. Второй интерфейс позволяет реализовывать такую обработку в отдельном потоке.

Единственный метод первого интерфейса — **`void`** `ProcessRequest (System.Web.HttpContext context)`. Все данные, связанные с запросом, ответом на него, приложением и контекстом, в котором работает данный обработчик, можно получить, используя различные свойства параметра этого метода.

Интерфейс `IHttpAsyncHandler` имеет два метода — `IAsyncResult BeginProcessRequest( HttpContext context, AsyncCallback cb,` **`object`** `extraData)` и **`void`** `EndProcessRequest( IAsyncResult result)`. Первый вызывается при передаче запроса данному обработчику, второй — для прекращения обработки.

Web-формы .NET являются аналогом серверных страниц Java. Они так же оформляются в виде документов, содержащих конструкции как HTML, так и одного из языков программирования, используемых в рамках .NET, и специальные конструкции, аналогичные директивам, тегам и скриптовым элементам JSP.

Специальные конструкции Web-форм включают *директивы*, имеющие тот же самый смысл, что и для серверных страниц Java, *объявления*, аналогичные JSP-объявлениям, конструкции *встроенного кода (code render)* и конструкции *привязки к данным (data binding expressions)*.

Директивы Web-форм имеют в целом точно такой же синтаксис, как и директивы JSP: `<%@ directive attribute1="value1" ... attributeN="valueN" %>`. Список директив шире, чем в JSP: имеется директива **`Page`**, аналог **`page`** в JSP, но с несколько отличающимся списком атрибутов, директива **`Import`**, аналог **`include`**, директива **`Control`** для описания пользовательских элементов управления, директива **`Register`** для определения синонимов (алиасов), и пр.

Объявления полей данных и методов в Web-формах обрамляются в тег `<script> … </script>`. Такой тег должен иметь атрибут `runat` со значением `server` и атрибут `language`, который определяет язык кода, написанного внутри тега. Он может также иметь атрибут `src` для указания URL файла, код из которого должен быть вставлен перед содержимым тега.

Конструкции встроенного кода обрамляются в тег `<% … %>`. Как и в JSP, могут использоваться выражения в виде `<%=… %>`. Комментарии тоже оформляются, как и в JSP, в виде тегов `<%-- … --%>`.

Конструкции привязки к данным имеют синтаксис `<%# expression %>` и работают примерно так же, как и выражения встроенного кода. Они могут использоваться и в значениях атрибутов элементов управления.

Конфигурационные файлы компонентов .NET, являющиеся аналогами дескрипторов развертывания в J2EE, оформляются в виде XML-документов специального вида и размещаются в различных директориях Web-приложения. В качестве корневого тега таких документов всегда выступает тег `configuration`. Он может содержать теги `location`, которые определяют конфигурацию для ресурсов, путь к которым указывается в атрибуте `path` таких тегов. Теги `location` для компонентов ASP.NET содержат тег `system.web`, который, в свою очередь, может содержать следующие теги (перечислены не все возможные, более полную информацию см. в [11]).

- `authentication`
  Определяет используемый вид аутентификации — атрибут `mode` задает используемый механизм (`Windows`, `Forms`, `Passport` или `None`), вложенные теги forms описывают свойства отдельных форм, используемых для аутентификации.
- `authorization`
  Определяет права доступа для пользователей, ролей и отдельных методов HTTP-запросов.

Разрешения на доступ указываются в атрибутах вложенного теги `allow`, запреты — в атрибутах вложенного тега `deny`.

- `compilation`
  Определяет параметры компиляции компонента ASP.NET.
- `customErrors`
  Определяет специфические для данного приложения ошибки и URL, на которые переходит управление при возникновении этих ошибок.
- `globalization`
  Определяет кодировки и локализацию запросов и ответов.
- `httpHandlers`
  Определяет отображение адресов и методов запросов на обрабатывающие их объекты типа `IHttpHandelr` или `IHttpHandlerFactory`.
- `pages`
  Определяет настройки для отдельных страниц.
- `sessionState`
  Описывает настройки для поддержки состояния сеансов работы с данным приложением.

Литература к Лекции 14

[1] Р. Монсон-Хейфел. Enterprise JavaBeans. СПб.: Символ-Плюс, 2002.
[2] Enterprise JavaBeans Specification, version 2.1.
Доступны по ссылке http://java.sun.com/products/ejb/docs.html.
[3] Сайт проекта NetBeans http://www.netbeans.org/.
[4] Документация MSDN по ADO.NET
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconaccessingdatawithadonet.asp.
[5] Hypertext Transfer Protocol — HTTP/1.1. RFC 2616.
Доступно по ссылке http://www.w3.org/Protocols/rfc2616/rfc2616.html.
[6] http://www.opennet.ru/docs/RUS/http/index.html
[7] Документация по JSP http://java.sun.com/products/jsp/docs.html.
[8] Б. У. Перри. Java сервлеты и JSP: сборник рецептов. М.: Кудиц-Образ, 2005.
[9] Документация MSDN по ASP.NET
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconintroductiontoasp.asp.
[10] Р. Андерсон, Б. Френсис, А. Хомер, Р. Хоуорд, Д. Сассмэн, К. Уотсон. ASP.NET 1.0 для профессионалов. М.: Лори, 2004.
[11] Схема конфигурационных файлов ASP.NET
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/gngrfASPNETConfigurationSectionSchema.asp.
[12] П. Аллен, Дж. Бамбара, М. Ашнаульт, Зияд Дин, Т. Гарбен, Ш. Смит. J2EE. Разработка бизнес-приложений. СПб.: ДиаСофт, 2002.
[13] Д. Просиз. Программирование для Microsoft.NET. М.: Русская редакция, 2003.

Лекция 15. Развитие компонентных технологий

Аннотация

Рассказывается о некоторых компонентных средах и технологиях, обрисовывающих направления дальнейшего развития стандартных платформ разработки Web-приложений. Также рассматриваются Web-службы, представляющие собой компонентную технологию другого уровня.

Ключевые слова

Struts, JSF, объектно-реляционный преобразователь, Hibernate, JDO, аспектно-ориентированное программирование, Spring, Web-службы, SOA, WSDL, SOAP, UDDI.

Текст лекции

Программисты, которые долгое время работают с технологиями разработки Web-приложений, представленными в последних двух лекциях, отмечают ряд неудобств, связанных с разработкой отдельных компонентов, построением приложения в целом и настройкой отдельных аспектов его работы. В данной лекции рассказывается о развитии компонентных технологий разработки Web-приложений, нацеленном на повышение их гибкости, удобства их создания и поддержки, а также на снижение трудоемкости внесения изменений в приложения такого рода.

В ряде аспектов разработка отдельных компонентов в рамках .NET несколько проще, тем разработка компонентов с той же функциональностью в рамках J2EE версии 1.4. В то же время разработка приложений в целом в рамках J2EE проще для начинающих разработчиков, поскольку имеющаяся по этой платформе документация четче определяет общую структуру приложений и распределение ответственности между разными типами компонентов в нем.

Большим достоинством J2EE является прозрачность и предсказуемость ее развития, поскольку все его шаги открыты в рамках четко определенного процесса компании Sun для внесения изменений в спецификации платформы и на каждом из этих шагов учитываются интересы множества участников. Развитие платформы J2EE определяется большим количеством открытых проектов отдельных разработчиков и организаций, предлагающих свои решения по построению сред функционирования Web-приложений (Web application frameworks).

Развитие же платформы .NET находится целиком в руках компании Microsoft и пока не является прозрачным для тех, кто не работает в ней или в одной из близких к ней компаний-партнеров. На основании выступлений отдельных представителей компании можно делать выводы, касающиеся лишь общих планов развития платформы, без каких-либо технических деталей. Поэтому в данной лекции рассматриваются, в основном, направления развития технологий J2EE.

Развитие технологий J2EE

Ряд разработчиков выделяет следующие проблемы удобства разработки и поддержки приложений J2EE версии 1.4.

- Громоздкость разработки компонентов EJB и неудобство их использования для описания структуры предметной области.
  Для разработки простейшего такого компонента необходимо определить два интерфейса, класс компонента и написать дескриптор развертывания.
  Полученные классы и интерфейсы достаточно сильно отличаются от обычных классов Java, с помощью которых разработчики описывали бы предметную область в рамках обычного приложения на платформе J2SE. Поэтому гораздо тяжелее вносить в них изменения, связанные с изменением требований к соответствующим объектам предметной области.
- Отсутствие удобной поддержки для отображения иерархии наследования классов в структуру базы данных приложения.
  Данные класса-предка и класса-наследника могут храниться в одной таблице, в разных и несвязанных таблицах, или общая часть данных может храниться в одной таблице, а

специфические данные класс-наследника — в другой. Однако для обеспечения правильной синхронизации данных в каждом из этих случаев достаточно много кода надо написать вручную. Поскольку во многих приложениях объектные модели данных содержат классы, связанные отношением наследования, отсутствие вспомогательных механизмов, автоматически обеспечивающих отображение таких классов на структуру базы данных, приносит много неудобств.

- Невозможность использовать в рамках приложения компоненты EJB, соответствующие данным в базе данных, и временные объекты того же типа, для которых не нужно иметь соответствующих записей в таблицах баз данных.
  Часто такая возможность оказывается удобной при программировании различных методов обработки данных внутри приложений.
- Громоздкость разработки сервлетов для обработки простых (тем более, сложных) запросов пользователя.
  При этом необходимо полностью проанализировать запрос, часто — найти в содержащемся в нем документе HTML поля формы, заполненной пользователем, и указанную им операцию их обработки. Только после этого можно переходить к собственно выполнению этого запроса, что, казалось бы, является основной функцией сервлета. Таким образом, большое количество усилий тратится только на то, чтобы выделить из запроса операцию, которую пользователь хочет произвести, а также ее аргументы.
- Неудобство использования в рамках JSP-страниц специализированных элементов пользовательского интерфейса. Для сравнения: в рамках ASP.NET можно использовать библиотечные и пользовательские элементы управления, которые помещаются на страницу при помощи специального тега, а в параметрах этого тега указываются, в частности, методы для обработки событий, связанных с действиями пользователя.

Для решения этих проблем используются различные библиотеки, инструменты и компонентные среды, созданные в сообществе Java-разработчиков. Некоторые такие библиотеки и техники станут стандартными средствами в рамках платформы J2EE новой версии 5.0 [1].

Jakarta Struts

Среда Jakarta Struts [2,3] создавалась затем, чтобы упростить разработку компонентов Web-приложения, предназначенных для обработки запросов пользователей, и сделать эту обработку более гибкой.

Основные решаемые такими компонентами задачи можно сформулировать следующим образом:

- выделить сам логический запрос и его параметры из HTML документа, содержащегося в HTTP-запросе;
- проверить корректность параметров запроса и сообщить пользователю об обнаруженной некорректности наиболее информативным образом;
- преобразовать корректный логический запрос и его параметры в вызовы соответствующих операций над объектами предметной области;
- передать результаты сделанных вызовов компонентам, ответственным за построение их представления для пользователя.

Как и в рамках базовой платформы J2EE, в Struts основным архитектурным стилем для Web-приложений является образец «данные-представление-обработка». При этом роль представления играют JSP-страницы, а роль обработчиков — сервлеты. Основные отличия Struts от стандартной техники J2EE связаны с большей специализацией сервлетов и некоторой стандартизацией обмена данными между сервлетом, обрабатывающим запросы пользователя, и JSP-страницей, представляющей их результаты.

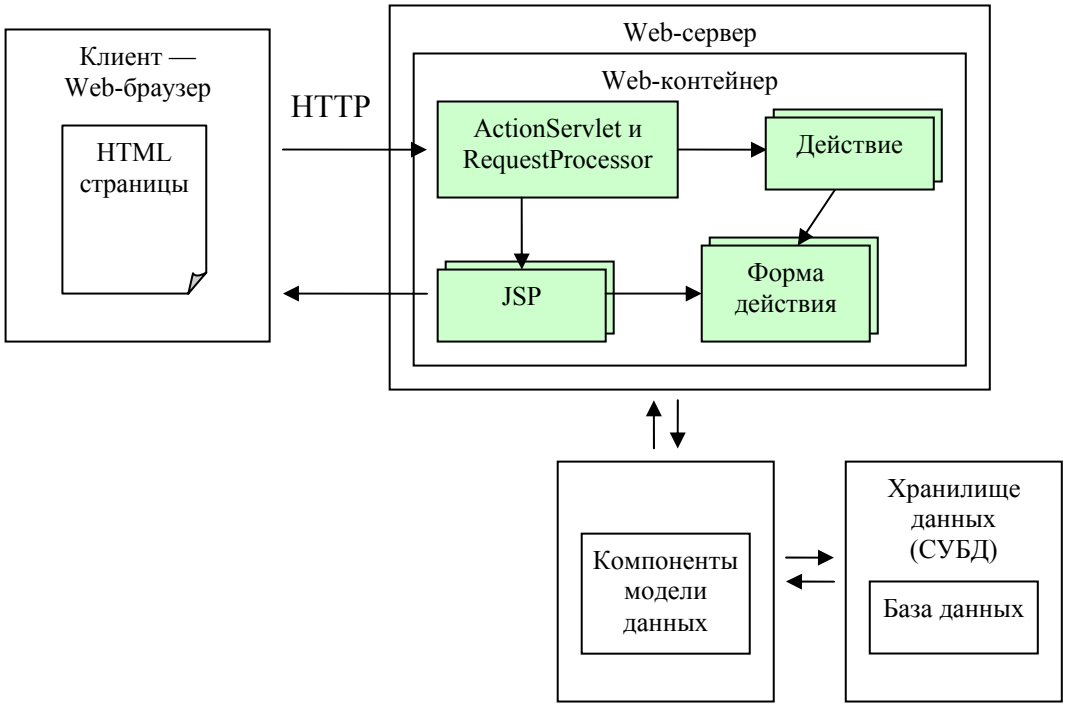


Рисунок 79. Общая схема архитектуры Web-приложений на основе Struts.

В рамках приложения на основе Struts используется ровно один стандартизированный сервлет (ActionServlet), анализирующий запросы пользователя и выделяющий из каждого запроса *действие (action)*, которое пользователь пытается выполнить. Для Интернет-магазина таким действиями, например, могут быть аутентификация (предоставление своего имени и пароля), получение данных о товаре, поиск товара, добавление товара к уже заказанным, изменение заказа, предоставление прав на скидку, выполнение заказа, получение статуса выполнения заказа, отмена заказа и пр. Для каждого действия создается отдельный *класс действия*. Такой класс должен быть наследником класса `org.apache.struts.action.Action` из библиотеки Struts и перегружать метод `ActionForward execute(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)` — именно он и вызывается для выполнения этого действия.

Один из параметров метода `execute()` в классе действия имеет тип *формы действия* `org.apache.struts.action.ActionForm`. Для каждого действия определяется свой класс формы действия, наследующий классу `ActionForm`. Объекты этого класса используются для передачи параметров действия — наиболее существенных данных запросов, описывающих данное действие.

В методе `execute()` класса действия обычно строятся или находятся компоненты бизнес-логики приложения, которые реализуют операции, соответствующие данному действию, а затем эти операции выполняются со значениями полей объекта формы действия в качестве аргументов.

Привязка запросов к действиям описывается в дополнительном конфигурационном файле struts-config.xml в формате XML, в теге `action-mappings`. Одно действие описывается с помощью вложенного тега `action`, который имеет следующие атрибуты.

- `path`
  Определяет шаблон URI, обращения к которым будут интерпретироваться как выполнение данного действия.
- `type`
  Определяет имя класса данного действия.
- `name`
  Задает уникальное имя для данного действия.

Привязка действий к определенным для них формам происходит с помощью тегов `from-bean`, вложенных в тег `form-beans`. Каждый тег `form-bean` имеет атрибут `name`, указывающий имя

действия для данной формы, и `type`, указывающий имя класса формы действия. Кроме того, такой тег может содержать вложенные теги `form-property`, описывающие свойства формы (в смысле JavaBeans) при помощи таких же атрибутов `name` (имя свойства) и `type` (тип свойства).

Помимо описанного механизма декомпозиции обработки запросов, среда Struts включает библиотеки классов Java, в том числе, классов часто встречающихся действий, и библиотеки пользовательских тегов, предназначенных для более удобного описания размещенных на JSP страницах элементов HTML-форм.

Java Server Faces

Java Server Faces (JSF) [4,5] включают библиотеку элементов управления WebUI `javax.faces` и две библиотеки пользовательских тегов, предназначенных для использования этих элементов управления в рамках серверных страниц Java. С помощью тегов библиотеки `jsf/html` элементы управления размещаются на странице, а с помощью тегов из `jsf/core` описывается обработка событий, связанных с этими элементами, и проверка корректности действий пользователя.

В аспекте построения WebUI на основе серверных страниц Java технология Java Server Faces является развитием подхода Struts (Struts включают решения и для других аспектов разработки приложений), предлагая более богатые библиотеки элементов WebUI и более гибкую модель управления ими. Эта модель включает следующие элементы.

- Возможность различного изображения абстрактного элемента управления (например, элемент управления «выбор одного из многих» может быть изображен как группа радио-кнопок, комбо-бокс или список).
- Возможность изменения визуальных стилей элементов управления.
- Возможность привязки изображаемых элементом управления значений к свойствам компонентов модели данных.
- Возможность привязки элементов управления к методам проверки корректности значений, устанавливаемых в них пользователем.

В дополнение к библиотекам элементов WebUI JSF предлагает определять правила навигации между страницами в конфигурационном файле приложения. Каждое правило относится к некоторому множеству страниц и при выполнении определенного действия или наступлении события предписывает переходить на некоторую страницу. Действия и события связываются с действиями пользователя или логическими результатами их обработки (такими результатами могут быть, например, успешная регистрация заказа в системе, попытка входа пользователя в систему с неправильным паролем и пр.).

Технология Java Server Face версии 1.2 войдет в состав будущей версии 5.0 платформы J2EE [6].

Управление данными приложения. Hibernate

Технологии обеспечения синхронизации внутренних данных приложения и его базы данных развиваются в настоящий момент достаточно активно. Технология EJB предоставляет соответствующие механизмы, но за счет значительного снижения удобства разработки и модификации компонентов. Обеспечение той же функциональности при более простой внутренней организации кода является основным направлением развития в данной области.

Возможным решением этой задачи являются ***объектно-реляционные преобразователи (object-relation mappers, ORM)***, которые обеспечивают автоматическую синхронизацию между данными приложения в виде наборов связанных объектов и данными, хранящимися в системе управления базами данных (СУБД) в реляционном виде, т.е. в форме записей в нескольких таблицах, ссылающихся друг на друга с помощью внешних ключей.

Одним из наиболее широко применяемых и развитых в технологическом плане объектно-реляционных преобразователей является Hibernate [7-9].

