Принцип инверсии зависимости (The Dependency Inversion Principle)

- Модули верхнего уровня не должны зависеть от модулей нижнего уровня.
- Оба должны зависеть от абстракции.
- Абстракции не должны зависеть от деталей.
- Детали должны зависеть от абстракций.

Содержание

- Определение
- Преимущество использования принципа инверсии зависимостей
- Использования шаблона «Адаптер» между уровнем абстракции и слоем нижнего уровня
- Принцип Инверсии Зависимостей в Java - фреймворке Spring
- Примеры
- Паттерн Стратегия
- Пример: Методы оплаты в интернет магазине

Принцип Инверсии Зависимостей Определение

- Роберт К. Мартин указал на эквивалентность формулировок DIP
- ***Зависеть от абстракций. Не зависеть от конкрементов.***
  Или
- ***Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.***
  Или
- ***Высокоуровневые модули не должны зависеть от низкоуровневых модулей. Оба должны зависеть от абстракций.***

DIP

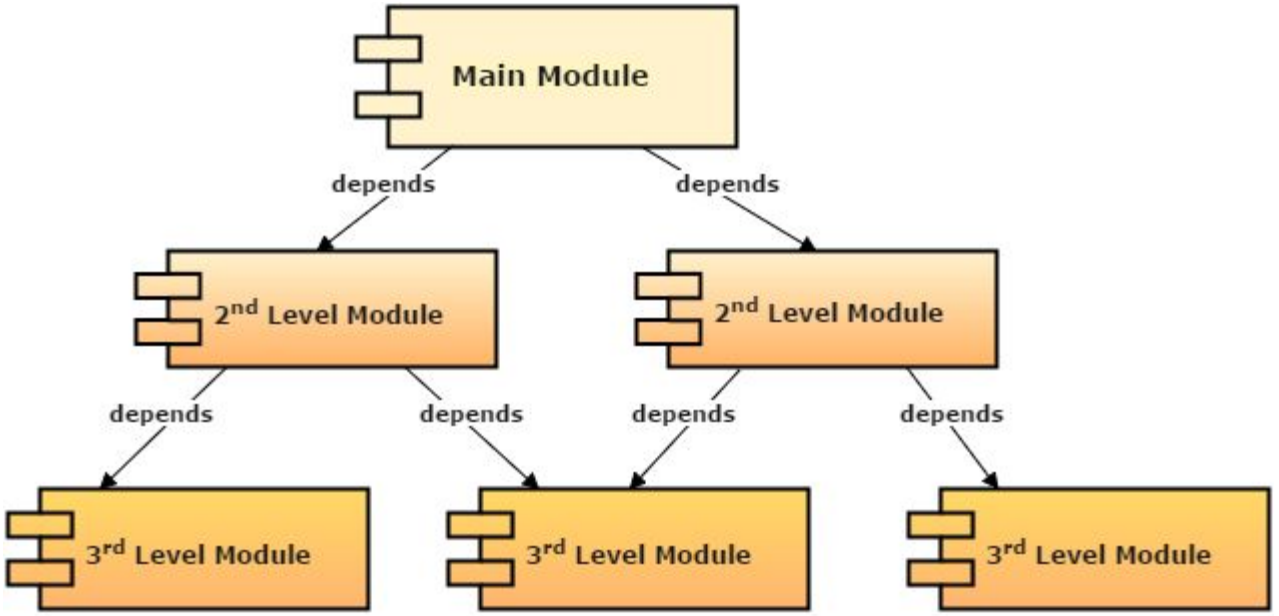
- Реализация модуля высшего уровня будет зависеть от низкоуровневых интерфейсов.
- Фактическая реализация модуля нижнего уровня может варьироваться.
- Вышестоящий модуль через абстрактный интерфейс, способен вызвать любую реализацию модуля нижнего уровня.

Как традиционные процедурные системы организуют зависимости

В процедурных системах, модули высшего уровня зависят от модулей нижних уровней.

На диаграмме ниже показано, процедурные структуры зависимостей.

В итоге образуются традиционные системы зависимостей “сверху-вниз”.

