Технологии программирования. Компонентный подход

В. В. Кулямин

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.