Базовая парадигма, лежащая в основе избранного Hibernate подхода, — это использование объектов обычных классов Java (быть может, оформленных в соответствии с требованиями

спецификации JavaBeans — с четко выделенными свойствами) в качестве объектного представления данных приложения. Такой подход даже имеет название-акроним POJO (plain old Java objects, простые старые Java-объекты), призванное показать его отличие от сложных техник построения компонентов, похожих на EJB.

Большое достоинство подобного подхода — возможность использовать один раз созданные наборы классов, представляющих понятия предметной области, в качестве модели данных любых приложений на основе Java, независимо от того, являются ли они распределенными или локальными, требуется ли в них синхронизация с базой данных и сохранение данных объектов или нет.

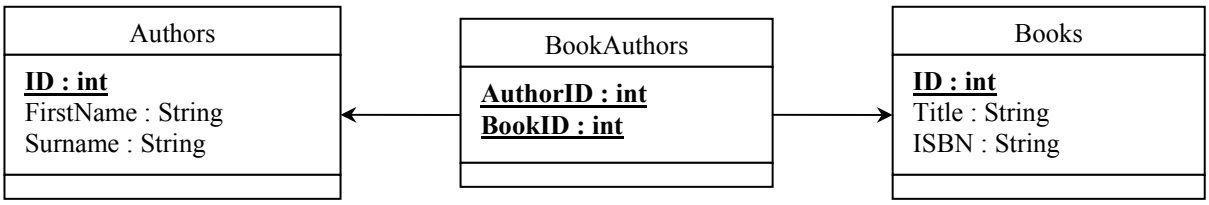


Рисунок 80. Реляционное представление данных о книгах и авторах.

Например, для представления в объектном виде данных о книгах и их авторах, соответствующих показанной на Рис. 80 группе таблиц, могут быть использованы представленные ниже классы.

```
import java.util.Set;
import java.util.HashSet;

public class Author
{
  private int id;

  private String firstName;
  private String surname;

  private Set books = new HashSet();

  public  int  getId ()       { return this.id; }
  private void setId (int id) { this.id = id;   }

  public String getFirstName ()               { return this.firstName;      }
  public void setFirstName (String firstName) { this.firstName = firstName; }

  public String getSurname ()               { return this.surname;    }
  public void   setSurname (String surname) { this.surname = surname; }

  public Set  getBooks ()          { return this.books;  }
  public void setBooks (Set books) { this.books = books; }
}

public class Book
{
  private int id;

  private String title;
  private String isbn;

  private Set authors = new HashSet();

  public  int  getId ()       { return this.id; }
  private void setId (int id) { this.id = id;   }

  public String getIsbn ()            { return this.isbn; }
  public void   setIsbn (String isbn) { this.isbn = isbn; }
```

```
  public String getTitle ()             { return this.title;  }
  public void   setTitle (String title) { this.title = title; }

  public Set  getAuthors ()            { return this.authors;    }
  public void setAuthors (Set authors) { this.authors = authors; }
}
```

Для определения отображения объектов этих классов в записи соответствующих таблиц используются конфигурационные файлы со следующим содержанием. Первый фрагмент представляет собой описание отображения объектов класса `Author` на записи таблицы `Authors`, которое обычно помещается в файл Author.hbm.xml.

```
<hibernate-mapping>
  <class name="Author" table="Authors">
    <id name="id" column="ID">
      <generator class="increment"/>
    </id>
    <property name="firstName" column="FirstName"/>
    <property name="surname"/ column="Surname">
    <set name="books" table="BookAuthors" inverse="true">
      <key column="AuthorID"/>
      <many-to-many column="BookID" class="Book"/>
    </set>
  </class>
</hibernate-mapping>
```

Второй фрагмент представляет собой содержание аналогичного файла Book.hbm.xml, описывающего отображение объектов класса `Book` на записи таблицы `Books`.

```
<hibernate-mapping>
  <class name="Book" table="Books">
    <id name="id" column="ID">
      <generator class="increment"/>
    </id>
    <property name="title" column="Title"/>
    <property name="isbn"/ column="ISBN">
    <set name="authors" table="BookAuthors" inverse="true">
      <key column="BookID"/>
      <many-to-many column="AuthorID" class="Author"/>
    </set>
  </class>
</hibernate-mapping>
```

При использовании объектов указанных классов в рамках приложения на основе Hibernate обеспечивается автоматическая синхронизация данных объектов с данными соответствующих записей, а также автоматическая поддержка описанного отношения типа «многие-ко-многим».

Кроме того, Hibernate поддерживает удобные средства для описания сложных соответствий между объектами и записями таблиц при использовании наследования. Так, легко могут быть поддержаны: отображение данных всех объектов классов-наследников в одну таблицу, отображение объектов разных классов-наследников в разные таблицы, отображение данных полей общего класса-предка в одну таблицу, а данных классов-наследников — в разные таблицы, а также смешанные стратегии подобных отображений.

В приложениях на основе Hibernate объекты одного и того же класса могут быть как хранимыми, т.е. представляющими данные, хранящиеся в базе данных, так и временными, не имеющими соответствующих записей в базе данных. Перевод объекта из одного из этих видов в другой осуществляется при помощи всего лишь одного вызова метода вспомогательного класса среды Hibernate.

С помощью дополнительной службы NHibernate [7] возможности среды Hibernate могут быть использованы и из приложений на базе .NET.

Java Data Objects

Еще более упростить разработку объектно-ориентированных приложений, данные которых могут храниться в базах данных, призвана технология Java Data Objects (JDO) [10,11].

В ее основе тоже лежит использование для работы с хранимыми данными обычных классов на Java, но в качестве хранилища данных может выступать не только реляционная СУБД, но вообще любое хранилище данных, имеющее соответствующий специализированный адаптер (в рамках JDO это должна быть реализация интерфейса `javax.jdo.PersistenceManager`). Основная функция этого адаптера — прозрачная для разработчиков синхронизация хранилища данных и набора хранимых объектов в памяти приложения. Он должен также обеспечивать достаточно высокую производительность приложения, несмотря на наличие нескольких промежуточных слоев между классами самого приложения и хранилищем данных, представляемых ими.

Использование JDO очень похоже на использование Hibernate. Конфигурационные файлы, хранящие информацию о привязке объектов Java классов к записям в определенных таблицах, а также о привязке коллекций ссылок на объекты к ссылкам между записями, также похожи на аналогичные файлы Hibernate. Обычно первые даже несколько проще, поскольку б***о***льшую часть работы по отображению полей объектов в поля записей базы данных берет на себя специализированный адаптер.

В дополнение JDO предоставляет средства для построения запросов с помощью описания свойств объектов, без использования более привычного встроенного SQL.

В целом, подход JDO является обобщением подхода ORM на произвольные хранилища данных, но он требует реализации более сложных специализированных адаптеров для каждого вида таких хранилищ, в то время как один и тот же ORM может использоваться для разных реляционных СУБД, требуя для своей работы только наличия более простого драйвера JDBC.

Enterprise Java Beans 3.0

В рамках следующей, пятой версии платформы J2EE [6] будет использоваться новый набор техник для построения компонентов EJB — стандарт EJB 3.0 [12].

Стандарт EJB версии 3.0 является существенным развитием EJB 2.1 в сторону упрощения разработки и поддержки приложений, использующих технологию EJB. При этом большинство нововведений основывается на тех же идеях, на которых строится работа с хранимыми данными в рамках Hibernate и JDO. Гевин Кинг (Gavin King), создатель первых версий Hibernate, является и одним из разработчиков стандарта EJB 3.0.

В рамках нового стандарта для создания компонента требуется описать только один класс, который не должен наследовать какому-то библиотечному классу или реализовывать какой бы то ни было интерфейс. Объекты этого класса могут быть как хранимыми, так и временными.

Различные описатели свойств компонента и его методов можно оформлять в виде аннотаций, стандартной конструкции для описания метаданных в Java 5. Примерами таких описателей являются: указание вида компонента — компонент данных или сеансовый; отображение свойств класса в поля таблиц и ссылки между ними; отметки нехранимых свойств и полей; отметки специальных методов, вызываемых при переходе между этапами жизненного цикла компонента; транзакционные атрибуты методов и пр. Заметим, что в .NET использование для этого аналогов аннотаций, атрибутов, уже реализовано. Остается и возможность использовать для этих целей XML-дескрипторы, аналогичные используемым в Hibernate конфигурационным файлам. Конфигурационные файлы более удобны для сложных приложений, поскольку сокращают количество мест, в которые нужно вносить модификации при изменениях в структуре базы данных или объектной модели данных приложения. В то же время применение аннотаций позволяет обеспечить более быструю и удобную разработку небольших приложений.

Среда Spring

Среда Spring [9,13-15] представляет собой одну из наиболее технологичных на данный момент сред разработки Web-приложений. В ее рамках получили дальнейшее развитие идеи специализации обработчиков запросов, использованные в Struts. В Spring также используются

элементы аспектно-ориентированного подхода к разработке ПО, позволяющие значительно повысить гибкость и удобство модификации построенных на ее основе Web-приложений.

Основная задача, на решение которой нацелена среда Spring, — интеграция разнородных механизмов, используемых при разработке компонентных Web-приложений. В качестве двух основных средств такой интеграции используются идеи *обращения управления (inversion of control)* и *аспектно-ориентированного программирования (aspect-oriented programming, AOP)*.

Обращением управления называют отсутствие обращений из кода компонентов приложения к какому-либо API, предоставляемому средой и ее библиотеками. Вместо этого компоненты приложения реализуют только функции, необходимые для работы в рамках предметной области и решения тех задач, с которыми приложению придется иметь дело. Построение из этих компонентов готового приложения, конфигурация отдельных его элементов и связей между ними — это дело среды, которая сама в определенные моменты обращается к нужным операциям компонентов. Конфигурация компонентов приложения в среде Spring осуществляется при помощи XML-файла springapp-servlet.xml. Этот файл содержит описание набора компонентов, которые настраиваются при помощи указания параметров инициализации соответствующих классов и значений своих свойств в смысле JavaBeans.

Другое название механизма обращения управления — *встраивание зависимостей (dependency injection)* — связано с возможностью указывать в коде приложения только интерфейсы некоторых объектов, а их точный класс, как и способ их получения (создание нового объекта, поиск при помощи службы каталогов, обращение к фабрике объектов и пр.) — описывать только в конфигурационном файле. Такой механизм позволяет разделить использование объектов и их конфигурацию и менять их независимо друг от друга.

Аналогичный механизм предполагается использовать и в рамках EJB 3.0 в следующем виде. При помощи конструкции `@Resource Type object;` некоторый объект может быть объявлен в коде как подлежащий отдельной конфигурации, а в конфигурационном файле для объекта с именем `object` указывается его точный тип и способ инициализации.

Аспектно-ориентированный подход к программированию основыван на выделении *аспектов* — отдельных задач, решаемых приложением — таким образом, чтобы их решение можно было организовать в виде выполнения определенных действий каждый раз, когда выполняется определенный элемент кода в ходе работы программы. При этом действия в рамках данного аспекта не должны зависеть от других аспектов и остальных действий, выполняемых программой. Не все задачи могут быть представлены как аспекты. Поэтому AOP-программа представляет собой композицию из «обычной» программы, описываемой вне рамок AOP, и аспектных действий, выполняемых в определенных точках «обычной» программы. Такие действия называют *указаниями (advice)*, точки их выполнения в «обычной» программе — *точками вставки (joinpoints)*, а наборы точек вставки, в которых должно выполняться одно и то же действие — *сечениями (pointcuts)*.

Примером указаний могут служить трассировка параметров вызова метода или проверка корректности инициализации полей объекта. В качестве примеров точек вставки можно привести момент перед вызовом определенного метода или сразу после такого вызова, момент после инициализации определенного объекта или перед его уничтожением. Сечения могут описываться условиями типа «перед вызовом в данном объекте метода, чье имя начинается на `"get"`» или «перед уничтожением объекта класса `A` со значением поля `value`, превышающим 0».

Spring дает возможность определять сечения и указания, выполняемые в них, в конфигурационных файлах приложения.

При помощи AOP в Spring реализована поддержка интеграции приложений с хранилищами данных при помощи различных технологий, примерами которых являются JDO и Hibernate. Для того чтобы использовать любую такую технологию в приложении на основе Spring, достаточно иметь специализированный адаптер, который реализует интерфейс, предлагаемый Spring для поддержки синхронизации данных между приложением и хранилищем данных. После указания этого адаптера в конфигурации приложения среда сама позаботится о том, чтобы всякий раз после

появления возможных различий между данными приложения и базы данных были вызваны методы этого адаптера, копирующие новые данные в ту или другую сторону.

Похожим образом поддерживается интеграция с различными реализациями служб поддержки транзакций и декларативное управление транзакциями. Методам обычного класса Java в конфигурационном файле (или с помощью аннотаций Java 5) можно приписать определенные транзакционные атрибуты, а также набор типов исключительных ситуаций, вызывающих откат транзакции. Для сравнения — в EJB 2.1 только исключения, чей тип является наследником `java.lang.RuntimeException`, `java.lang.Error` или `javax.ejb.EJBException`, вызывают автоматический откат транзакции. Адаптер конкретной реализации службы транзакций также указывается в конфигурации приложения.

Использование обращения управления позволяет также упростить описание конфигурации сервлетов и *контроллеров* (Spring-аналог действий из Struts) и определение самих контроллеров.

Ajax

Рассказывая о развитии технологий разработки Web-приложений, невозможно обойти вниманием набор техник, известный под названием Ajax [16,17] и используемый для снижения времени реакции Web-интерфейсов на действия пользователя.

Вообще говоря, Web-технологии не очень хорошо приспособлены для построения пользовательского интерфейса интерактивных приложений, т.е. таких, где пользователь достаточно часто выполняет какие-то действия, на которые приложение должно реагировать. Они изначально разрабатывались для предоставления доступа к статической информации, которая меняется редко и представлена в виде набора HTML-страниц. Обычно при обмене данными между Web-клиентом и Web-сервером клиент изредка посылает серверу простые и небольшие по объему запросы, а тот в ответ может присылать достаточно объемные документы.

В интерактивных приложениях обмен данными между интерфейсными элементами приложения и обработчиками запросов несколько иной. Обработчик достаточно часто получает запросы и небольшие наборы их параметров, а изменения, которые происходят в интерфейсе после получения ответа на запрос, обычно тоже невелики. Часто нужно изменить содержание лишь части показываемой браузером страницы, в то время как остальные ее элементы представляют более стабильную информацию, являющуюся элементом дизайна сайта или набором пунктов его меню. Для отражения этих изменений не обязательно пересылать с сервера весь HTML-документ, что предполагается в рамках традиционного обмена информацией с помощью Web. Точно так же, если бы корректность вводимых пользователем данных можно было бы проверить на стороне клиента, обработка некорректного ввода происходила бы гораздо быстрее и не требовала бы вообще никакого обмена данными с сервером.

Ajax пытается решить эти задачи при помощи комбинации кода на JavaScript, выполняющегося в браузере, и передаваемых время от времени между клиентом и сервером специальных XML-сообщений, содержащих только существенную информацию о запросе или изменениях HTML-страницы, которые должны быть показаны. В рамках браузера в отдельном потоке работает ядро Ajax, которое получает сообщения JavaScript-кода о выполнении пользователем определенных действий, выполняет проверку их корректности, преобразует их в посылку соответствующего запроса на сервер, преобразует ответ сервера в новую страницу или же выдает уже имеющийся в специальном кэше ответ. Запросы на сервер и их обработка осуществляются часто асинхронно с действиями пользователя, позволяя заметно снизить ощущаемое время реакции системы на них.

На настоящий момент Ajax еще не является полноценным элементом компонентных технологий. Используемые в его рамках техники имеют некоторые проблемы, связанные с переносимостью между различными платформами, а также не вполне четко разграничивают ответственность между отдельными подсистемами приложения. Однако, в дальнейшем, аналогичные техники наверняка будут включены в стандартные платформы для разработки Web-приложений — J2EE и .NET.

Web-службы

В настоящее время совместно с компонентными технологиями на базе J2EE и .NET широкое распространение получают *Web-службы* или *Web-сервисы (Web services)* [18-22], представляющие собой компонентную технологию другого рода, реализуемую поверх этих платформ. Основными отличительными признаками ***служб (services)*** любого вида служат следующие.

- Служба чаще всего предоставляет некоторую функцию, которая полезна достаточно широкому кругу пользователей.
- Как и другие компоненты, службы имеют четко описанный интерфейс, выполняющий определенный контракт. Именно контракт фиксирует выполняемую службой функцию.
- Контракт службы не должен зависеть от платформ, операционных систем и языков программирования, с помощью которых эта служба может быть реализована. Интерфейс и реализация службы строго отделяются друг от друга.
- Службы вызываются асинхронно. Ждать или нет результатов вызова службы, решает сам клиент, обратившийся к ней.
- Службы совершенно независимы друг от друга, могут быть вызваны в произвольных комбинациях и всякий раз работают в соответствии со своим контрактом.
  Контракт службы не должен зависеть от истории обращений к ней или к другим службам. Но он может зависеть, например, от состояния каких-либо хранилищ данных.
- Обращение к службе возможно с любой платформы, из любого языка программирования, с любой машины, имеющей какую-либо связь с той, на которой размещается реализация службы. Реализации служб должны предоставлять возможность обратиться к ним и динамически обнаружить их в сети, а также предоставлять необходимую дополнительную информацию, с помощью которой можно понять, как с ними работать.

Архитектура приложений, построенных из компонентов, являющихся такими службами, называется ***архитектурой, основанной на службах (service oriented architecture, SOA)***.

Практически единственным широко используемым видом служб являются Web-службы. ***Web-службами*** называют службы, построенные с помощью ряда определенных стандартных протоколов, т.е. использующие протокол SOAP и инфраструктуру Интернет для передачи данных о вызовах операций и их результатах, язык WSDL для описания интерфейсов служб и реестры UDDI для регистрации служб, имеющихся в сети. Существуют многочисленные дополнительные технологии, протоколы и стандарты, относящиеся к другим аспектам работы Web-служб, однако они пока применяются гораздо реже. Web-службы могут быть реализованы на основе J2EE, .NET или других технологий.