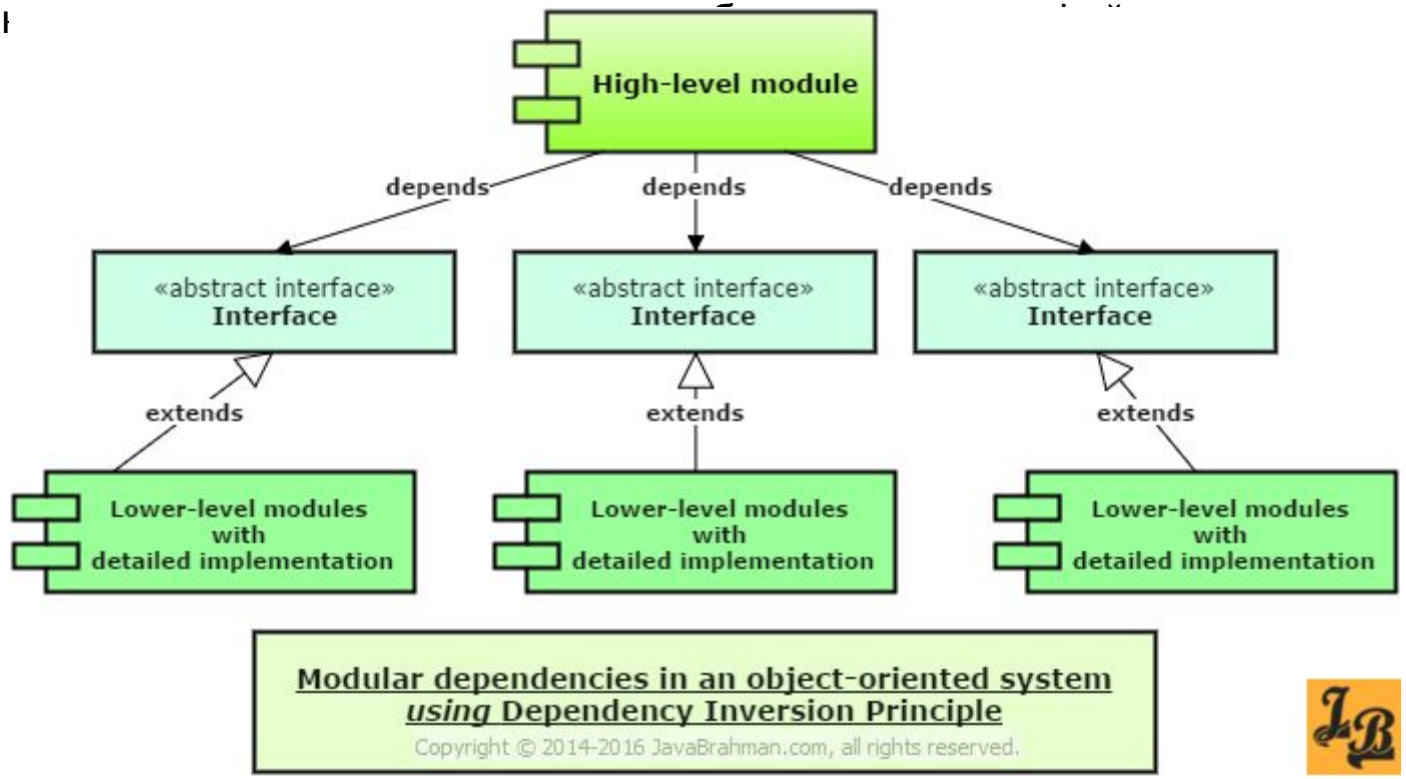


Modular dependencies in a procedural system

ООП

Принцип инверсии зависимости, настаивает на том, что зависимости структур должны быть инвертированы при проектировании объектно-ориентированных систем.

- Модули на высоком уровне зависят от абстрактных интерфейсов, которые определяются на основе услуг которые модуль высокого уровня требует от низкоуровневых модулей.
- Низ



Преимущество использования принципа инверсии зависимостей

- В процедурной системе, можно увидеть тесную связь, слоя с подслоем.

Любое изменение в суб-слое будет иметь волновой эффект в следующем, более высоком уровне и может распространяться еще дальше вверх. Эта связь делает процедурную систему сложной и дорогостоящей для поддержания и расширения функциональности слоев.

- Принцип инверсии зависимостей, с другой стороны, отвергает жесткое сцепление между слоями за счет введения слоя абстракции между ними.