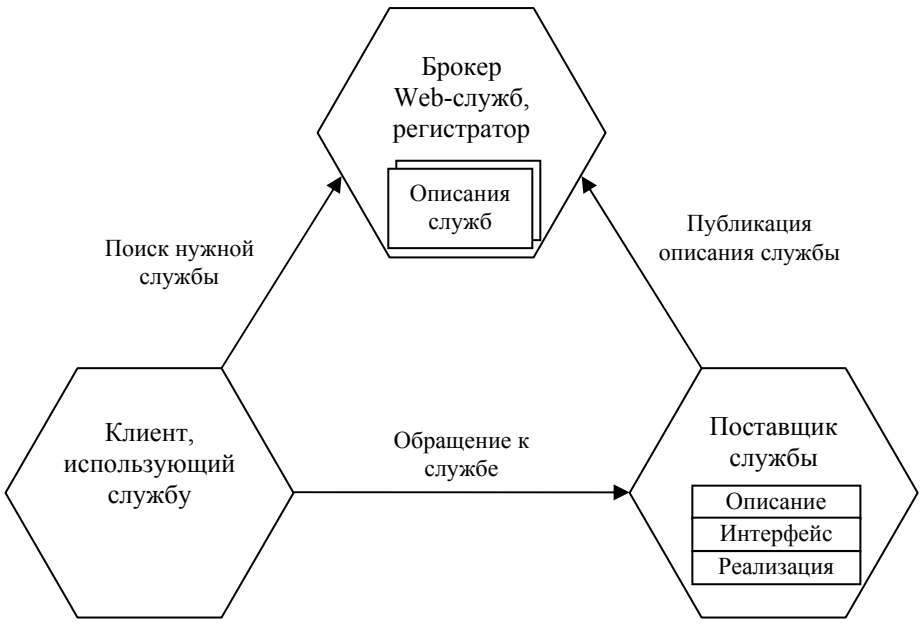


Рисунок 81. Схема архитектуры приложений на основе Web-служб.

Основное назначение Web-служб — предоставление бизнес-услуг организациями отдельным пользователям, а также другим организациями на единой технологической основе. При взаимодействии между различными организациями (business-to-business, B2B) Web-службы делают возможным оказание всех необходимых услуг вообще без вмешательства человека, если, конечно, урегулированы все вопросы, касающиеся стоимости услуг, защищенности данных об оказанных услугах, и доверия организаций друг к другу.

Таким образом, Web-службы представляют собой компонентную технологию построения крупномасштабных распределенных приложений, ориентированных на предоставление бизнес-услуг. В таких приложениях могут участвовать тысячи Web-служб, работающих на совершенно разных платформах, в различных организациях и реализованных с использованием разных технологий. В каждый момент времени такое приложение может не существовать как единое целое: какие-то его компоненты могут не участвовать в текущей работе, другие вообще не работать в связи с отключением соответствующих серверов, третьи — перемещаться с одной машины на другую. Все его компоненты объединяются только едиными стандартами описания интерфейсов и обеспечения взаимодействия между Web-службами.

Схема архитектуры приложений на основе Web-служб, предложенная IBM в конце 1990-х годов, изображена на Рис. 81.

После создания новой службы ее поставщик — организация или частное лицо, которое предоставляет ее, — регистрирует службу у брокера Web-служб (или у нескольких таких брокеров), помещая в его реестр описание службы. Такое описание содержит описание услуги, предоставляемой Web-службой и описание способа доступа к ней — протокол доступа, адрес и порт, к которому надо обращаться.

Клиент, которому понадобилась некоторая услуга, обращается к брокеру Web-служб. Найдя службу, которая, судя по описанию, эту услугу оказывает, он обращается по указанному в ее спецификации адресу и получает описание интерфейса для обращения к этой службе. После этого клиент может обращаться к службе в соответствии с этим интерфейсом.

Использование на всех этапах описанного процесса стандартных описаний в форматах, основанных на XML, позволяет полностью автоматизировать его.

Описание интерфейса Web-служб

Языком описания интерфейса Web-служб служит *WSDL* (читается «виздэл») — *Web Services Description Language, язык описания Web-служб* [23]. Этот язык служит аналогом (и некоторым обобщением) языков описания интерфейсов (IDL), используемых при реализации удаленных вызовов процедур и методов. В настоящее время используется версия WSDL 1.1, но в 2006 году выйдет версия 2.0, в которой достаточно много новых элементов.

Описание интерфейса работы с Web-службой на WSDL состоит из двух частей — абстрактной и конкретной. Абстрактная часть описания содержит определения типов данных, используемых в обращениях к данной службе и определения абстрактных операций, которые можно «вызвать» у службы. Напомним, что все такие «вызовы» являются асинхронными обращениями. Конкретная часть содержит описание привязки операций к определенным адресам, протоколам доступа и портам.

- Типы данных описываются внутри тега `<types>`. Они могут основываться на встроенных XML-типах и использовать XML Schema для описания сложных структур данных.
- С помощью тегов `<message>` описываются типы сообщений, которыми стороны могут обмениваться в ходе работы службы. Для сообщения указывается, является ли оно входящим или исходящим, а его описывается структура в терминах определенных ранее типов данных.
- Далее определяются операции, которые могут включать в себя обмен сообщениями нескольких типов. Для операции указывается используемый в ее рамках шаблон обмена сообщениями. Примерами шаблонов являются: однократное уведомление со стороны

службы, запрос со стороны клиента, запрос-ответ. Всего в WSDL 1.1 есть 4 вида шаблонов, в WSDL 2.0 — уже 9 видов.

- Операции группируются в интерфейсы, которые в WSDL 1.1 названы *типами портов (port types)*.
- С помощью элемента `<binding>` определяется привязка интерфейсов к их реализациям. Она задает конкретные форматы сообщений и протоколы их посылки/получения для некоторого интерфейса. Один интерфейс может иметь несколько привязок.
- Элемент `<port>` определяет порт, задающий конкретные адрес и порт некоторой привязки, а также, возможно, транспортный протокол для передачи сообщений на этот адрес.
- Наконец, элемент `<service>` описывает службу целиком, указывая набор портов для доступа к различным ее интерфейсам.

Связь

Связь между Web-службами и их клиентами осуществляется по протоколу *SOAP (Simple Object Access Protocol, простой протокол доступа к объектам)* [24]. Протокол SOAP является протоколом уровня представления по модели OSI, т.е. он определяет формат сообщений, которые пересылаются с помощью некоторого транспортного протокола, в качестве которого обычно используются HTTP, HTTPS, TCP, иногда SMTP.

Формат сообщений SOAP основан на XML. SOAP-сообщение состоит из следующих частей.

- Конверт (envelope) — содержит сообщение целиком.
- Заголовок (header) — содержит значения некоторых дополнительных атрибутов сообщения, используемых при его обработке или переадресации. Заголовок может отсутствовать и используется обычно для передачи информации о координации нескольких сообщений, идентификации сеансов и передачи разного рода сертификатов для защиты информации.
- Тело (body) — основное содержимое сообщения, должно относится к одному из типов сообщений, которыми можно обмениваться с данной службой согласно описанию ее интерфейса. Должно быть в любом сообщении.

Простой пример SOAP-сообщения приведен ниже.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Header>
    <t:Transaction
      xmlns:t="http://company.com/soap-headers/attrs"
      SOAP-ENV:mustUnderstand="1">
      5
    </t:Transaction>
  </SOAP-ENV:Header>

  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="http://company.com/web-services/trading">
      <symbol>DEF</symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Кроме определения формата сообщений, протокол SOAP определяет процесс их обработки различными посредниками и получателями.

Именование

Роль служб именования и каталогов в приложениях на основе Web-служб играют реестры Web-служб, организованные в соответствии со стандартом UDDI (Universal Description, Discovery and Integration, универсальный стандарт описания, поиска и интеграции) [25].

Существует всего лишь несколько универсальных реестров, регистрирующих любые доступные в Интернет Web-службы. Каждый из них поддерживается одной из крупных компаний, играющих заметную роль в развитии технологий разработки Web-служб. Такие реестры есть у IBM, Microsoft, SAP, NTT. К сожалению, они содержат не очень много записей о работающих Web-службах. Гораздо больше специализированных реестров, предназначенных для использования в рамках одной организации или компанией и ее партнерами.

UDDI описывает структуру реестров Web-служб. Каждая запись в таком реестре является XML-документом. Наиболее важная информация содержится в документах следующих видов.

- `businessEntity`. Такой документ описывает организацию (или лицо), предоставляющую набор Web-служб. В частности, он содержит название (имя), адрес, контакты, профиль, т.е. характеристику области ее (его) деятельности.
- `businessService`. Это список Web-служб, предоставляемых некоторой организацией.
- `bindingTemplate`. Описывает технические аспекты предоставляемых служб, в частности, адреса, к которым нужно обращаться, списки дополнительных описаний (`tModels`).
- `tModel` (technical model). Содержит дополнительную информацию о службе, в частности, предоставляемые ею услуги, условия и ограничения ее использования, предоставляемые гарантии и пр.

Помимо структуры реестра, UDDI определяет интерфейс для работы с ним, позволяющий публиковать или удалять информацию о предоставляемых службах, изменять собственника служб, искать нужные службы по набору характеристик и т.д.

Процессы

Поскольку Web-службы считаются совершенно независимыми от реализации компонентами, а управление процессами и потоками сильно зависит от платформы, в контексте Web-служб они не рассматриваются. Можно считать, что каждый экземпляр Web-службы работает в своем отдельном процессе, к которому не имеют доступа все остальные экземпляры других Web-служб или той же самой службы.

Синхронизация и целостность

Базовые и общепризнанные стандарты построения Web-служб (WSDL, SOAP и UDDI) не рассматривают вопросы синхронизации работы нескольких Web-служб. В то же время, эти вопросы очень важны при построении одних служб на базе других и разработке приложений из наборов взаимодействующих Web-служб.

Одной из попыток стандартизации протоколов совместной работы Web-служб является технология *WS-Coordination (Web Services Coordination)* [18,26]. Она предлагает набор протоколов, языков и инфраструктуру их использования, совместно позволяющих описывать и осуществлять синхронизацию и координацию нескольких Web-служб, которые работают над одной задачей.

Для обеспечения целостности при совместной работе нескольких служб могут использоваться технологии на основе стандартов *WS-Transactions* и *WS-BusinessActivity* [18,26], построенных на базе WS-Coordination.

Задачи синхронизации могут решаться с помощью средств, помогающих строить приложения на основе композиции Web-служб или при помощи их «оркестровки» (web services orchestration) [18]. Одним из таких подходов является *BPEL (Business Process Execution Language, язык исполнения бизнес-процессов)* [18,27]. Это графический язык, дающий возможность описать достаточно сложные потоки работ, каждая из которых выполняется отдельной службой, и скомпилировать такое описание в реализацию новой Web-службы.

Отказоустойчивость

Возможность обеспечения отказоустойчивости Web-служб заложена в архитектуру приложений на их основе. Ее можно добиться дублированием их реализаций и регистрацией нескольких точек доступа к службам, реализующим один и тот же интерфейс.

Для обеспечения отказоустойчивости при передаче сообщений разрабатывается дополнительный стандарт *WS-Reliability* [28], расширяющий SOAP. Использование WS-Reliability позволяет гарантировать доставку сообщений, используемых в работе Web-служб.

Защита

Наиболее вероятным кандидатом на место широко используемого стандарта защиты информации, передаваемой в сообщениях при работе с Web-службами, является стандарт *WS-Security* [18,29].

Он расширяет SOAP, добавляя в заголовки сообщений этого протокола информацию, с помощью которой можно подтвердить целостность сообщения, подтвердить личность отправителя или затруднить доступ к его содержанию для третьих партий, определив алгоритм шифрования содержимого.

Литература к Лекции 15

[1] Java Platform Enterprise Edition Specifications, version 5. Доступны через http://java.sun.com/j2ee/5.0/index.jsp.
[2] Web-сайт проекта Apache Struts http://struts.apache.org/.
[3] C. Cavaness. Programming Jakarta Struts. O'Reilly, 2002.
[4] Java Server Faces Specification, version 1.2. Доступна на http://java.sun.com/j2ee/javaserverfaces/download.html.
[5] H. Bergsten. JavaServer Faces. O'Reilly, 2004.
[6] Спецификации J2EE 5.0. Доступны через http://java.sun.com/javaee/5/javatech.html.
[7] Web-сайт проекта Hibernate http://www.hibernate.org/.
[8] C. Bauer, G. King. Hibernate in Action. Manning, 2004.
[9] B. A. Tate, J. Gehtland. Better, Faster, Lighter Java. O'Reilly, 2004.
[10] Web-сайт технологии JDO http://jdocentral.com/.
[11] D. Jordan, C. Russell. Java Data Objects. O'Reilly, 2003.
[12] Спецификации Enterprise Java Beans 3.0. Доступны через http://java.sun.com/products/ejb/docs.html.
[13] Web-сайт проекта Spring http://www.springframework.org/.
[14] R. Johnson. Expert One-on-One J2EE Design and Development. Wrox, 2002.
[15] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, C. Sampaleanu. Professional Java Development with the Spring Framework. Wiley, 2005.
[16] http://developer.mozilla.org/en/docs/Category:AJAX:Articles.
[17] D. Crane, E. Pascarello, D. James. Ajax in Action. Manning, 2005.
[18] G. Alonso, F. Casati, H. Kuno, V. Machiraju. Web Services. Concepts, Architectures and Applications. Springer-Verlag, 2004.
Сайт этой книги http://www.inf.ethz.ch/personal/alonso/WebServicesBook.
[19] Сайт IBM, посвященный Web-службам и SOA. http://www-128.ibm.com/developerworks/webservices/.
[20] Э. Ньюкомер. Веб-сервисы. Для профессионалов. СПб.: Питер, 2003.
[21] Х. Дейтел, П. Дейтел, С. Сантри. Технологии программирования на Java 2. Книга 3: Корпоративные системы, сервлеты, JSP, Web-сервисы. М.: Бином, 2003.
[22] А. Феррара, М. Мак-Дональд. Программирование web-сервисов для .NET. СПб.: Питер-BHV, 2003.
[23] Спецификации WSDL 1.1 http://www.w3.org/TR/wsdl.
[24] Спецификации SOAP 1.2 http://www.w3.org/TR/soap/.
[25] Web-сайт стандарта OASIS UDDI http://www.uddi.org/.
[26] Спецификации WS-Coordination, WS-Transactions и WS-BusinessActivity. Доступны через http://www-128.ibm.com/developerworks/library/specification/ws-tx/.
[27] Спецификации BPEL. Доступны через http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.

[28] Спецификации WS-Reliability. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm.
[29] Спецификации WS-Security. http://www-128.ibm.com/developerworks/webservices/library/ws-secure/.

Лекция 16. Управление разработкой ПО

Аннотация

Рассматриваются основные деятельности, входящие в компетенцию руководителей проектов. В общем рассказе о некоторых аспектах управления ресурсами, персоналом, рисками и коммуникациями проекта выделены особенности управления проектами по созданию ПО.

Ключевые слова

Организационная культура, структура организации, заинтересованные лица, спонсор, менеджер, заказчик, пользователь, команда проекта, цели проекта, содержание проекта, ресурсы, иерархическая структура работ, метрики сложности ПО, COCOMO II, сетевая диаграмма, PERT-диаграмма, диаграмма Ганта, критический путь, мотивация персонала, сплоченная команда, управление рисками, разрешение конфликтов, ведение переговоров.

Текст лекции

Задачи управления проектами

Эта лекция посвящена управлению проектами по разработке, поддержке или модификации программного обеспечения. С достаточно общих позиций можно считать задачей управления проектами эффективное использование ресурсов для достижения нужных результатов. Всегда нужно получить как можно более хорошие результаты, используя при этом как можно меньше ресурсов. При этом в первую очередь возникают два вопроса: что именно считается «хорошим» результатом проекта и чем, какими ресурсами, можно пользоваться для его достижения.

Чтобы ответить на вопрос о том, какими критериями руководствуются при оценке проектов, и чего нужно добиваться, надо рассмотреть его с разных аспектов.

Одним из самых важных критериев является *экономическая эффективность проекта*, т.е. отношение суммы доходов разного рода, полученных в его результате, ко всем затраченным ресурсам. К сожалению, эти доходы чаще всего невозможно определить заранее. Поэтому при оценках проекта вместо дохода от его результатов рассматривают качество создаваемого ПО во всех его аспектах, т.е. набор имеющихся функций, надежность работы, производительность, удобство для всех категорий пользователей, а также удобство расширения и внесения изменений. Характеристики качества должно соотноситься с требованиями рынка или заказчика и с уже имеющимися продуктами.

С более общих позиций и экономические, и неэкономические показатели результативности проекта объединяют в понятие ***ценностей***, создаваемых в его ходе. Эти ценности могут иметь различную природу в зависимости от проекта, от организации, в рамках которой он проводится, от национально-культурных особенностей рынка и персонала и пр. Кроме того, ценности выстраиваются в иерархию в зависимости от уровней рассмотрения проектов.

- В одном конкретном проекте основной ценностью может быть достижение запланированного качества результатов в указанный срок и в рамках определенного бюджета.
  В то же время, могут быть получены и другие ценности: достигнута высокая сплоченность команды; новые члены коллектива приобретут серьезные знания и полезные навыки; команда овладеет новыми технологиями; ее члены получат повышение и/или поощрения, которые повысят их лояльность компании и т.п.
- На уровне нескольких зависящих друг от друга проектов (такую группу проектов называют *программой*), в ходе которых создаются и дорабатываются несколько продуктов на единой платформе, а также могут оказываться различные услуги, связанные с этими продуктами, ценности связаны, прежде всего, с качеством общей архитектуры создаваемых продуктов. В одном проекте работа иногда ведется по принципу «сдали-и-забыли», т.е. основные усилия направлены на то, чтобы заказчик подписал акт приемки работ или его аналог, после чего поставщик перестает отвечать за результаты, поэтому часто такой аспект

качества ПО, как удобство внесения изменений, игнорируется. Однако для бизнеса организации в целом проведение таких проектов небезопасно. Среди исследователей и экспертов-практиков преобладает взгляд на любую программную систему как на систему *развивающуюся*, полезность которой значительно снижается, если нет возможности расширять ее, тем более — исправлять серьезные ошибки, которые всегда есть в сложных программах. Заказчик всегда сталкивается с проблемами поддержки ПО и рано или поздно столкнется и с необходимостью его развития. На уровне группы проектов игнорирование удобства модификации ПО, а также вопросов, связанных с организационными и экономическими последствиями изменений в общей архитектуре, просто губительно.

- На уровне организации в целом или подразделения, в рамках которого может одновременно проводиться много проектов, связанных по предметной области, используемым технологиям и просто по вовлеченным в них людям, возникают другие ценности. Это может быть отлаженность производственных процессов, высокая технологическая экспертиза и технологическое лидерство в своей области, низкая текучка кадров, повышение оборота, прибыли, капитализации, доли продаж в рамках отрасли, занимаемого среди поставщиков такой же продукции места по экономическим и технологическим показателям.

Поскольку каждый проект проводится в рамках какой-то организации, то принятая в ней система ценностей влияет и на оценку каждого конкретного проекта (см. далее).

Основные виды ресурсов, используемых в любом проекте, следующие.

- **Время.**
  Этот ресурс всегда жестко ограничен. Продолжительность проекта фиксирована, это одно из главных отличий проектов от обычной операционной деятельности, которой нужно заниматься неопределенной время. Чаще всего эти ограничения определяются интересами заказчика, выраженными в контракте, или решением руководства, основанном на анализе рынка и информации о действиях конкурентов.
  Более того, даже при попытке создать «вялотекущий» проект без четкой установки его временных рамок (иногда при помощи такого приема руководство организации пытается выполнить нужные внутренние работы, не выделяя на них достаточных ресурсов), руководитель проекта должен настаивать на их определении. Иначе проекта как такового просто не получится — эффективное использование времени играет очень важную роль в успешности достижения необходимых результатов.
- **Бюджет.**
  Бюджет проекта тоже всегда ограничен — деньги, как известно, лишними не бывают. Деньги часто рассматриваются как практически универсальный эквивалент других ресурсов: за счет вложения дополнительных денег пытаются выиграть во времени, привлечь дополнительный персонал и пр. Однако полностью в деньги можно перевести только используемое оборудование и материалы, да и то с некоторыми потерями во времени на их приобретение и подготовку к работе.
  Вместе со временем бюджет задает основные ограничения на содержание и возможные результаты проекта.
- **Персонал.**
  Персонал иногда рассматривается как возобновляемый ресурс, имеющий денежный эквивалент («наймем проектировщика за 1500 у.е. в месяц»). Однако чаще всего люди ведут себя не совсем так, как оборудование или мебель, — они не позволяют себя «передвигать», «убирать» и «добавлять» с такой же легкостью. Имея определенный персонал, нельзя получить нужный результат с помощью заранее известной последовательности действий. Даже для получения одних и тех же результатов от одного и того же человека в разных обстоятельствах требуется применять различные подходы. Обычно лишь работников среднего уровня можно найти в нужном количестве за среднюю зарплату, а чтобы нанять высокопрофессионального ответственного члена проектной команды (руководителя, бизнес-аналитика, архитектора, специалиста по проектированию

интерфейсов и пр.), часто нужно пойти на достаточно высокие затраты. Или же придется идти на аренду такого специалиста, что стоит еще дороже. С другой стороны, человека, который со временем вырастет до такого уровня, можно нанять практически случайно. Большое значение для успеха проекта также имеет построение из отдельных людей настоящей команды, которая дала бы больший результат, чем сумма результатов отдельной работы ее членов. Зачастую действительно сложное ПО без такой команды в рамках выделенного бюджета создать невозможно.

- **Используемое оборудование, инструменты, материалы, и т.п.**
  Это ресурсы классического типа, которые могут использоваться однократно или многократно, могут изнашиваться или не изнашиваться в результате использования (а также просто со временем) и достаточно адекватно могут быть обменены на деньги.

Дальнейшее содержание лекции посвящено общим вопросам управления проектами и отдельным деятельностям, которым менеджер проекта должен уделять внимание в его ходе. Рассматриваемые подходы в основном применимы к проектам в любой области, а при управлении разработкой ПО нужно учитывать следующие ее особенности.

- Создаваемые программы нематериальны. Это порождает проблемы двух видов.
  - Программы обладают потрясающей гибкостью, они не оказывают сопротивления воздействиям, как физические материалы. Все знают, что построить дом можно из твердого и плотного материала, а чтобы сделать батут нужно использовать материал мягкий, гибкий и прочный. В мире же программ можно построить все, что угодно, из одних и тех же базовых конструкций.
    Поэтому иногда кажется, что раз суть требуемых изменений в программе понятна, на их воплощение нужно немного усилий. Это не так. Работа с элементами программ в этом аспекте не слишком отличается от работы с кирпичами и строительными блоками. А если эти блоки еще и стоят кое-как, то при попытке передвинуть их программиста вообще может «завалить» — отладка полученной программы потребует колоссальных усилий.
  - Движение к нужному результату при разработке ПО очень тяжело проконтролировать. При возведении здания или постройке корабля можно непосредственно наблюдать за тем, как продвигается работа. При создании сложной программной системы силами многих разработчиков нужно аккуратно подбирать индикаторы того, как идут дела, иначе легко впасть в заблуждение относительно истинного положения вещей.
- Программные системы практически всегда уникальны.
  Каждая из них обладает своим набором характеристик (включая все реализуемые функции, производительность при их выполнении, все элементы пользовательского интерфейса и т.п), так или иначе отличающихся от характеристик других программ, даже делающих «то же самое». Если обладающая нужными свойствами (в том числе и подходящей ценой) программа уже имеется, незачем создавать ее заново — достаточно приобрести ее или взять ее код и скомпилировать. Поэтому практически каждая *разрабатываемая* программа уникальна — она должна иметь такие характеристики, которыми не обладает ни одна уже созданная.
  - Тем самым, почти каждый проект по разработке ПО включает элементы творчества, создания того, чего еще никто не делал. Крупные же проекты требуют решения сразу нескольких ранее не решенных задач. Управление проектами с элементами творческой деятельности очень сильно отличается от управления проектами, в которых заранее ясно, что именно надо делать и как.
  - Другое следствие этой уникальности ПО — отсутствие стандартных процессов разработки. Нет целостных подходов к созданию ПО, которые годились бы для всех случаев, а не только для определенного вида проектов. Кроме того, для хорошо определенных процессов, таких, как RUP, XP, Microsoft Solution Framework или DSDM (Dynamic Systems Development Method, Метод разработки динамичных систем), недостаточно четко очерчены области их применимости. Каждый раз менеджеру

проекта приходится только на основании своего опыта и советов экспертов принимать решение о том, какой процесс разработки использовать и как его модифицировать для достижения большей эффективности в конкретном проекте.

- Есть много аргументов в пользу того, что программный код является *проектом*, а не *конечным продуктом*.
  При разработке ПО переход от проекта к продукту почти полностью автоматизирован — требуется лишь скомпилировать код и развернуть систему в том окружении, где она будет работать. А само программирование гораздо больше напоминает разработку проекта здания, чем его строительство по уже готовому проекту. То же, что в разработке ПО обычно называется проектом или дизайном, представляет собой лишь набросок окончательного проекта, определяющий основные его черты и требующий дальнейшей детализации. Таким образом, разработка программ отличается от других инженерных видов деятельности тем, что в основном состоит из проектирования, а не изготовления продукта.
  Это еще одна причина того, что программирование всегда включает элемент творчества. Кроме того, проблемы, с которыми сталкивается руководитель проекта разработки ПО, гораздо более похожи на проблемы периода проектирования здания, самолета или корабля, чем на проблемы периода их постройки.

Окружение проекта

Среди аспектов окружения проекта, оказывающих на его ход существенное влияние, отметим структуру проводящей проект организации, организационную культуру различных вовлеченных в проект организаций, которую руководителю проекта надо учитывать при выработке стратегии поведения, а также заинтересованных в проекте лиц.

Структура организации-исполнителя проекта

Полномочия руководителя и ход проекта в значительной мере зависят от ***структуры организации***, в рамках которой проводится проект, т.е. от тех правил, согласно которым в этой организации группируются ресурсы и происходит выделение ресурсов под проекты. Различают следующие структуры организаций [1].

- ***Функциональная.*** В такой организации подразделения выделяются по их области деятельности или этапам производственных процессов — в ней есть финансовый, плановый, маркетинговый, опытно-конструкторский и производственный отделы. Проекты ведутся сотрудниками нескольких разных подразделений, а руководство проектом осуществляется за счет координации их деятельности, через руководителей соответствующих отделов. Руководитель проекта практически всегда член дирекции. Выделение ресурсов, необходимых проекту, должно осуществляться на уровне дирекции, которая дает поручения руководству отделов выделить соответствующую часть ресурсов. Такая схема позволяет собрать вместе сотрудников, обладающих знаниями и умениями в одной области, и развивать их экспертизу. Она помогает выполнять очень крупные проекты. С другой стороны, она не слишком гибка и предполагает высокую косвенность управления проектом и ограниченность общего количества проектов, проводимых организацией.
- ***Проектная.*** В организации такого типа подразделения выделяются для проведения конкретных проектов. Руководитель такого временного подразделения является руководителем соответствующего проекта и полностью распоряжается выделенными для него ресурсами.
  Эта схема обладает высокой гибкостью и приспособляемостью под нужды проекта, но может требовать дополнительных усилий для составления проектной команды, поскольку слабо мотивирует развитие персонала.
- ***Продуктовая.*** Подразделения такой организации отвечают за разработку, развитие и поддержку определенных продуктов или семейств близких продуктов. В каждом таком

подразделении может одновременно выполняться несколько проектов, связанных с данным продуктом. Руководителями проектов обычно являются сотрудники этого отдела, которые вполне распоряжаются выделенными для проекта ресурсами.
Продуктовая схема позволяет дополнить гибкость и простоту управления проектами в проектной схеме легкостью подбора подходящего персонала. Недостатком ее может являться выработка слишком узкой специализации у сотрудников и трудности расформирования большого подразделения при отказе от продолжения работ над некоторым продуктом.

- ***Ориентированная на клиента.*** Подразделения таких организаций формируются для удовлетворения нужд крупных клиентов или групп клиентов. Проекты для такого клиента ведутся внутри соответствующего подразделения.
  Эта схема позволяет уменьшить усилия, необходимые для понимания нужд клиентов. В целом она похожа на продуктовую, но при возникновении нужды в новом продукте может осложнить подбор персонала в соответствующую группу.
- ***Территориальная.*** Подразделения формируются согласно географическому положению. Проекты бывают локальными, целиком проводящимися в рамках одного подразделения, или распределенными — включающими ресурсы нескольких подразделений.
  В ее рамках удобнее проводить локальные проекты, а распределенные всегда требуют дополнительных усилий по координации работ.
- ***Матричная.*** Это гибрид нескольких схем, обычно проектной или продуктовой и функциональной. В такой организации есть и функциональные подразделения, в которых группируются ресурсы, и проектные группы, формируемые под конкретный проект из служащих функциональных подразделений. Ресурсы проекта передаются в соответствующую группу, и ими распоряжается руководитель проекта. Руководители функциональных подразделений, тем не менее, могут даже во время проекта иметь определенную власть над своими подчиненными.
  Эта схема может сочетать достоинства функциональной и проектной, но может и порождать проблемы, связанные с двойной подчиненностью участников проектных групп и разницей между возложенной на них ответственностью и предоставленными полномочиями.

Организационная культура

При выборе той или иной стратегии действий менеджер проекта должен учитывать и организационную культуру организации-исполнителя и других связанных с проектом организаций. ***Организационной*** или ***корпоративной культурой*** называют совокупность общих убеждений, норм, ценностей и принятых стилей поведения служащих данной организации. Выделяют следующие виды организационной культуры [2,3].

- ***Иерархическая (закрытая).*** Работа такой организации основана на формальных правилах, четко определенных полномочиях ее служащих и отношениях между ними. Развитие в ней представляется как стабильный и плавный, без изменений сложившийся структуры, рост показателей эффективности и рентабельности основных операций. Решения принимаются только формально уполномоченными на это лицами, многие решения требуют прямого вмешательства высшего руководства.
  Работа в такой организации может продвигаться, если она никак не нарушает сложившихся отношений или же патронируется руководством, имеющим полномочия менять существующий порядок в необходимых пределах. Любой новый подход закрепляется формулировкой новых правил и стандартов и повышением эффективности работы организации и ее подразделений при сохранении стабильной структуры бизнес-процессов.
- ***Рыночная (открытая).*** Деятельность этой организации ориентирована на завоевание и удержание ее места на рынке. Основные приоритеты — конкурентоспособность, хорошая репутация на рынке и совместная работа всех служащих над их достижением и поддержкой. Решения принимаются руководителями отдельных групп на основании

экспертных оценок, исходящих из указанных целей, и обеспечения измеримых результатов на пути к ним.
Работа в такой организации продвигается, если она поддерживает ее конкурентоспособность, позволяет оценить вклад отдельных лиц, стимулирует сотрудничество между работниками. Новый подход должен преподноситься в виде гибких, адаптирующихся под нужды рыночной целесообразности норм и получить одобрение экспертов.

- ***Инновационная (произвольная).*** Работа организации этого типа ориентирована на инновации, создание передовых, уникальных продуктов и услуг. Ценностями в ней служат творческая атмосфера, технологическое лидерство, внедрение новаторских подходов. Работа в ней продвигается, если поддерживает творческие инициативы вовлеченных в нее служащих, использует передовые достижения в некоторой области, способствует созданию уникальных технологий, продуктов и услуг. Нужно убедить каждого участника или «законодателей мод» в полезности ведущейся деятельности, подтвердить использование новаторских подходов и предоставить свободу творческих изменений.
- ***Семейная (синхронная).*** Такая организация обладает хорошей внутренней атмосферой, активно использует программы вовлечения сотрудников в бизнес и корпоративные обязательства перед ними. Ее ценностями являются гармоничное развитие ее служащих, их профессиональный рост и повышение навыков командной работы, традиции организации, сплоченность коллектива и моральный климат.
Работа в такой организации продвигается, если она обеспечивает профессиональный рост и развитие персонала, повышение удовольствия от работы, обеспечивает движение организации к некоторой цели, достижение которой необходимо обосновать как дополнительную ценность (например, представив как наиболее вероятный сценарий развития данной отрасли).

Заинтересованные в проекте лица

Ход проекта также существенно зависит от намерений, действий, информирования и поддержки нужд его ***участников*** или ***заинтересованных лиц (stakeholders)*** — всех лиц и организаций, имеющие связанные с проектом интересы, или тех, на ком результаты проекта как-то отразятся. Заинтересованные в проекте лица могут играть в нем следующие роли.

- ***Спонсор проекта.*** Это лицо или группа лиц, предоставляющая ресурсы для проекта или формирующая инвестиционно привлекательную ситуацию вокруг него. Очень важно еще до начала проекта определить его спонсора, поскольку именно с ним нужно решать все вопросы, касающиеся обеспечения проекта ресурсами.
- ***Менеджер проекта.*** Это лицо, ответственное за управление проектом, т.е. обеспечение наилучших его результатов в рамках заданных ограничений.
Иногда эту роль называют руководителем проекта, но в последнее время одной из функций руководителя принято считать лидерство: наличие авторитета, умение вовлекать людей в проект, создавать команду, вести людей за собой. Менеджер в классическом понимании не обязан быть лидером, он выполняет только административные обязанности.
- ***Лидер проекта.*** Это наиболее авторитетный человек в команде проекта, к чьему мнению прислушиваются, кто принимает большинство технических решений по ходу проекта. Часто лидер и менеджер проекта — одно лицо.
- ***Заказчик.*** Это лицо или организация, которые получают результаты проекта в собственность того или иного вида.
- ***Пользователи.*** Эта лица и организации, непосредственно использующие результаты проекта в своей деятельности.
- ***Организация-исполнитель.*** Это организация, в которой проводится проект и которая несет ответственность перед заказчиком за его выполнение. Такая организация может быть создана для проведения одного конкретного проекта и состоять только из его команды.

- ***Команда проекта.*** Это служащие организации-исполнителя, выполняющие работы по проекту. Менеджер проекта и лидер проекта входят в команду.
- ***Команда управления проектом.*** Это часть команды проекта, непосредственно участвующая в деятельности по управлению проектом. Сюда входят менеджер проекта и его лидер, может входить секретарь менеджера, эксперты, помогающие в принятии решений и т.д.

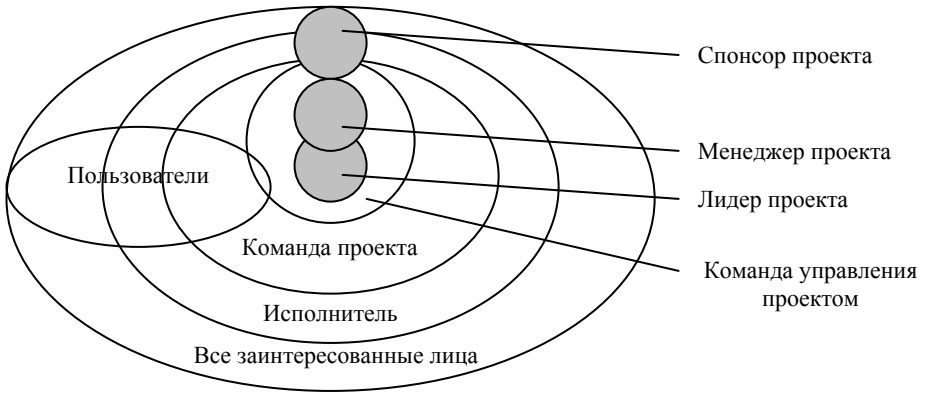


Рисунок 82. Взаимоотношения между заинтересованными лицами проекта.

- ***Источники влияния.*** Это лица, группы лиц или организации, не связанные прямо с проведением проекта и использованием его результатов, но имеющие свои интересы в том или ином развитии событий в проекте и способные повлиять на него. Это могут быть государственные и общественные организации, компании-конкуренты, средства массовой информации, отдельные лица, чью жизнь проект как-то затрагивает, служащие связанных с проектом организаций, способные повлиять на принимаемые решения, или те, чей карьерный рост может зависеть от проекта и пр.

При управлении проектом большое внимание должно уделяться потребностям и ожиданиям всех заинтересованных лиц. Обязанностью руководителя является формирование содержания проекта в таком виде, чтобы их потребности были как-то учтены.

Для этого сначала надо выявить самих заинтересованных лиц и установить их потребности и ожидания. Нужно выработать компромиссное решение (причем часто не одно, а несколько альтернативных), удовлетворяющее противоречивые интересы разных групп. Стоит уделить время тому, чтобы выяснить юридические обязательства проекта перед каждым из них, набор благоприятных возможностей или проблем, которые они могут создать, определить их возможные стратегии и стратегию поведения по отношению к каждому заинтересованному лицу и форму взаимодействия с ним, в частности, его информирования. При этом следует учитывать степень влияния каждого из заинтересованных лиц, силу и длительность воздействия, которое они могут оказать на проект, действуя ему на пользу или во вред. Обычной ошибкой является пренебрежение теми заинтересованными лицами, которые не связаны с проектом юридически, но имеют в нем свои интересы и могут серьезно влиять на его ход.

Нужно проводить переговоры, обеспечивать информационное освещение проекта, поддерживать контакты так, чтобы показывать разным лицам, как их нужды будут учтены, убеждать их действовать на благо проекта (или не действовать против него). По крайней мере, следует вовремя узнавать об их возможных шагах и планировать ответные или поддерживающие действия.

Виды деятельности, входящие в управление проектом

Виды деятельности, которыми приходится заниматься руководителю проекта или группе управления проектом для обеспечения его успешного выполнения, можно разделить на следующие области.

- *Управление содержанием проекта и качеством.*
  В эту область входят четкое определение целей проекта, его точного ***содержания*** (***project***

scope — что именно должно быть сделано в его рамках, какие результаты должны быть получены, включая все промежуточные, и какие работы должны быть проведены для этого), определение критериев качества результатов, процедур его обеспечения и контроля, выполнение этих процедур, а также критерии завершения проекта и действия по его завершению.