Так, слои более высокого уровня, не напрямую зависят от нижнего уровня, а зависят от абстракции.

Слои на нижнем уровне могут быть потом изменены или расширены без страха *нарушения* высших слоев в зависимости пока они подчиняется контракту абстрактного интерфейса.

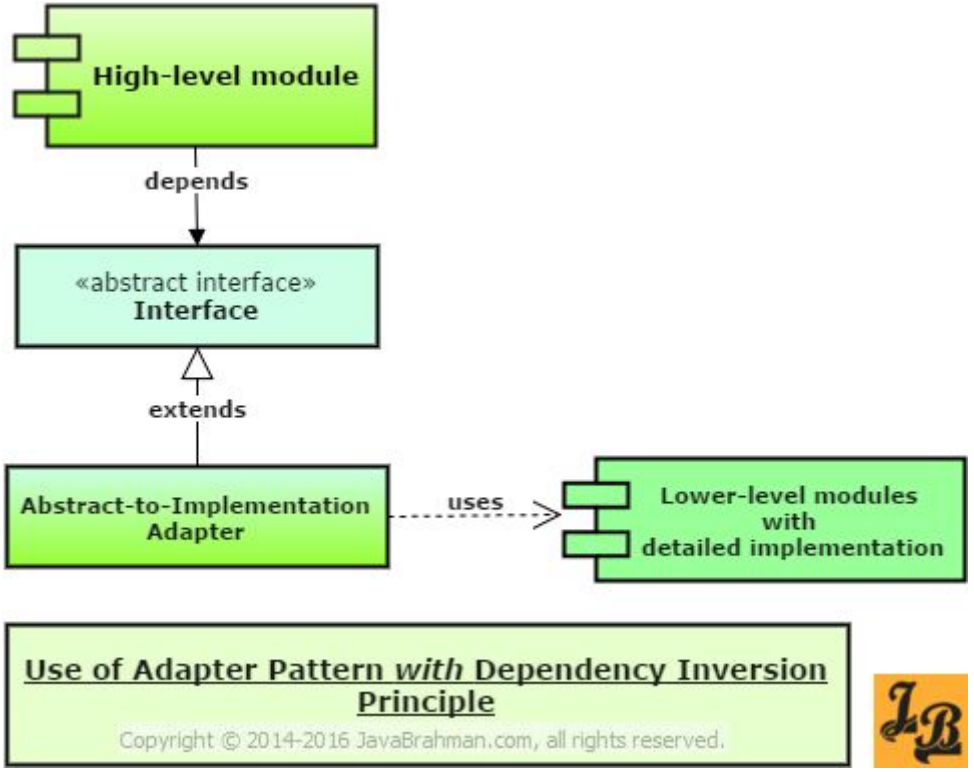
- Таким образом, при инверсии зависимостей слой абстракции между высшим и низшим слоями, осуществляется слабая связь между слоями, которая является весьма полезной для поддержания и расширения всей системы в целом.