- *Управление ресурсами проекта.*
  Эта область включает выполнение оценок ресурсоемкости работ, включая их стоимость и продолжительность, составление графиков выполнения работ в проекте. Кроме того, практически отдельной областью является *управление персоналом проекта*, которое включает планирование использования персонала в проекте, обучение персонала и набор команды проекта, организацию командной работы и мотивацию, делегирование полномочий и управление конфликтами.
- *Управление рисками.*
  Управление проектными рисками связано с преодолением или использованием на благо проекта последствий неопределенности, всегда окружающей его, и ошибок, которые люди допускают в любой сложной деятельности. Управление рисками включает их выявление, анализ и оценку, выработку стратегии действий в отношении рисков, их отслеживание в ходе проекта и реализацию действий по их уменьшению, преодолению или использованию.
- *Управление коммуникациями и информационное обеспечение проекта.*
  Включает составление предложений по проекту, выбор продавцов, поставщиков и субподрядчиков, ведение переговоров, определение информационной политики для внешних заинтересованных лиц и внутри проектной команды, выработку способов информирования участников проекта и согласования решений, определение процедур подготовки и содержания отчетов, передачу результатов в использование.
- *Управление конфигурациями и изменениями.*
  Эта область связана с установлением процедур обработки изменений и запросов на их выполнение, включения их в содержание проекта (или удаления за его рамки). В рамках этой же деятельности должны определяться ***конфигурации*** материалов проекта — согласованные наборы определенных версий всех продуктов и документов, вырабатываемых в ходе проекта, правила выделения таких конфигураций и перехода от одной конфигурации к другой.
- *Управление проектной средой и технологиями.*
  В рамках этой деятельности определяются методы, техники, инструменты и технологии, используемые в проекте, подбираются необходимые инструменты, материалы и помещения, они подготавливаются к использованию и настраиваются под нужды проекта.
- *Контроль и мониторинг состояния проекта.*
  Эта область включает все действия, связанные с получением достоверной информации о текущем состоянии проекта, отслеживание выполнения всех планов и процедур. Полученные данные анализируются и могут приводить к выполняемым по ходу проекта действиям из других областей, направленным на предотвращение возникновения проблем и разрешение уже появившихся.

Определение любой процедуры, которое вырабатывается в ходе одной из деятельностей в рамках управления проектом, должно включать определение условий ее запуска, входных документов и материалов, действий, которые необходимо выполнить, указание возможных способов их выполнения, условий окончания и выходных документов. В ходе проекта все установленные процедуры должны выполняться при достижении условий их запуска.

Далее подробнее будут рассмотрены перечисленные области деятельности, кроме последних трех. Некоторые вопросы, относящиеся к последним трем, более техническим областям, будут освещены в рамках других областей.

Управление содержанием проекта и качеством

Управление *содержанием проекта (project scope)* является одним из критически важных для его успеха видов деятельности. Проект с нечетко определенным содержанием обречен на неудачу. Ясное же его определение — как постановка правильного вопроса — дает половину успешного ответа.

Большинство действий, связанных с определением содержания проекта, должно быть проведено уже на этапе подготовки к нему, еще до официального начала, — иначе не будет достаточных данных для принятия решения о запуске проекта.

Начинать определение содержания проекта надо с выяснения его целей. Основная цель должна описываться одной фразой и делать понятным предназначение проекта, очерчивать реальные потребности некоторых лиц и организаций, для удовлетворения которых предназначены его результаты.

После определения целей можно уточнять общие требования к продуктам или услугам, которые должны быть получены в результате проекта, основные ограничения на возможные решения, способы получения решений, включающие несколько альтернатив.

Основные элементы содержания проекта следующие.

- ***Целевые критерии проекта.***
  Эти критерии определяют ряд показателей и их значения, означающие успех проекта, а также другие значения, говорящие о необходимости его закрытия. Цели проекта обязательно должны иметь составляющие, ориентированные на клиентов и пользователей его результатов, иначе даже самый успешный внутри организации проект может привести к созданию продуктов, которые не будут использоваться.
- ***Иерархическая структура работ (work breakdown structure).***
  Это иерархическое разбиение всей работы, которую необходимо выполнить для достижения целей проекта, на более мелкие операции и действия, до такого уровня, на котором способы выполнения этих действий вполне ясны и соответствующие работы могут быть оценены и спланированы. Она включает также определение промежуточных результатов всех составляющих эту структуру работ.

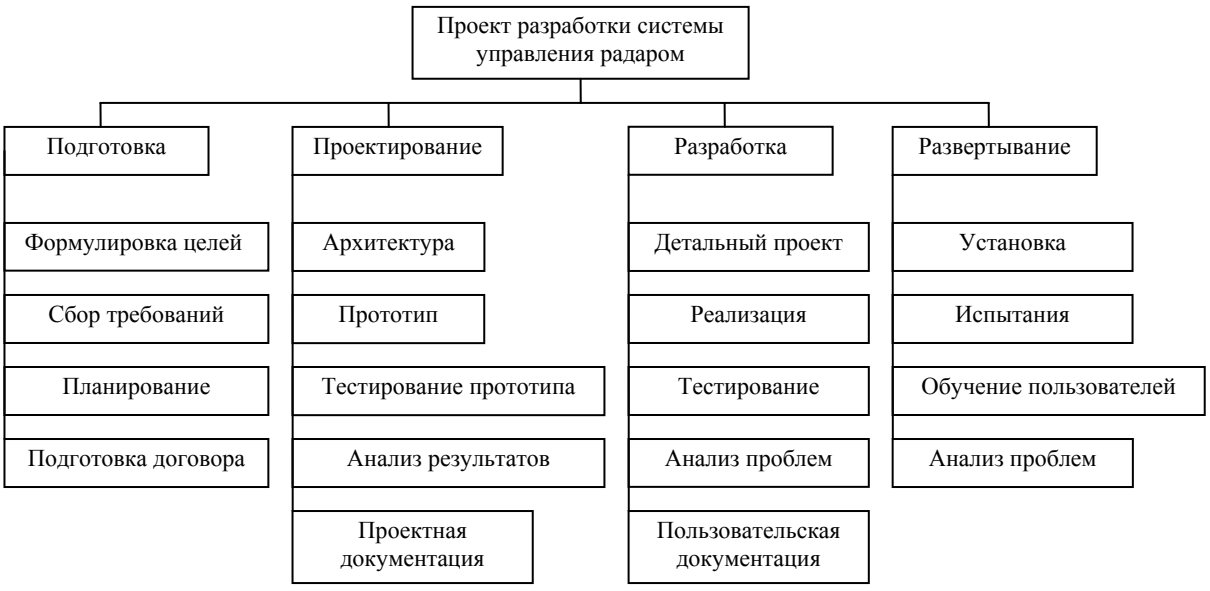


Рисунок 83. Пример структуры работ проекта, построенной на основе декомпозиции задач.

Обе составляющие содержания проекта, конечно же, требуют подтверждения у заинтересованных лиц. Только после определения этих двух элементов можно проводить обоснованное планирование работ и ресурсов для их выполнения, определение конфигураций проекта, необходимых для его выполнения инструментов и технологий и возможных рисков.

Пропуски в структуре работ выливаются в неожиданные работы и возрастание затрат как времени, так и денег на проект в целом.

При разработке структуры работ можно взять за основу набор задач, полученных декомпозицией целей и задач проекта в целом, или набор промежуточных результатов (deliverables), предаваемых заказчику, который позволит постепенно получить необходимые итоговые результаты. Пример структуры задач для гипотетического проекта разработки ПО управления радаром аэропорта, построенный первым способом, показан на Рис. 83. Аналогичный пример, построенный вторым способом, показан на Рис. 84.

Управление качеством тоже становится возможным, когда выявляются все результаты, которые нужно получить в ходе проекта, и все работы, которые необходимо выполнить. Оно включает выделение ключевых показателей качества для всех результатов, окончательных и промежуточных, а также выполняемых работ. Должны быть определены используемые метрики качества, целевые значения, которые должны быть достигнуты по этим метрикам в ходе проекта, техники, которые позволят обеспечить достижение этих показателей, способы проверки того, что они достигнуты, а также процедуры устранения обнаруженных дефектов.

Рисунок 84. Пример структуры работ проекта, построенной на основе поставляемых результатов.

Метрики ПО

Одна из базовых задач управления ресурсами — адекватная оценка затрат ресурсов на отдельные выполняемые в проекте работы. Такая оценка дает возможность планирования работ проекта и затрат на их проведение. При оценке ресурсоемкости работ основную роль играют выбранные ***метрики***, с помощью которых измеряется сложность или трудоемкость работ, а также требующееся на их выполнение время.

При выборе метрик нужно четко определить, что именно мы хотим измерить, и постараться найти измеримый показатель, как можно более близкий к этому. Типичные ошибки состоят как в том, что ничего не измеряется, так и в том, что измеряются вещи, слабо связанные с тем, что хотелось бы оценивать, лишь бы что-нибудь измерять и заносить в отчеты стройные ряды чисел.

Например, измерение времени, проводимого служащим на рабочем месте в качестве меры интенсивности его труда и вклада в процветание организации, не вполне адекватно: человек может занять себя на работе и другими вещами. Соответственно, введение административных мер по поощрению «много работающих» и наказанию «мало работающих» только на основании таких измерений может иметь «неожиданные» последствия — показатели «хорошей работы» будут расти, а реальная работа не станет выполняться быстрее или лучше.

На настоящий момент не существует достаточно адекватных и одновременно простых в использовании метрик трудоемкости разработки ПО, которые можно было бы оценивать до создания этого ПО и применять для планирования его разработки. Скорее всего, они и не будут

найдены. В этой ситуации используются разнообразные эмпирические подходы, комбинирующие простые в использовании метрики со сложными, но более адекватными.

Одна из первых идей, которая приходит в голову при оценке трудоемкости и времени разработки ПО, — оценить сначала сложность или размер программы, а потом умножить полученную оценку на производительность исполнителей. Похоже, однако, что природа разработки программ такова, что ее трудоемкость слабо связана с размерами результата (например, приводимая ниже модель COCOMO, описанная ниже, выражает эту связь в достаточно сложном виде). Часто оказывается, что оценить сразу трудоемкость по аналогии с имеющимся примерами можно точнее, чем оценив сначала размер. Тем не менее, метрики размера и сложности ПО часто используются для оценки трудоемкости разработки.

Самой простой и наиболее широко используемой метрикой является *размер программы в строках ее кода (lines of code, LOC).* Ее основное достоинство — понятность и простота вычисления. Ее недостатки — не очень хорошая адекватность в качестве метрики трудоемкости разработки программы, зависимость от используемых языков и технологий и трудность предварительной оценки размера ПО. Практика показывает, что качественная программа часто имеет несколько меньший объем, чем программа с теми же функциями, но менее удобная для сопровождения или совершающая больше ошибок. В то же время на разработку первой программы может уйти в два-три раза больше усилий.

С другой стороны, производительность разных разработчиков очень сильно отличается, но обычно руководители групп и организаций примерно представляют себе среднее значение по группе или организации. В терминах строк кода она обычно лежит в пределах от 5000 до 50000 строк хорошо отлаженного кода без учета комментариев за 1 человеко-год.

Более хитрой метрикой сложности программы являются *функциональные точки (functional points, FP)* [4,5]. Количество функциональных точек в программной системе вычисляется примерно следующим образом.

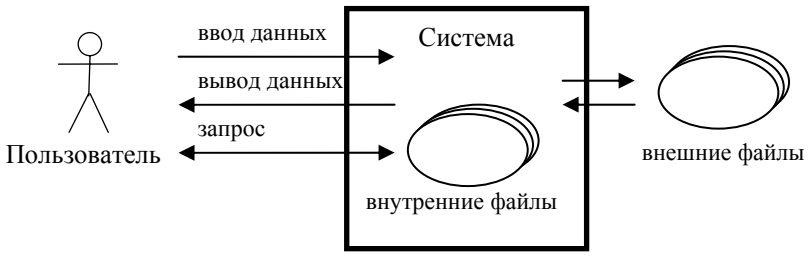


Рисунок 85. Схема рассмотрения системы при оценке ее сложности в функциональных точках.

- Выделяются обращения к системе с целью ввода данных, с целью получения каких-то уже имеющихся в ней данных (отчеты), и с запросами, в ходе которых данные вводятся в систему, перерабатываются и выдаются какие-то результаты. Дополнительно определяются группы взаимосвязанных данных (называемые файлами) внутри системы и аналогичные группы, лежащие вне ее, но используемые в ее работе.
- Для всех данных из перечисленных пяти категорий оценивается их сложность (по шкале «низкая»-«средняя»-«высокая»).
- Итоговая сложность программной системы вычисляется как сумма сложностей выявленных отдельных представителей этих пяти категорий. Сложность ввода, вывода, запроса или группы данных вычисляется умножением оценки сложности составляющих данных на весовой коэффициент, который можно найти в стандартах [4,5] или определить на основе собственного опыта. Обычно весовые коэффициенты групп данных больше, чем коэффициенты для вводов, выводов или запросов.

Количество строк кода, приходящихся на одну функциональную точку, зависит от используемых технологий и языка программирования и меняется от 300 для программирования на ассемблере до 5-10 для компонентных технологий на базе языков высокого уровня.

Другие исходные элементы используются при подсчете так называемых *объектных точек* [6]. В этом случае рассматриваются экраны, формы и отчеты, присутствующие в системе, классы, а также модули, написанные на необъектных языках. Сложность каждого из таких элементов оценивается отдельно, после чего их сложности складываются, тоже с разными весовыми коэффициентами для разных категорий элементов.

Обе эти метрики хорошо применимы к так называемым *информационным системам,* т.е. системам, основные функции которых связаны с накоплением и хранением больших объемов данных, а также с предоставлением доступа и интерактивной обработкой запросов к ней. Оценка в таких терминах компилятора, системы обмена сообщениями или автоматизированной системы навигации корабля будет менее адекватной.

Наиболее известным методом оценки трудоемкости и времени проекта, основанным на большом количестве данных из проведенных ранее проектов, является ***конструктивная модель стоимости*** версии II ***(Constructive Cost Model II, COCOMO II)*** [7-9].

В рамках этой модели оценки трудоемкости проекта и времени, требующегося на его выполнение, определяются тремя разными способами на разных этапах проекта.

- На самых ранних этапах, когда примерно известны только общие требования, а проектирование еще не начиналось, используется *модель состава приложения (Application Composition Model)*. В ее рамках трудоемкость проекта оценивается в человеко-месяцах по формуле
  $Effort = A*Size$.
  - *Size* представляет собой оценку размера в терминах экранов, форм, отчетов, компонентов и модулей будущей системы. Каждый такой элемент оценивается с коэффициентом от 1 до 10 в зависимости от своей сложности.
  - Коэффициент *A* учитывает возможное переиспользование части компонентов и производительность разработки, зависящую от опытности команды и используемых инструментов и оцениваемую числом от 4 до 50.
    A = (100 - (процент переиспользования))/производительность.
- На следующих этапах, когда требования уже в основном известны и начинается разработка архитектуры ПО, используется *модель этапа предварительного проектирования (Early Design Model)* и следующие формулы.
  Для трудоемкости (в человеко-месяцах):
  $Effort = A*(Size)^B*M_E$ + (трудозатраты на автоматически генерируемый код)
  Для времени (в месяцах):
  $Time = T*Effort_S^{(0.28+0.2*(B-1.01))}*Sced$.
  - Коэффициент *A* считается равным 2.45, а *T* считается равным 3.67.
  - *Size* — оценка размера ПО в тысячах строк кода.
  - *B* — фактор процесса разработки, который вычисляется по формуле:
    $B = 0.91 + 0.01*\Sigma_{i=1..5}W_i$,
    где факторы Wi принимают значения от 0 до 5:
    - W_1 — предсказуемость проекта для данной организации, от полностью знакомого (0) до совсем непредсказуемого (5);
    - W_2 — гибкость процесса разработки, от полностью определяемого командой при выполнении общих целей проекта (0) до полностью фиксированного и строгого (5);
    - W_3 — степень удаления рисков, от полной (0) до небольшой (5), оставляющей около 80% рисков;
    - W_4 — сплоченность команды проекта, от безукоризненного взаимодействия (0) до больших трудностей при взаимодействии (5);
    - W_5 — зрелость процессов в организации, от 0 до 5 в виде взвешенного количества положительных ответов на вопросы о поддержке ключевых областей процесса в модели CMM (Лекция 2).

- M_E — произведение семи коэффициентов затрат, каждый из которых лежит в интервале от 1 до 6:
  - возможности персонала;
  - надежность и сложность продукта;
  - требуемый уровень повторного использования;
  - сложность платформы;
  - опытность персонала;
  - использование инструментов;
  - плотность графика проекта.
- $Effort_S$ обозначает оценку трудоемкости без учета плотности графика, а *Sced* — требуемое сжатие времени выполнения проекта.

- После того, как разработана архитектура ПО, оценки должны выполняться с использованием *постархитектурной модели (Post-Architecture Model)*.
  Формула для трудоемкости (в человеко-месяцах):
  $Effort = A*(K_{req}*Size)^B*M_P$ + (трудозатраты на автоматически генерируемый код)
  Для времени — та же формула, что и в предыдущей модели (в месяцах):
  $Time = T*Effort_S^{(0.28+0.2*(B-1.01))}*Sced$.
  - *Size* = (размер нового кода в тыс. строк) + *RSize*, где
    RSize = (размер переиспользуемого кода в тыс. строк) * (100 – *AT*)/100 * (*AA* + 0.4**DM* + 0.3**CM* + 0.3**IM* + *SU*)/100
    - *AT* — процент автоматически генерируемого кода;
    - *AA* — фактор трудоемкости перевода компонентов в повторно используемые, от 0 до 8;
    - *DM* — процент модифицируемых для переиспользования проектных моделей;
    - *CM* — процент модифицируемого для переиспользования кода;
    - *IM* — процент затрат на интеграцию и тестирование повторно используемых компонентов;
    - *SU* — фактор понятности переиспользуемого кода, от 10 для простого, хорошо структурированного, до 50 для сложного и непонятного; равен 0, если *CM* = *DM* = 0.
  - Все коэффициенты, кроме K_{req} и M_P, имеют те же значения, что и в предыдущей модели.
  - Коэффициент K_{req} вычисляется как (1 + (процент кода, выброшенного из-за изменений в требованиях)/100).
  - Коэффициент MP является произведением 17-ти коэффициентов затрат, имеющих значения от 1 до 6:
    - надежность продукта;
    - сложность продукта;
    - размер базы данных разрабатываемого приложения;
    - требуемый уровень повторного использования;
    - требуемый уровень документированности;
    - уровень производительности по времени;
    - уровень требований к занимаемой оперативной памяти;
    - изменчивость платформы;
    - возможности аналитика проекта;
    - возможности программистов;
    - опыт работы команды в данной предметной области;
    - опыт работы команды с используемыми платформами;
    - опыт работы команды с используемыми языками и инструментами;

- уровень текучести персонала в команде;
- возможности используемых инструментов;
- возможности общения между членами команды;
- фактор сжатия графика проекта.

Для тех, кто уже отчаялся понять все хитросплетения этой модели, скажем, что имеются программные инструменты, автоматизирующие расчеты по ее формулам.

Управление ресурсами

В данном разделе рассматриваются вопросы управления ресурсами проекта, исключая специфические аспекты, касающиеся только персонала.

Одной из основных задач управления ресурсами является планирование проекта на основе имеющихся ресурсов и оценок ресурсоемкости отдельных работ, а также доработка планов при возникновении изменений в ходе проекта, связанных с неожиданными работами, изменением оценок или изменением доступных ресурсов.

При разработке графика проекта нужно выполнить следующие действия.

- Уточнить имеющуюся структуру работ проекта для того, чтобы использовать ее в рамках выбранного процесса разработки. Например, структура работ может соответствовать набору функций, которые должны иметься в результирующем продукте. Использование процесса XP может потребовать составить план поставок не в соответствии с наборами поставляемых функций, а в соответствии с понедельным планом поставок. В этом случае тяжело предвидеть наборы функций, которые будет иметь очередная поставляемая версия продукта, но можно спланировать еженедельные поставки, их обкатку у пользователей, анализ результатов и доработки по его итогам.
- Установить зависимости между отдельными работами, присутствующими в уточненной структуре работ проекта. Зависимости могут иметь разный характер: финиш-старт (работа может быть начата только после конца другой), старт-старт (работа может быть начата только с началом другой), старт-финиш и финиш-финиш. Если вы встречаете более сложную зависимость типа «работу можно начать, только если сделана некоторая часть другой», это признак того, что работы декомпозированы недостаточно детально и нужно разбить вторую работу на части.
  Вставив фиктивные работы, не требующие ресурсов и времени, можно все зависимости привести к виду финиш-старт.

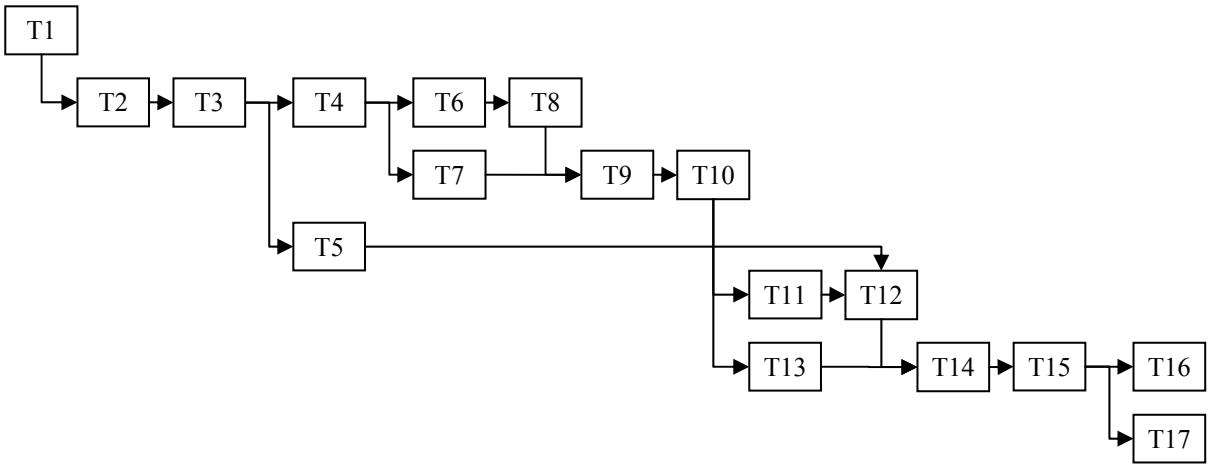


Рисунок 86. Пример сетевой диаграммы проекта.

- Оценив время выполнения и трудоемкость каждой из работ, можно построить ***сетевую диаграмму*** проекта, пример которой приведен на Рис. 86. Расшифровка названий работ, их трудоемкости и время выполнения приведены ниже, в Таблице 13.

Код	Название	Трудоемкость, ч·мес	Время, мес
T1	Формулировка целей и содержания проекта	0.6	0.3
T2	Сбор и анализ требований	3.0	1.0
T3	Разработка архитектуры	6.0	2.0
T4	Первичное планирование	0.6	0.3
T5	Разработка маркетинговых документов	0.3	0.3
T6	Реализация прототипа	3.0	1.0
T7	Детальное проектирование	6.0	2.0
T8	Испытания прототипа	0.6	0.3
T9	Анализ результатов испытаний и изменения проекта	1.0	0.3
T10	Детальное планирование	1.0	0.3
T11	Разработка и отладка пользовательского интерфейса	3.0	1.0
T12	Разработка пользовательской документации	2.0	1.0
T13	Реализация	8.0	2.0
T14	Тестирование	4.0	1.0
T15	Доработка по результатам тестирования	4.0	1.0
T16	Развертывание	1.5	0.5
T17	Обучение пользователей	2.0	0.5

Таблица 13. Работы проекта, сетевая диаграмма которого показана на Рис. 86.

Если связи между работами в сетевой диаграмме превратить в вершины, а вершины-работы в связи (и добавить две новых вершины — начало и конец проекта), то получится так называемая ***PERT-диаграмма*** (PERT — program evaluation and review technique, техника оценки и обзора программ). PERT-диаграмма, соответствующая приведенной ранее сетевой, показана на Рис. 87. Часто различий между этими двумя видами диаграмм не делают, называя обе и сетевыми, и PERT-диаграммами.

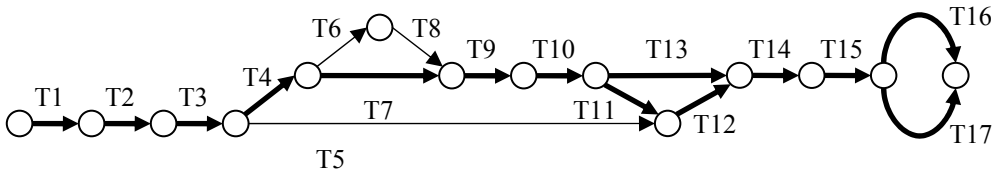


Рисунок 87. PERT-диаграмма для рассматриваемого примера проекта.

Сетевые и PERT-диаграммы используются для планирования продолжительности проекта и выделения ***критических путей*** — последовательностей работ от начала до конца проекта, сумма длительностей которых максимальна среди таких последовательностей. В примере, представленном на Рис. 87 критических путей несколько — работы, лежащие на них, изображены жирными стрелками. Выполнить проект быстрее, чем за время, требующееся для прохождения по критическому пути, нельзя. Поэтому критические пути используют для планирования основных поставок в ходе проекта. В нашем примере длительность проекта не может быть меньше, чем 10.7 месяцев. Кроме того, любая задержка в одной из работ, попавшей на критический путь, обязательно вызовет задержку проекта в целом, значит, такие работы требуют повышенного внимания во время проекта.

Оценка длительности и трудоемкости работ в виде одного числа чаще всего неудобна и может привести к неправильным выводам. На практике используют несколько оценок — максимальную, минимальную, иногда еще и наиболее вероятную. С их помощью можно пополнить набор критических путей и получить более полную информацию о критических работах в проекте, уточнив планы.

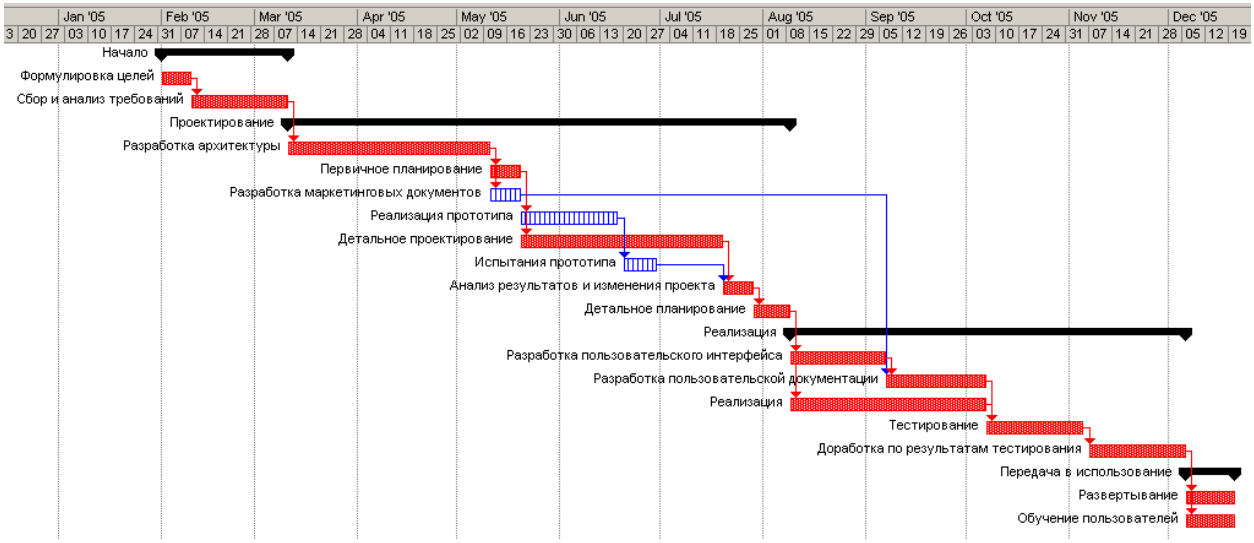


Рисунок 88. Диаграмма Ганта для рассматриваемого примера проекта.

Расписание работ удобно изображать с помощью диаграмм Ганта (Gantt chart). Эта диаграмма показывает и связи между работами, и их длительность во времени. Диаграмма Ганта для рассмотренного примера показана на Рис. 88.

На этой диаграмме также показаны 4 выделенных группы работ: начало, проектирование, реализация и передача в использование.

- Все оценки и планы, сделанные только на основе продолжительностей выполнения отдельных работ, действительны, если у нас имеется достаточно других ресурсов, в частости — персонала.
  На следующем шаге (после вынесения предварительной оценки трудоемкости и продолжительности работ) нужно привязать их к имеющемуся в проекте персоналу и другим ресурсам (оборудованию, материалам и пр.). При этом может оказаться, что некоторые независимые работы не могут проводиться одновременно, поскольку для этого не хватает ресурсов.

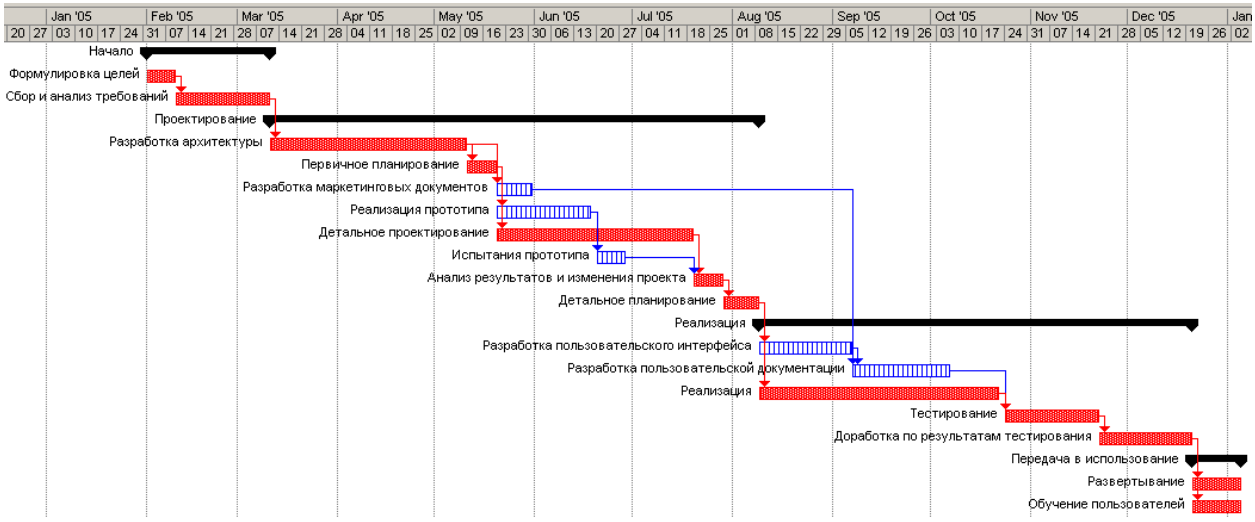


Рисунок 89. График рассматриваемого проекта, построенный с учетом доступных ресурсов.

Допустим, что в нашем примере, как первичное планирование, так и разработка маркетинговых документов требуют полной вовлеченности менеджера проекта. При этом вторая работа может быть начата только после окончания первой. В разработке пользовательского интерфейса могут быть задействованы аналитик, проектировщик интерфейсов и программист. При ограниченности команды проекта это потребует увеличить время на выполнение работы по реализации. Итоговая картина после перепланирования рассматриваемого проекта для команды из менеджера, аналитика-тестировщика, архитектора, трех программистов, один из которых также является

проектировщиком пользовательского интерфейса, и еще одного тестировщика, способного писать техническую документацию, показана на Рис. 89. Она отличается от предыдущей изменениями, удлинившими проект на 0.5 месяца.

Естественно, после учета доступных ресурсов критические пути проекта могут измениться.

Кроме того, риски, связанные с непредвиденными ситуациями, неаккуратной оценкой возможностей персонала или технологий и пр., требуют наличия определенных временных и ресурсных резервов при планировании проектов.

- В ходе самого проекта необходимо тщательно следить за выполнением запланированных работ в срок, за доступностью ресурсов, отслеживать проблемы, способные привести к срыву графика, а также неучтенные при первоначальном планировании факторы. Реальные длительности и трудоемкости работ могут отличаться от оценочных, что потребует построения новых, уточненных планов проекта. Грамотная отработка изменений в планах и расписаниях — ничуть не менее важная задача, чем их первоначальное составление. В частности, чем быстрее менеджер проекта проинформирует о возможных или необходимых изменениях заказчика, спонсора и других лиц, тем более он может рассчитывать на их понимание и помощь в решении проблем проекта.

Специфика управления персоналом

В силу того, что на поведение человека оказывают влияние многочисленные факторы, включая психологические и социальные, люди, работающие в проекте, не могут рассматриваться как обычные ресурсы. Люди склонны по-своему оценивать всю информацию, ставшую им доступной разными путями, преломлять ее через призму своего личного, уникального опыта и характера, и поступать, на первый взгляд, нерационально. Человеку чаще всего недостаточно слов «Вот задача, вот зарплата, — давай, работай!», чтобы начать делать то, что от него хотел бы руководитель проекта.

Перечислить все особенности управления персоналом достаточно трудно. Ниже приведен список лишь некоторых из них, тех с которыми весьма часто приходится сталкиваться на практике.

- *Производительность.*
  Одной из наиболее специфических черт управления персоналом при разработке ПО являются особенности производительности людей. Разработка программ остается в большой степени творческой деятельностью, требует зачастую очень специальных знаний и умений, глубокого понимания вопросов организации информации и аккуратного планирования работы, поэтому нельзя ожидать от людей, участвующих в ней, каких-то средних показателей производительности. Опыт и многочисленные эксперименты [10,11] показывают, что производительность отдельных разработчиков в терминах объема и качества создаваемых ими программ за единицу времени может различаться до 10 раз, и этот разрыв можно наблюдать и в рамках одной организации, и в одной команде.

  На производительность человека, занимающегося обдумыванием вопросов организации ПО, большое влияние оказывает окружение. Эта работа, с одной стороны, требует погружения, и разработчику необходимо достаточно много времени проводить в уединении, вдали от шума и суеты. С другой стороны, иногда требуется консультация эксперта, мнение руководителя или просто информация от других сотрудников по какому-то вопросу. Тишина, когда это нужно, и общение, когда оно необходимо, в правильном сочетании позволяют очень быстро продвигаться в решении трудных задач. А любая другая их комбинация — шум на рабочем месте и отсутствие возможности вовремя посоветоваться со специалистом — часто катастрофически сказывается на производительности такого труда.

  Определение индивидуальной производительности труда является одним из краеугольных камней управления персоналом в большинстве проектов. Но измерить производительность программиста, архитектора или даже тестировщика ПО не так-то просто — самая значимая часть их работы выполняется в их сознании. Поэтому оценить ее аккуратно можно, только

опираясь на общение с сотрудниками, при обоюдном доверии их и руководителя друг к другу. Любые административные выводы из полученной информации (премии, повышения зарплаты, выговоры, вычеты или увольнения) могут привести к недоверию и искажению исходных данных, если только эти выводы не связаны с очевидными фактами принесения пользы или вреда проекту. Поэтому одно из главных правил здесь — крайняя осторожность в выводах и учет личных способностей каждого служащего.

Еще одна особенность разработки ПО связана с достаточно глубокой спецификой практически всех проектов и необходимостью обучения для адекватного вхождения в проект. Нельзя считать, что новый сотрудник сразу будет демонстрировать компетентность в вопросах, связанных с проектом. Обычно он несколько первых дней вносит в него отрицательный вклад, поскольку оттягивает на себя внимание и время других его участников, требует, чтобы его ввели в курс дела. Затем, постепенно, его вклад начинает расти и достигает максимума его возможностей за время от одного до шести месяцев, в зависимости от сложности области проекта.
Соответственно, планирование хода проекта с надеждой на эту максимальную производительности с самой даты прихода разработчика в проект является самообманом. Замена одного из участников проекта на нового стоит значительно больше, чем разность их зарплат, а добавляя новых работников в проект, который отстает от графика, чаще всего вы только увеличите его отставание [10].

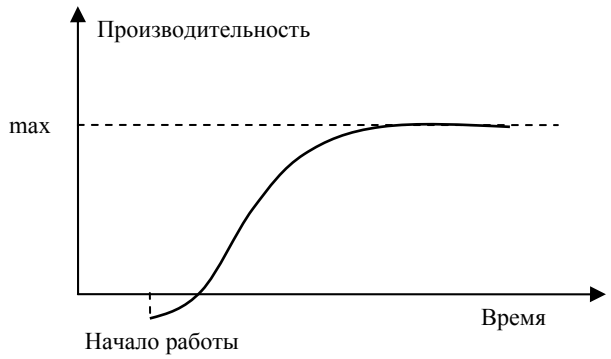


Рисунок 90. Производительность новых сотрудников в проекте.

Другая сторона учета необходимости обучения связана с оценкой хода проекта. Часто дела в проекте через некоторое время после его начала идут медленнее, чем планировалось и хотелось бы. Это отставание от графика может быть вызвано влиянием периода обучения, во время которого сотрудники демонстрируют меньшую производительность. Опытные руководители в таких ситуациях не делают поспешных выводов и не торопятся признать проект провалившимся.