Использования шаблона «Адаптер» между уровнем абстракции и слоем нижнего уровня

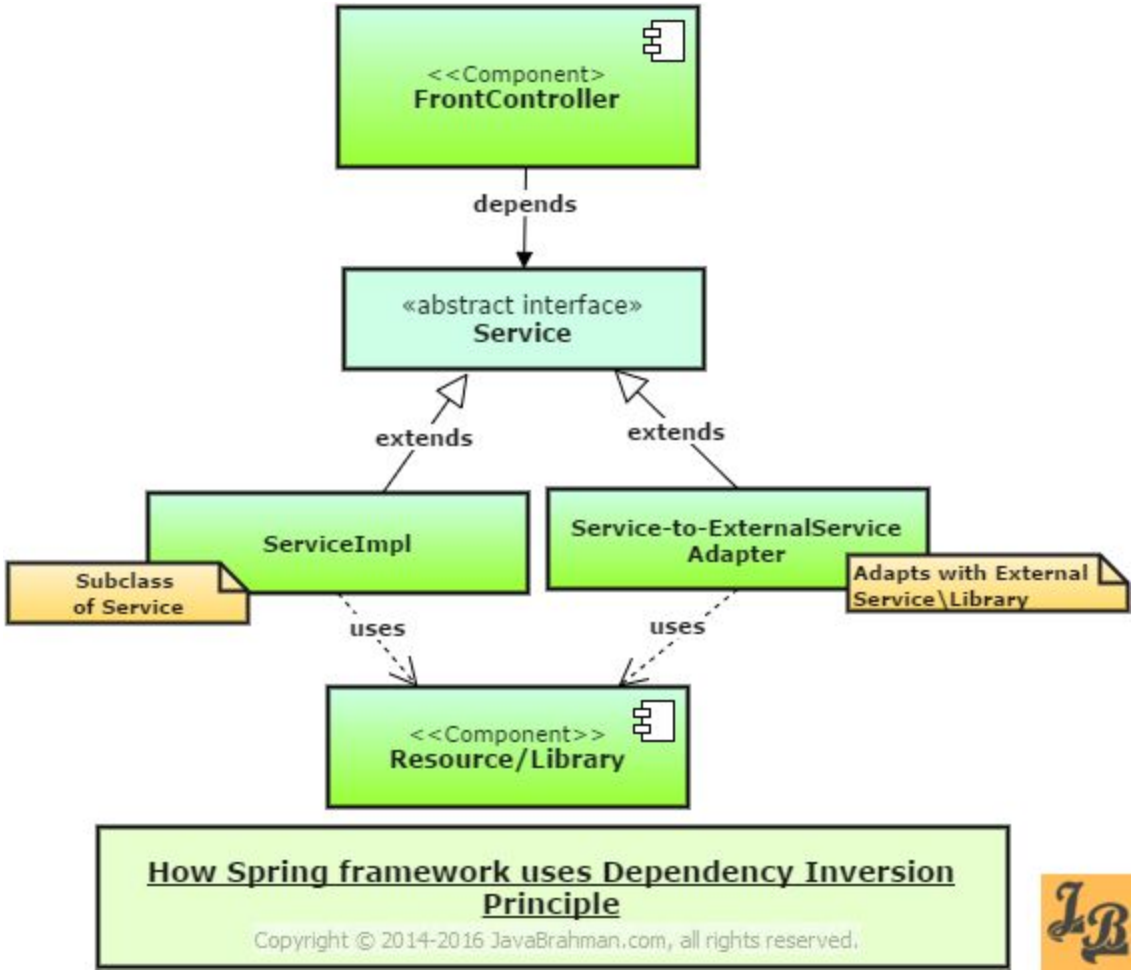
Могут возникнуть сценарии, когда *нельзя* напрямую выразить абстракции интерфейсов от нижнего уровня.

Это может произойти в тех случаях, когда низкоуровневые модули являются частью внешней библиотеки, или модулей дистанционного обслуживания более низкого уровня, таких как веб-службы и т. д.

В таких случаях используется реализация шаблон проектирования адаптер.



Принцип Инверсии Зависимостей в Java - фреймворке Spring



Основные замечания, которые можно сделать из вышеприведенной схемы

Компонент Фронт контроллер использует абстракции *Service* фреймворка Spring.

Фактическая реализация, *ServiceImpl*, осуществляется во время выполнения.

ServiceImpl расширяет услугу и содержит логику выполнения для общения с базой данных/ресурсов.

В случае, если обслуживание вызывает внешняя/Библиотечная служба, то в службе ExternalService используется адаптер, который также расширяет абстракции обслуживания, общение с внешним залом/Библиотекой.

Лицевой слой Контроллера слабо-связан со слоем/доступа к базе данных и с ресурсами.

Следовательно, изменения в нижних слоях, таких как изменения в базе данных, не влияют на фронт-контроллер до тех пор, пока абстракция службы остается неизменной.

Слой сервиса в Java фреймворка Spring, следовательно, использует принцип инверсии зависимостей.

Пример

```
class Book
{
   public string Text { get; set; }
   public ConsolePrinter Printer { get; set; }

   public void Print()
   {
      Printer.Print(Text);
   }
}

class ConsolePrinter
{
   public void Print(String text)
   {
     System.out.println(text);
   }
}
```

Комментарий

- Класс Book, представляющий книгу, использует для печати класс ConsolePrinter.
- При подобном определении класс Book зависит от класса ConsolePrinter.
- Более того мы жестко определили, что печать книгу можно только на консоли с помощью класса ConsolePrinter.
- Другие же варианты, например, вывод на принтер, вывод в файл или с использованием каких-то элементов графического интерфейса - все это в данном случае исключено.
- Абстракция печати книги не отделена от деталей класса ConsolePrinter.
- Все это является нарушением принципа инверсии зависимостей.