- *Знания и умения.*
Создание сложных программ предполагает умение работать с информацией и алгоритмами на достаточно высоком уровне абстракции, предвидеть свойства той или иной организации данных и действий над ними. Наблюдать эти навыки непосредственно практически невозможно, что-то сказать об их наличии у человека может лишь опыт участия в сложных проектах. При наборе людей в проект нужно помнить, что адекватную оценку разработчику ПО очень трудно дать только по результатам одного-двух разговоров с ним. Вместе с тем, часто приходиться идти на риск, беря в проект новичков. Проект по разработке ПО не сможет стать успешным, если его команда состоит целиком из новичков, она всегда должна включать нескольких опытных и умелых разработчиков.

  Большую ценность для организаций часто имеют общие знания и умения сотрудников, которые явно не формулируются, но активно используются в работе. Иногда потерю такого знания можно ощутить только после ухода из проекта сразу нескольких опытных работников. Эти знания и навыки нужно уметь выделять, формулировать в четком виде и осознанно передавать их новичкам. Чаще же всего такая передача происходит сама собой,

во время общения с более опытными коллегами, незаметно для самих людей, если они обладают достаточно высокой мотивацией к работе.

Для успешной работы в сложном проекте разработки ПО почти всем его участникам, а не только менеджеру, приходится часто принимать решения, так или иначе влияющие на ход проекта. Это предполагает наличие у разработчиков активной жизненной позиции, способности выслушивать разные точки зрения, способности сделать выбор в сложной ситуации, когда среди многих возможных решений нет однозначно лучших, умения аргументировать свою позицию и свои решения. Это требует и открытости, доверия друг к другу, способности положиться на другого человека в достаточно важных вопросах.

- *Мотивация персонала.*
  Мотивация людей — один из самых сложных вопросов управления. Готовых рецептов в этом вопросе предложить нельзя, однозначно можно сказать лишь, что мотивация сугубо индивидуальна и умение применять ее требует внимательного отношения к сотрудникам и понимания основных черт личности каждого из них.

  Считается, что потребности человека, определяющие основные цели его действий, образуют иерархию — *пирамиду Маслоу* (Maslow) [12], Рис. 91.

Рисунок 91. Иерархия человеческих потребностей.

Направление снизу вверх в этой пирамиде соответствует развитию потребностей при росте и развитии человека, а также движению от наиболее базовых и безусловных к обусловленным различными обстоятельствами.

Мотивация людей должна затрагивать разные их потребности и ни в коем случае не сводиться только к денежной составляющей, расположенной на втором-третьем уровне этой пирамиды. Социальные потребности побуждают людей посвящать часть своего времени семье и близким, общению с другими людьми, деятельности в рамках различных неформальных групп. Только одно уважение к этим потребностям, возможность поговорить с коллегами у кофеварки или иногда уйти чуть раньше, чтобы забрать ребенка из яслей, уже может поднять их работоспособность. Открытое признание заслуг человека, его ценности для организации, позволяет удовлетворить потребность в оценке. Потребности в самореализации отвечает предоставление человеку определенной ответственности, возможности самому определять план и содержание своей работы, вести себя индивидуально в одежде, оформлении рабочего места, расположении вещей на столе (довольно часто организации пытаются регламентировать все эти вопросы).

Существует довольно много других подходов к мотивации.

- Деление людей на 3 типа [6]:
  - *люди с целевой ориентацией,* получающие достаточно мотивации от решения задач, постановка которых им понятна;
  - *люди с самоориентацией*, стремящие к личному успеху и признанию и готовые работать для достижения личных целей;
  - *люди с внешней ориентацией*, мотивация которых определяется возможностью работать в коллективе единомышленников, решать общие задачи.

В соответствии с этой классификацией, чтобы побудить человека к действию, нужно выяснить его основную мотивацию и предложить ему нечто отвечающее ей.
Другим важным наблюдением является необходимость иметь хотя бы одного человека с внешней ориентацией в проекте — такие люди часто играют роль клея, соединяющего команду из сильно отличающихся по характеру людей. Отмечено, что женщины чаще мужчин имеют внешнюю ориентацию.

- *Теория справедливости* [1] утверждает, что человек постоянно сопоставляет затрачиваемые усилия и их результаты с получаемыми благами: зарплатой, льготами, признанием, возможностями для профессионального роста и развития, моральным климатом в группе и пр. При этом учитываются блага, получаемые соседом, знакомым, человеком с аналогичным опытом или знаниям в другой организации и т.д., за те же усилия и результаты. Если человек чувствует себя в чем-то обделенным по сравнению с другими людьми, прилагающими, по его оценке, столько же усилий, это снижает его мотивацию. Если же он получает столько же или несколько больше, это побуждает его продолжать работать так же или даже повысить усилия.
  Теория справедливости объясняет, почему так называемые «безбилетники» (free riders) — члены группы, менее интенсивно работающие по сравнению с остальными ее участниками, пользуясь высокими показателями производительности группы в целом, — оказывают пагубное воздействие на мотивацию коллектива в целом.
- В рамках *теории ожиданий* [1] считается, что мотивация человека повышается от осознания им своей способности добиться поставленных целей и получить определенные блага в результате этого. Соответственно, четкая постановка конкретных и интересных задач, ясное определение ценностей проекта, выделение небольших и обозримых, но не простых и скучных, работ, напоминания о прежних успехах, определение критериев вознаграждения в соответствии с вкладом и личными предпочтениями, повышают мотивацию работников.
  Однако известно, что более сильное влияние на мотивацию оказывают незапланированные и нерегулярные вознаграждения и подарки.
- Другие подходы отмечают важность для мотивации таких факторов, как получение оценок результатов от руководства и коллег, автономности, возможности использовать разнообразные навыки и знания в различных областях, участия в работе сплоченной команды.
  В то же время насаждение группового мышления, невозможность высказать свое мнение и привести возражения против принятых правил и решений, которые могут появиться при слишком высоком контроле со стороны руководителя или в группе людей с одинаковыми личностными характеристиками, пагубно сказываются на мотивации.
  Социальная пассивность, отсутствие инициативы и снятие личной ответственности за решение вопросов, связанных с ходом проекта, но не входящих напрямую в обязанности данного сотрудника, также снижают мотивацию коллектива.

Развитие потенциала подчиненных можно осуществлять на основе модели *ситуационного лидерства* [13,14], основные положения которой состоят в следующем.

- Готовность подчиненного выполнить некоторую работу определяется двумя характеристиками: его способностью — знаниями и умениями по отношению к решаемой задаче — и его желанием взяться за нее.
- Развитие подчиненного в терминах его способностей и мотивации происходит по некоторой кривой, аналогичной показанной на Рис. 92. Сначала новый работник мало что умеет, но хочет показать свою полезность и активно принимается за выполнение данных ему заданий, потом умения и знания возрастают, но мотивация уменьшается, а затем, при еще большем росте умений, человеку начинает нравиться то, что он делает.
- Руководитель должен применять к подчиненным индивидуальный подход на основе сочетания двух видов управления. *Директивное управление* заключается в выдаче

четких заданий, определении способов, критериев и сроков их выполнения и жестком контроле. *Поддерживающее управление* направленно на объяснение причин тех или иных действий, обсуждение решений и способов их реализации, поддержку уверенности в себе и самостоятельности, передачу инициативы в руки подчиненного.

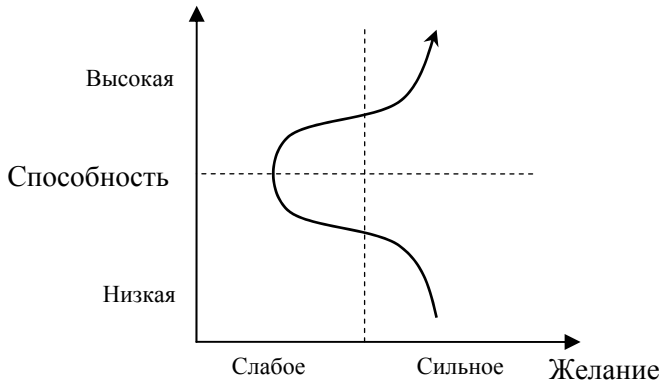


Рисунок 92. Развитие подчиненного в координатах способности и желания.

- Доля директивного управления должна быть обратно пропорциональна способности подчиненного справиться с задачей, а доля поддерживающего — обратно пропорциональна его мотивации.
  При этом новички получают четкие распоряжения, которые нужно выполнить. Более опытным, но недостаточно мотивированным сотрудникам нужно не только давать указания, но и объяснять их смысл. От еще более опытных, но недостаточно самостоятельных надо требовать их мнения по поводу предстоящих действий и принимать решение на основе обсуждения. А высокопрофессиональные и мотивированные подчиненные требуют только самых общих указаний и обратной связи относительно оценки их действий.

- *Построение сплоченной команды.*
  Большое значение для мотивации участников проекта и для успеха проекта в целом имеет сплоченность его команды. Сплоченные команды демонстрируют гораздо более высокую производительность, более эффективны при обучении новых служащих, их члены более заинтересованы в успехе проекта, более склонны к проявлению инициативы, менее подвержены разочарованиям от первых неудач, и т.д. Поэтому построение настоящей сплоченной команды является одной из важных задач руководителя проекта.

  Настоящую сплоченную команду нельзя сделать, сконструировать, — ее построение правильнее называть выращиванием. Как в педагогике или сельском хозяйстве, можно только посадить семена, создать благоприятные факторы и ждать, появится или нет нужный урожай. Факторы, которые способствуют созданию сплоченной команды, таковы [11].

  - Разнородность начального состава и общие мотивации. Людям легче сплотиться, если они обладают разными, дополняющими друг друга навыками и чертами характера, нуждаются в поддержке и помощи друг друга. При этом, конечно, нельзя перебарщивать и собирать в одну группу слишком разных людей с очень разными целевыми установками. Большую помощь в построении группы оказывают люди с внешней мотивацией, нацеленные на общение с коллегами.
  - Неформальные отношения и частое общение. Сплочению способствует поддержка общения между членами группы, не обязательно относящегося к работе, но связанного с выполнением общей деятельности. Обычно в качестве стимулов к этому используют совместные поездки, совместные обеды и торжества, совместный отдых. Нужно поддерживать и общение на рабочие темы — совместные обсуждения целей проекта, основных работ, способов решения возникающих задач, проводить мозговые штурмы. В частности, члены команды не должны сидеть в удаленных помещениях, тем более — в

разных городах. Для обсуждения общих вопросов должно быть выделено специальное помещение.

- Высокие стандарты качества. Группа должна быть объединена общим стремлением сделать результаты проекта лучше. Это придает команде некоторый оттенок элитарности, который сближает ее членов.
- Открытый стиль руководства. Доверие к членам группы, поощрение инициативы, самостоятельности и самоорганизации, поощрение поведения, нацеленного на сотрудничество. Член сплоченной группы сам всегда готов оказать помощь коллеге и не постесняется попросить помощь у других. Он доверяет своим товарищам и знает, что они также полагаются на него.
- Надлежащее техническое обеспечение работы команды и управление. Организация удобной для работы среды. Сотрудники команды должны видеть, что руководитель действительно заботится о том, чтобы им было удобнее работать вместе, чтобы они могли работать над проектом по существу, а не терять время на бюрократические и организационные вопросы.
- Поощрение индивидуальности и личностного отношения к работе, создание возможностей для самовыражения. Используя все грани своей личности, человек может внести наибольший вклад в общую работу.

Есть и набор факторов, которые наоборот, препятствуют сплочению команды.

- Неясные цели проекта, нудная и механическая работа, постановка задач, которые не воспринимаются людьми как достойные и интересные.
- Нехватка ресурсов и времени, давление со стороны руководства, снижение требований к качеству результатов, сверхурочная работа. Команда обычно готова поработать немного больше, когда близкий к завершению проект чуть-чуть не попадает в нужный срок. Но использование свыше 2-х часов сверхурочных работ в день и более недели в месяц очень плохо сказывается на качестве результатов, на работоспособности и на мотивации людей.
- Защитный стиль руководства, при котором руководитель пытается контролировать результаты всех подчиненных, не доверяет им полностью. Лишение подчиненных возможности самим принимать решения в рамках их работы и проявлять самостоятельность губит инициативу. Насаждение конкуренции между членами одной команды порождает недоверие и разрушает сплоченность.
- Стандартизация и бюрократия. Неудобная для работы обстановка — шумные, тесные офисы, труднодоступность мест для проведения обсуждений, географическое разделение команды, размещение в далеких друг от друга помещениях или даже в разных городах — делает совместную работу слишком трудной, чтобы она могла приносить удовольствие.

- *Конфликты.*

  При долгой совместной работе нескольких людей практически всегда неизбежны конфликты. Они не всегда имеют только отрицательные последствия, иногда конфликт возникает из-за различного понимания целей проекта или обуславливается проводимыми в организации изменениями. Некоторые такие конфликты иногда полезно искусственно подтолкнуть, иначе не получится выявить интересы вовлеченных в них лиц и содействовать более глубокому их удовлетворению при разрешении конфликта.

  Конфликты в проекте могут возникать по поводу его целей и приоритетов, по поводу административной политики и процедур, по поводу персонала и других ресурсов, времени и бюджета, из-за графика проекта, из-за технических разногласий по способам его реализации, а также из-за личных взаимоотношений людей.

  Руководителю, стремящемуся к успеху проекта, прежде всего нужно знать о возможных конфликтах по его поводу. Для этого надо выявить всех заинтересованных лиц и достаточно адекватно представлять их интересы, особенно интересы и потребности

спонсоров, организации-исполнителя и организации-заказчика и их руководства. Нужно также научиться хорошо понимать людей, работающих в проектной группе.

Выделяют пять основных методов поведения при конфликтах.

- ***Уклонение.*** Стремление избежать конфликтной ситуации, замалчивать и игнорировать в максимальной степени.
  Этот метод используется, если необходимо выиграть время для более полного осознания ситуации, когда проблема может разрешиться сама или когда возможный выигрыш, как и возможный проигрыш, достаточно малы.
  Уклонение не разрешает конфликт. Чаще всего, при этом конфликт только усиливается и может проявиться вновь в более серьезной форме.
- ***Сглаживание.*** Перенос внимания сторон на общие ценности, отказ от рассмотрения спорных вопросов при таком же поведении другой стороны.
  Такой метод применяется для сохранения хороших отношений, при необходимости избежать конфронтации для достижения общих целей.
  Он разрешает конфликт лишь на короткое время, которое может быть использовано сторонами для подготовки к более глубокому разрешению.
- ***Силовое разрешение.*** Принуждение одной из сторон принять точку зрения другой.
  Оно используется при необходимости быстро разрешить конфликт, когда одна из сторон более близка к позиции руководителя или находится в более выгодном положении, когда руководитель уверен в своих силах и не нуждается в одобрении своих действий.
  Этот метод разрешает конфликт на короткое или среднее время. Конфликт может проявиться позднее в другой форме.
- ***Компромисс.*** Этот метод требует от каждой из сторон пойти на некоторые уступки, что, однако, часто не дает им тех результатов, которых они хотели бы.
  Он используется для временного решения сложных проблем, для сохранения отношений между сторонами, не имеющими преимуществ друг перед другом, когда их возможные выигрыши не слишком высоки, а сохранение отношений имеет большое значение.
  Компромисс разрешает конфликт на среднее или долгое время.
- ***Сотрудничество.*** Открытое сопоставление точек зрения и интересов конфликтующих сторон с целью нахождения максимально выгодных им обеим решений.
  Применяется, если возможный выигрыш для обеих сторон достаточно велик, когда стороны питают взаимное уважение друг к другу и пытаются обеспечить долговременное сотрудничество, когда время, потраченное на разрешение конфликта, с лихвой окупится полученными преимуществами.
  Этот метод обеспечивает долговременное или полное разрешение конфликта.

Для эффективного управления конфликтами руководителю нужно поощрять открытость, доверие и сотрудничество между подчиненными, а также другими заинтересованными лицами, заботиться о получении необходимой информации о конфликтующих интересах. Кроме этого, надо уметь использовать эту информацию, иметь навыки поиска решений, быть способным внимательно выслушивать и убеждать разные стороны, а также принимать на себя проявления агрессии отдельных участников конфликтов.

- *Лидерство и влияние.*
  Один из аспектов управления персоналом — развитие руководителем проекта своих собственных качеств как лидера, способного эффективно концентрировать усилия людей на нужных задачах, вести за собой и обучать персонал работе в условиях ограниченных ресурсов и высоких требований. Частично этот аспект обеспечивается умением использовать развивающее управление, например, ситуационное лидерство.

  Умения руководителя хорошо развиваются при их использовании в как можно более разнообразных проектах, проводящихся в разных областях. Это дает руководителю опыт и знания, которые невозможно получить на каких-либо курсах и тренингах.

Важная характеристика лидера — это *влияние*, которым он обладает. Влияние рассматривается как одна из составляющих власти, наряду с полномочиями, определяющими официальные рамки власти управленца в организации, и статусом, определяющим сложившееся отношение к занимаемой им позиции (а не к самому человеку, что как раз и является его личным влиянием).

Влияние руководителя поддерживается и укрепляется за счет нескольких факторов.

- Укрепление репутации эксперта в технических аспектах и предметной области, что повышает влияние на руководство организации, и реальных знаний в этой области и технологиях, что повышает доверие со стороны подчиненных.
- Акцент социальных взаимоотношений внутри организации, лежащий ближе к деловым аспектам, чем к предпочтениям и привычкам. Более важно поддерживать и укреплять связи с теми, кто может помочь в достижении деловых целей, а не с более статусными руководителями или старыми знакомыми. Нужно развивать связи с другими экспертами и специалистами, которые могут быть востребованы.
- Выбор правильной тактики общения в зависимости от его целей и другой стороны. Например, человеку труднее отказать в просьбе, если он обращается с ней в личной беседе, более легко — по телефону, еще легче — если просьба сформулирована в письме.
- Внимание к нуждам партнера, гибкость и способность идти на компромисс при четкой формулировке и твердой защите своих потребностей. Умение правильно истолковывать невербальные сигналы, жесты, уклонение от разговора и пр.

Управление рисками

Несмотря на все попытки спланировать ход проекта, в нем всегда присутствуют неожиданные ситуации. Неопределенность не может быть удалена совсем из нашей жизни, да и не стоит ее удалять — полностью предсказуемая жизнь очень скучна.

Неопределенности, влияющие на проект, могут грозить ему отрицательными последствиями, вплоть до полного провала, или положительными. Неопределенности первого вида принято называть ***рисками*** проекта, а второго — ***благоприятными возможностями***.

И рисками, и благоприятными возможности нужно уметь управлять. Что означает такое управление? Во-первых, это умение предвидеть определенные такие возможности и риски, во-вторых, это умение оценивать их последствия для проекта, и, в-третьих, это умение выстраивать ход проекта так, чтобы воспользоваться благоприятными возможностями, если они наступят, и противостоять действию рисков, если они реализуются.