В соответствии с принципом инверсии зависимостей

Теперь попробуем привести наши классы в соответствии с принципом инверсии зависимостей, отделив абстракции от низкоуровневой реализации:

```
interface IPrinter{
   void Print(String text);}

class Book{
   public String Text { get; set; }
   public IPrinter Printer { get; set; }

   public Book(IPrinter printer)    {
      this.Printer = printer;    }
    public void Print()    {
      Printer.Print(Text);    }}

class ConsolePrinter implements IPrinter{
   public void Print(String text)    {
      System.out.println("Печать на консоли");    }}

class HtmlPrinter  implements  IPrinter{
   public void Print(String text)    {
      System.out.println("Печать в html");    }
}
```

Сильная связанность. Приложение, которое занимается рассылкой отчетов

```
public class Program {
    public static void main()
    {   Reporter reporter = new Reporter();
        reporter.SendReports();    } }
```

Главный объект в бизнес-логике – Reporter

```
public class Reporter {
    public void SendReports()   throws NoReportsException    {
        ReportBuilder reportBuilder = new ReportBuilder();
       List<Report> reports = reportBuilder.CreateReports();

       if (reports.Count == 0)
          throw new NoReportsException();
       EmailReportSender  reportSender = new EmailReportSender();
       foreach (Report report : reports)
       {
          reportSender.Send(report);       }    }
}
```

Reporter просит ReportBuilder создать список отчетов, а потом один за другим отсылает их с помощью объекта EmailReportSender.

Проблема Тестируемость

Как протестировать функцию SendReports? Давайте проверим поведение функции, когда ReportBuilder не создал ни одного отчета. В этом случае она должна создать исключение NoReportsException:

```
public class ReporterTests
{
    [Fact]
    public void IfNotReportsThenThrowException()
    {
        Reporter reporter = new Reporter();
        reporter.SendReports();
        // ???
    }
}
```

Мы должны «сказать» ReportBuilder'у вернуть пустой список, и тогда функция SendReports выбросит исключение.

Но в текущей реализации Reporter'а сделать мы этого не можем.

Получается, мы не можем задать такие входные данные, при которых SendReports выкинет исключение.

Значит в данной реализации объект Reporter очень плохо поддается тестированию.

Связанность

SendReports, кроме своей прямой обязанности, слишком много знает и умеет:

- знает, что именно ReportBuilder будет создавать отчеты
- знает, что все отчеты надо отсылать через email с помощью EmailReportSender
- умеет создавать объект ReportBuilder
- умеет создавать объект EmailReportSender

Здесь нарушается принцип единственности ответственности. Проблема заключается в том, что в данный момент внутри функции SendReports объект ReportBuilder создается оператором new.

А если у него появятся обязательные параметры в конструкторе? Нам придется менять код в классе Reporter да и во всех других классах, которые использовали оператор new для ReportBuilder'а.

Первые пункты нарушают принцип открытости/закрытости

Дело в том, что если мы захотим с помощью нашей утилиты отсылать сообщения через SMS, то придется изменять код класса Reporter.

Вместо EmailReportSender мы должны будем написать SmsReportSender. Еще сложнее ситуация, когда одна часть пользователей класса Reporter захочет отправлять сообщения через emal, а вторая через SMS.

Объект Reporter зависит не от абстракций, а от конкретных объектов ReportBuilder и EmailReportSender.

Можно сказать, что он "сцеплен" с этими классами. Это и объясняет его хрупкость при изменениях в системе.

Может оказаться, что Reporter жестко зависит от двух классов, эти два класса зависят еще от 4х других. Получится, что вся система – это клубок из стальных ниток, который нельзя ни изменить, ни протестировать.

Этот пример наглядно показывает нарушение принципа инверсии зависимостей.

Применяем принцип инверсии зависимостей

Для начала вынесем интерфейсы IReportSender из EmailReportSender и IReportBuilder из ReportBuilder.

```
public interface IReportBuilder
{
    List<Report> CreateReports();
}

public interface IReportSender
{
    void Send(Report report);
}
```

Передадим создание объектов объекту Reporter в конструктор

```
public class Reporter implements IReporter {
     private IReportBuilder reportBuilder;
     private IReportSender reportSender;
       public Reporter(IReportBuilder reportBuilder, IReportSender
reportSender)     {
         this.reportBuilder = reportBuilder;
         this.reportSender = reportSender;     }
       public void SendReports()     {
        List<Report> reports = reportBuilder.CreateReports();
         if (reports.Count == 0)
           throw new NoReportsException();
         foreach (Report report : reports)         {
           reportSender.Send(report);         }     }
}
```

main

Во время создания объекта Reporter в самом начале программы мы будем задавать конкретные IReportBuilder и IReportSender и передавать их в конструктор:

```
public static void main()
{
     ReportBuilder builder = new ReportBuilder();
     SmsReportSender sender = new SmsReportSender();
    Reporter reporter = new Reporter(builder, sender);

    reporter.SendReports();
}
```

Решенные проблемы

- Тестируемость

Теперь есть возможность передавать в конструктор Reporter'а объекты, которые реализуют нужные интерфейсы.

- Связанность

Реализован главный принцип инверсии зависимостей. Reporter зависит только от абстракций (интерфейсов).

Как быть, если мы хотим отсылать отчеты не через email, а через SMS? Теперь сделать это проще простого . Надо передать в конструктор Reporter'а не EmailReportSender, а SmsReportSender. Код самого Reporter'а мы изменять уже не будем.

Принципы

Инициал	Представляет	Название, понятие
S	SRP	**Принцип единственной ответственности** (***The Single Responsibility Principle***) Существует лишь одна причина, приводящая к изменению класса.
O	OCP	**Принцип открытости/закрытости** (***The Open Closed Principle***) «программные сущности … должны быть открыты для расширения, но закрыты для модификации.»
L	LSP	**Принцип подстановки Барбары Лисков** (***The Liskov Substitution Principle***) «объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы.» (контрактное программирование).
I	ISP	**Принцип разделения интерфейса** (***The Interface Segregation Principle***) «много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения.»
D	DIP	**Принцип инверсии зависимостей** (***The Dependency Inversion Principle***) «Зависимость на Абстракциях. Нет зависимости на что-то конкретное.»

Стратегия

Стратегия — это поведенческий паттерн, выносит набор алгоритмов в собственные классы и делает их взаимозаменяемыми.

Другие объекты содержат ссылку на объект-стратегию и делегируют ей работу.

Программа может подменить этот объект другим, если требуется иной способ решения задачи.

Применимость

Стратегия часто используется в Java коде, особенно там, где нужно подменять алгоритм во время выполнения программы..

Примеры Стратегии в стандартных библиотеках Java:

java.util.Comparator#compare(), вызываемые из Collections#sort().

javax.servlet.http.HttpServlet: метод service(), а также все методы doXXX()принимают объекты HttpServletRequest и HttpServletResponse в параметрах.

javax.servlet.Filter#doFilter()

Признаки применения паттерна: Класс делегирует выполнение вложенному объекту абстрактного типа или интерфейса.

Пример: Методы оплаты в интернет магазине

В этом примере Стратегия реализует выбор платёжного метода в интернет магазине.

Когда пользователь сформировал заказ, он получает выбор из нескольких платёжных средств: электронного кошелька или кредитной карты.

В данном случае конкретные стратегии платёжных методов не только проводят саму оплату, но и собирают необходимые данные на форме заказа.