Как можно предвидеть риски? Для этого можно воспользоваться опытом предыдущих проектов, проанализировать допущения и слабые места данного проекта, провести опрос экспертов или мозговой штурм, нацеленный на предсказание возможных неблагоприятных для хода проекта событий и обстоятельств. Чтобы такие обсуждения не превращались в базар, а проходили по определенному плану, можно использовать классификацию рисков.

- Риски проекта, влияющие на его ход.
  - *Технологические риски* — недостаточная производительность и гибкость используемых технологий и инструментов.
  - *Кадровые риски* — вероятность не набрать команду или набрать неподходящую, возможность отсутствия у ее членов необходимых навыков, возможность их низкой производительности, вероятность возникновения серьезных конфликтов.
  - *Риски требований* — возможность изменений в требованиях к результатам.
  - *Коммерческие риски* — вероятность неправильной оценки коммерческих составляющих проекта: неверной оценки рынков сбыта, времени и стоимости проекта; возможность непредвиденных расходов.

- *Управленческие риски* — вероятность выбора неправильных целей проекта, недостаточного контроля состояния проекта, возможность принятия неправильных решений и неэффективных мер.
- *Производственные риски* — невозможность или нерентабельность производства продукции и сбои в производстве. При производстве ПО достаточно малы, сводятся к сбоям в изготовлении коробок с продуктом.

- Риски продукта, влияющие на результаты проекта.
  - *Технические риски* — возможность неуспеха в достижении запланированных показателей качества результатов проекта, вероятность вообще не получить нужный результат.
  - *Эксплуатационные риски* — невозможность использования продукта или неготовность организаций-пользователей к его эксплуатации.
  - *Правовые и общественные риски* — возможность возникновения патентных споров, конфликтов с коммерческими, общественными и государственными организациями по поводу самого продукта или его использования.
- Бизнес-риски, относящиеся к ведению дел в организациях, связанных с проектом.
  - *Контрактные риски* — ненадежность соисполнителей, (суб)подрядчиков и поставщиков, возможность возникновения юридических претензий.
  - *Инвестиционные риски* — вероятность отказа или задержек в финансировании со стороны части или всех инвесторов проекта.
  - *Сбытовые риски* — возможность неполучения запланированных доходов от реализации результатов проекта, отказа пользователей от продуктов, сбоев в каналах сбыта.
  - *Конъюнктурные риски* — возможность опережения проекта аналогичными проектами конкурентов, блокады ими рынка, непредвиденной конкуренции.

После определения основных рисков, они подвергаются количественной и качественной оценке. При оценке рисков необходимо обратить внимание на следующие их параметры.

- Последствия для проекта и уникальные составляющие связанной с риском ситуации.
- Вероятность возникновения риска.
- Возможный ущерб для проекта в терминах стоимости, временных задержек, а также самой возможности его продолжения.

Часто вероятность возникновения рисков и ущерб от них оценивается по 4-х или 5-ти балльной шкале, например, ущерб может рассматриваться как незначительный, терпимый, серьезный и катастрофический.

В дальнейшем вырабатывается стратегия преодоления наиболее серьезных рисков — имеющих достаточно высокую вероятность возникновения или способных принести большой ущерб. Обычно используются три вида стратегий преодоления рисков.

- *Стратегии предотвращения или обхода рисков.* Они направлены на снижение вероятности риска или полное избавление от него. Например, низкое качество некоторых компонентов ПО может быть преодолено за счет покупки готовых, уже проверенных рынком компонентов.
- *Стратегии минимизации ущерба.* Например, риск, связанный с болезнью или увольнением архитектора проекта может быть предотвращен введением роли второго архитектора, который обязан быть в курсе всех проектных решений, принимаемых архитектором, и может заменить его. Кроме того, можно организовать еженедельные обсуждения основных проектных решений, чтобы все члены команды получали информацию о них.
- *Планирование реакции на возникающие проблемы.* Например, на случай отказа одного из инвесторов проекта от продолжения его финансирования может быть подготовлен список потенциальных дополнительных инвесторов, с которыми нужно связаться в этом случае, а

также возможных предложений, которые могли бы заинтересовать их в проекте и склонить к участию в его финансировании.

Аналогично, для благоприятных возможностей могут быть разработаны стратегии по повышению их вероятности, усилению их последствий и их правильному использованию.

Наконец, необходимо иметь резервы, позволяющие в ходе проекта преодолевать те риски, которые вообще невозможно было предвидеть. Это могут быть временные резервы для выполнения работ, особенно лежащих на критическом пути, а также резервы занятости персонала и других ресурсов, на случай непредвиденных работ и серьезного превышения оценок трудоемкости.

В ходе проекта нужно постоянно контролировать возникновение ситуаций, связанных с рисками и благоприятными возможностями, обеспечивать реализацию действий по их преодолению (или усилению) и реагированию на них. Кроме того, полезно отслеживать изменения в вероятностях возникновения рисков и тяжести их последствий, что помогает избавиться от необходимости действий по преодолению тех рисков, которые становятся незначительными, а также отметить возникновение новых угроз или благоприятных ситуаций.

Управление коммуникациями и информационным обеспечением

Огромное значение для успеха проекта имеет и организация обмена информацией внутри его команды и со всем остальным окружением. Можно выделить четыре аспекта коммуникаций, важных для проекта.

- Представительские связи.
  Это все вопросы, касающиеся ознакомления заказчиков, пользователей, людей из внешнего окружения с проектом, его задачами и результатами. Оно особенно полезно в начале, при запуске проекта, для обеспечения положительного общественного мнения и информирования заинтересованных общественных организаций. В конце проекта подача информации о нем должна смещаться в сторону представления результатов и их характеристик.
- Координация работ.
  Эта деятельность связана с налаживанием путей передачи информации внутри проектной команды. Управление коммуникациями внутри проекта направлено на то, чтобы каждый член команды мог вовремя получить всю информацию о его целях, содержании, ресурсах, текущих работах и проблемах, которая нужна ему для более эффективной работы. Этот аспект коммуникаций включает в себя поддержку общения между членами команды и определение процедур подготовки отчетов, запросов и других информационных материалов, которыми они обмениваются в ходе проекта, определение форм обратной связи, с помощью которой оцениваются результаты их деятельности.
- Обмен информацией внутри организации-исполнителя.
  Поддержка обмена информации о проекте между его участниками и другими работниками организации-исполнителя может быть полезна для обеспечения положительного образа проекта внутри организации. Кроме того, в этот вид коммуникации входит и информирование руководства о ходе проекта, его проблемах, необходимых ресурсах и пр.
- Разведка и сбор внешней информации.
  Этот аспект коммуникаций важен для правильной постановки целей проекта и формирования его содержания. Сбор и анализ имеющихся данных должны иметь ограниченное влияние на проект — на некоторой стадии нужно удовлетвориться уже собранной информацией и перейти к работе.

Отдельными важными деятельностями в управлении коммуникациями являются составление предложений по проведению проектов и ведение переговоров.

- *Составление предложений.*
  Прежде чем проект начнется, необходимо, чтобы появились заказчик и спонсор, готовые выделить ресурсы на его проведение. Для этого необходимо заранее распространять

информацию об области деятельности и областях экспертизы вашей организации, налаживать связи со служащими других организаций, способными при возникновении потребности подсказать руководству обратиться в вашу организацию.

При поступлении запроса на предложение о будущем проекте, прежде чем составлять его, стоит обратить внимание на следующие вопросы.

- *Бюджет и финансовый цикл.* Имеет ли возможный заказчик свободные средства для формирования бюджета проекта, и может ли он их получить в ближайшее время? Если нет, вряд ли стоит тратить время на разработку предложения именно сейчас, хотя терять связь с таким клиентом не надо.
  Когда возможно выделение денег на данный проект в рамках финансового цикла организации-заказчика? Чем ближе она к окончанию срока, к которому подготавливается бюджет на следующий год или полугодие, и чем больше незакрепленных средств, тем вероятнее выделение денег на проект. Если же бюджет организации на следующий год уже сверстан — лучше попытаться подать предложение к началу следующего финансового цикла.
- *Полномочия.* Имеет ли человек, вышедший на контакт с вами, полномочия на санкционирование проекта, и имеет ли он доступ к лицам, которые могут принимать решение об этом? Если нет, скорее всего, направленное ему предложение не будет успешным.
- *Потребности и возможности.* Существует ли потребность в этом проекте, с которой согласны все принимающие решения лица в организации-заказчике? Достаточно ли она четко сформулирована, или это «принеси то, не знаю что»? Как эта потребность соотносится с возможностями вашей организации? Насколько вашей организации будет легко справиться с неясностями в ее формулировке? Каков риск не удовлетворить нужды клиента, потратив много усилий и денег? Есть ли у вашей организации возможность покрытия расходов на неудачный проект? Если соотношение рисков и возможных доходов не очень хорошее, может быть, не стоит составлять предложение прямо сейчас.

Если ответы на приведенные вопросы дают хорошие шансы получить заказ на проект и ваша организация уверена в возможности выполнения такого проекта, можно браться за составление предложения. Обычно составляет предложение будущий руководитель проекта, с помощью нескольких помощников, включая экспертов по отдельным вопросам деятельности организации и состояния рынка.

Предложение обычно включает следующие разделы.

- *Ответы на вопросы клиента и оценка соответствия его критериям.* Этот раздел содержит ответы на те вопросы, которые ранее были сформулированы заказчиком относительно деятельности и положения вашей организации. Он может перечислять ее оценки по набору критериев, указанных заказчиком как критически важные для него (надежность, опыт работы в данной области, возможность учета особенностей заказчика и пр.).
- *Технический раздел.* Описывает характеристики продукта или услуги, получаемой в результате проекта. Определяет основные способы достижения таких характеристик, возможные проблемы, работы и ответственность за их выполнение.
- *Управленческий раздел.* Описывает возможности вашей организации, используемые методы управления и обоснование схем оплаты проекта.
- *Стоимость.* Содержит информацию о затратах, ценах и условиях оплаты работы.

- *Ведение переговоров.*
  При контакте с внешними заинтересованными лицами, а иногда и с руководителями отдельных подразделений внутри организации-исполнителя проекта, необходимо проводить переговоры. В ходе переговоров выявляются интересы различных сторон, а они получают представление о нуждах проекта. С другой стороны, во время переговором часто

находятся технические, организационные и административные решения, позволяющие решить проблемы проекта. Основная задача переговоров — добиться согласия по тем вопросам, по которым исходные мнения сторон отличались.

В ходе подготовки к переговорам нужно выполнить ряд действий.

- Провести консультации с заинтересованными лицами, находящимися на вашей стороне, и собрать информацию об интересах, критериях и возможных действиях другой стороны.
- Определить ваши интересы. По всем параметрам, по которым нет согласия, нужно определить удовлетворяющие вас решения и оптимальные значения. При этом ряд параметров может представлять интерес для данного проекта независимо от будущих ваших отношений с другой стороной (это *основные интересы*), а другие характеристики будут влиять на возможные будущие отношения (это *дополнительные интересы*). Важной особенностью интересов часто является их неосязаемость и нечеткость. Для их прояснения как раз и выполняется предыдущий шаг.
- Определить проблемы, т.е. те позиции и параметры, по которым у вас нет согласия с другой стороной, а также приоритеты, которые четко обозначают, чем и насколько можно пожертвовать ради достижения других выгод.
- Определить возможные предложения. Обычно, исходя из ваших интересов, определяются три вида предложений: начальные позиции на переговорах, оптимальные результаты с учетом известных вам интересов другой стороны, а также предельные позиции, выйти за рамки которых вы не можете себе позволить.
  Каждое предложение должно быть обосновано, например, с помощью принципа справедливости (деление издержек пополам или в соответствии с выгодами) или принципа необходимости (б***о***льшие издержки несет та сторона, которой решение необходимо в большей степени).
  Для определения возможных предложений нужно также выявить возможные альтернативы ведению переговоров — обращение в суд, к другому поставщику, разрыв отношений и пр. Определив цену этих альтернатив для вашей стороны, можно не принимать те предложения, которые будут менее выгодны, тем самым усиливая собственную позицию.

В команде, участвующей в переговорах, выделяют следующие роли.

- *Лидер,* который руководит переговорами и формулирует позиции.
- *Аналитик,* который внимательно выслушивает противоположную сторону, иногда вмешиваясь в дискуссию, чтобы переформулировать утверждения оппонентов или уточнить определенные моменты. Это нужно, чтобы лидер получил более четкое понимание позиции другой стороны и имел больше времени для ее обдумывания.
- *Протоколист,* который не высказывается без явной просьбы. Он наблюдает за действиями другой стороны, пытаясь понять ее интересы и проблемы, фиксируя ее предложения и контрпредложения своей стороны. Он может быть экспертом в какой-либо области и, по просьбе лидера, выступать с экспертной оценкой тех или иных предложений.

Обычно выделяют пять стратегий ведения переговоров, аналогичных методам разрешения конфликтов, — поскольку переговоры всегда нацелены на урегулирование какого-то конфликта между участвующими в них сторонами. Взаимоотношения между стратегиями и уровнями приверженности интересам сторон изображены на Рис. 93.

- ***Пассивное ожидание.*** Это стремление избежать переговоров, не участвовать в них или отложить их проведение.
  Такая стратегия нацелена на затягивание переговоров для получения дополнительных преимуществ с течением времени или использование альтернативных решений. Используется, когда заинтересованность в результатах переговоров, как своих, так и

другой стороны, мала, или когда с течением времени ваша позиция значительно усиливается.

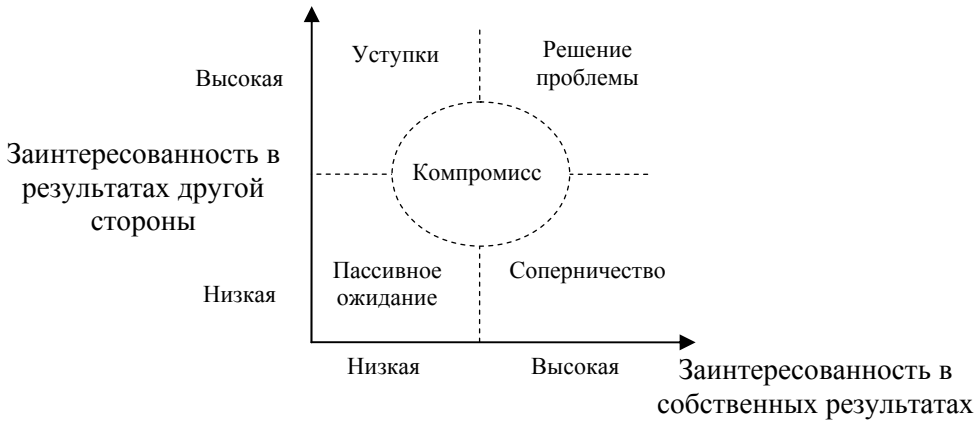


Рисунок 93. Выбор стратегии проведения переговоров.

- ***Уступки.*** Это одностороннее изменение своей позиции в сторону увеличения ее выгодности для противоположной стороны.
  Используется при необходимости быстро достичь решения и большой заинтересованности в продолжении отношений с другой стороной. При этом заинтересованность в собственной выгоде должна быть меньше.
  В результате непродуманных уступок другая сторона может не воспринять ваших интересов или прийти к выводу, что вы сами нечетко их понимаете или не очень привержены им, а это порождает сомнения в вашей надежности как партнера.
- ***Соперничество.*** Это попытки убедить другую сторону в необходимости сделать ее предложения более выгодными для вашей стороны и менее выгодными для нее самой.
  Такая стратегия применяется, когда другая сторона менее заинтересована в своих результатах, чем ваша — в своих, и риск значительно ухудшить отношения невелик.
  При этом могут использоваться запугивание, например, альтернативными переговорам возможностями, и отклонение любых предложений другой стороны.
  Может привести к ухудшению отношений между сторонами, и к снижению статуса и влияния человека, возглавлявшего переговоры с другой стороны.
- ***Компромисс.*** Эта стратегия предполагает взаимные уступки сторон, которые, однако, могут не привести к выгодному для обеих решению.
  Он используется для быстрого решения сложных вопросов, для сохранения отношений между сторонами, не имеющими преимуществ друг пред другом, когда заинтересованность в результатах обеих сторон не слишком велика.
- ***Решение проблемы.*** Такая стратегия предполагает открытое сопоставление интересов и приоритетов для нахождения взаимовыгодного решения с наибольшим выигрышем для обеих сторон.
  Применяется при наличии достаточного времени, взаимном доверии сторон и их обоюдной высокой заинтересованности в получении выгодных результатов.
  Часто стороны не готовы сразу открыть друг другу свои истинные намерения, и такая стратегия реализуется методом проб и ошибок, подачей множества предложений с разными комбинациями параметров решения, после долгих обсуждений и споров.

Деятельность по управлению проектами сложна и разнообразна. В этой лекции рассказано лишь о малой части всего, что необходимо знать и уметь хорошему руководителю. Тем, кто уже знаком с этим, а также тем, кому еще придется использовать на практике описанные в данной лекции методы и приемы, пожелаем главного, что нужно руководителю: умения работать с людьми, умения наблюдать и учиться, а также удачи.

Литература к Лекции 16

[1] Управление проектами. Сборник статей под ред. Дж. Пинто. СПб.: Питер, 2004.

[2] К. Камерон, Р. Куинн. Диагностика и измерение организационной культуры. СПб.: Питер, 2001.
[3] Л. Константайн, Л. Локвуд. Разработка программного обеспечения. СПб.: Питер, 2004.
[4] ISO/IEC TR 14143-4:2002. Information Technology — Software Measurement — Functional Size Measurement — Part 4. Reference Model.
[5] ISO/IEC 20926:2003. Software Engineering — IFPUG 4.1. Unadjusted Functional Size Measurement Method — Counting Practices Manual.
[6] И. Соммервилл. Инженерия программного обеспечения. М.: Вильямс, 2002.
[7] У. Ройс. Управление проектами по созданию программного обеспечения. М.: Лори, 2002.
[8] COCOMO II Model Manual. 1999. http://sunset.usc.edu/research/COCOMOII/.
[9] COCOMO II User Manual. 1999. http://sunset.usc.edu/research/COCOMOII/.
[10] Ф. Брукс. Мифический человеко-месяц или Как создаются программные системы. СПб.: Символ-Плюс, 2001.
[11] Т. Демарко, Т. Листер. Человеческий фактор: успешные проекты и команды. СПб.: Символ-Плюс, 2005.
[12] A. A. Maslow. Motivation and Personality. Harper and Row, NY, 1954.
[13] P. Hersey, K. H. Blanchard. Management of Organizational Behavior: Utilizing Human Resources. Prentice Hall, 1977.
[14] К. Бланшар, П. Зигарми, Д. Зигарми. Одноминутный менеджер и ситуационное руководство. Минск: Попурри, 2002.
[15] Руководство к Своду знаний по управлению проектами (PMBOK Guide). ANSI/PMI 99-001-2004.