PayStrategy.java: Общий интерфейс стратегий оплаты

```
package strategy.example.strategies;
/* Общий интерфейс всех стратегий. */
   public interface PayStrategy {
 boolean pay(int paymentAmount);
 void collectPaymentDetails();
 }
```

PayByPayPal.java: Оплата через PayPal

```
package strategy.example.strategies;
 import java.io.*;
import java.util.*;
public class PayByPayPal implements PayStrategy {
private static final Map<String, String> DATA_BASE = new HashMap<>();
 private final BufferedReader READER = new BufferedReader(new
     InputStreamReader(System.in));
 private String email;
 private String password;
 private boolean signedIn;
 static {
 DATA_BASE.put("amanda1985", "amanda@ya.com");
DATA_BASE.put("qwerty", "john@amazon.eu");
}
```

Собираем данные от клиента.

```
@Override
 public void collectPaymentDetails() {
try { while (!signedIn) {
System.out.print("Enter user email: ");
email = READER.readLine();
 System.out.print("Enter password: ");
 password = READER.readLine();
if (verify()) { System.out.println("Data verification was successful"); }
else { System.out.println("Wrong email or password!"); }
 } }
 catch (IOException ex) { ex.printStackTrace();
 } }
```

*Если клиент уже вошел в систему, то для следующей оплаты данные вводить * не придется*

```
private boolean verify()
 { setSignedIn(email.equals(DATA_BASE.get(password)));
 return signedIn; }
@Override
public boolean pay(int paymentAmount) {
 if (signedIn) { System.out.println("Paying " +
    paymentAmount + " using PayPal");
 return true; }
else { return false; }
} private void setSignedIn(boolean signedIn) {
this.signedIn = signedIn; } }
```

strategies/PayByCreditCard.java: Оплата кредиткой

```
package strategy.example.strategies;
 import java.io.*;
/** * Конкретная стратегия. Реализует оплату
    корзины интернет магазина кредитной * картой
    клиента. */
 public class PayByCreditCard implements PayStrategy {
private final BufferedReader READER = new
    BufferedReader(new InputStreamReader(System.in));
private CreditCard card;
```

/** Собираем данные карты клиента. */

```
@Override
 public void collectPaymentDetails() {
try { System.out.print("Enter card number: ");
 String number = READER.readLine();
 System.out.print("Enter date 'mm/yy': ");
 String date = READER.readLine();
System.out.print("Enter cvv code: ");
 String cvv = READER.readLine();
 card = new CreditCard(number, date, cvv);
```

// Валидируем номер карты.

```
} catch (IOException ex) { ex.printStackTrace(); } }
/** * После проверки карты мы можем совершить
    оплату. Если клиент продолжает * покупки, мы не
    запрашиваем карту заново. */
 @Override
 public boolean pay(int paymentAmount) {
 if (cardIsPresent()) { System.out.println("Paying " +
    paymentAmount + " using Credit Card");
 card.setAmount(card.getAmount() - paymentAmount);
return true;
 } else { return false; } }
 private boolean cardIsPresent() { return card != null; } }
```

CreditCard.java: Кредитная карта

```
package strategy.example.strategies;
 public class CreditCard {
 private int amount;
 private String number;
 private String date;
 private String cvv;
 public CreditCard(String number, String date, String cvv) {
this.amount = 100_000;
 this.number = number;
 this.date = date;
 this.cvv = cvv; }
public void setAmount(int amount) {
 this.amount = amount; }
public int getAmount() { return amount; } }
```

Order.java: Класс заказа

```
package strategy.example.order;
 import strategy.example.strategies.PayStrategy;
 /** * Класс заказа. Ничего не знает о том каким способом
    (стратегией) будет * расчитывается клиент, а просто
    вызывает метод оплаты. Все остальное стратегия *
    делает сама. */
public class Order {
 private static int totalCost = 0;
 private boolean isClosed = false;
 public void processOrder(PayStrategy strategy) {
 strategy.collectPaymentDetails();
 // Здесь мы могли бы забрать и сохранить платежные
    данные из стратегии.
 }
```

Класс заказа

```
public void setTotalCost(int cost) {
 this.totalCost += cost; }
public static int getTotalCost() {
return totalCost; }
 public boolean isClosed() {
 return isClosed; }
 public void setClosed() {
 isClosed = true; } }
```

Клиентский код

```
package strategy.example;
 import strategy.example.order.Order;
 import strategy.example.strategies.PayByCreditCard;
import strategy.example.strategies.PayByPayPal;
 import strategy.example.strategies.PayStrategy;
import java.io.*;
 import java.util.*;
public class Demo {
 public static Map<Integer, Integer>  priceOnProducts = new HashMap<>();
 public static BufferedReader reader = new BufferedReader(new
     InputStreamReader(System.in));
 private static Order order = new Order();
private static PayStrategy strategy;
```

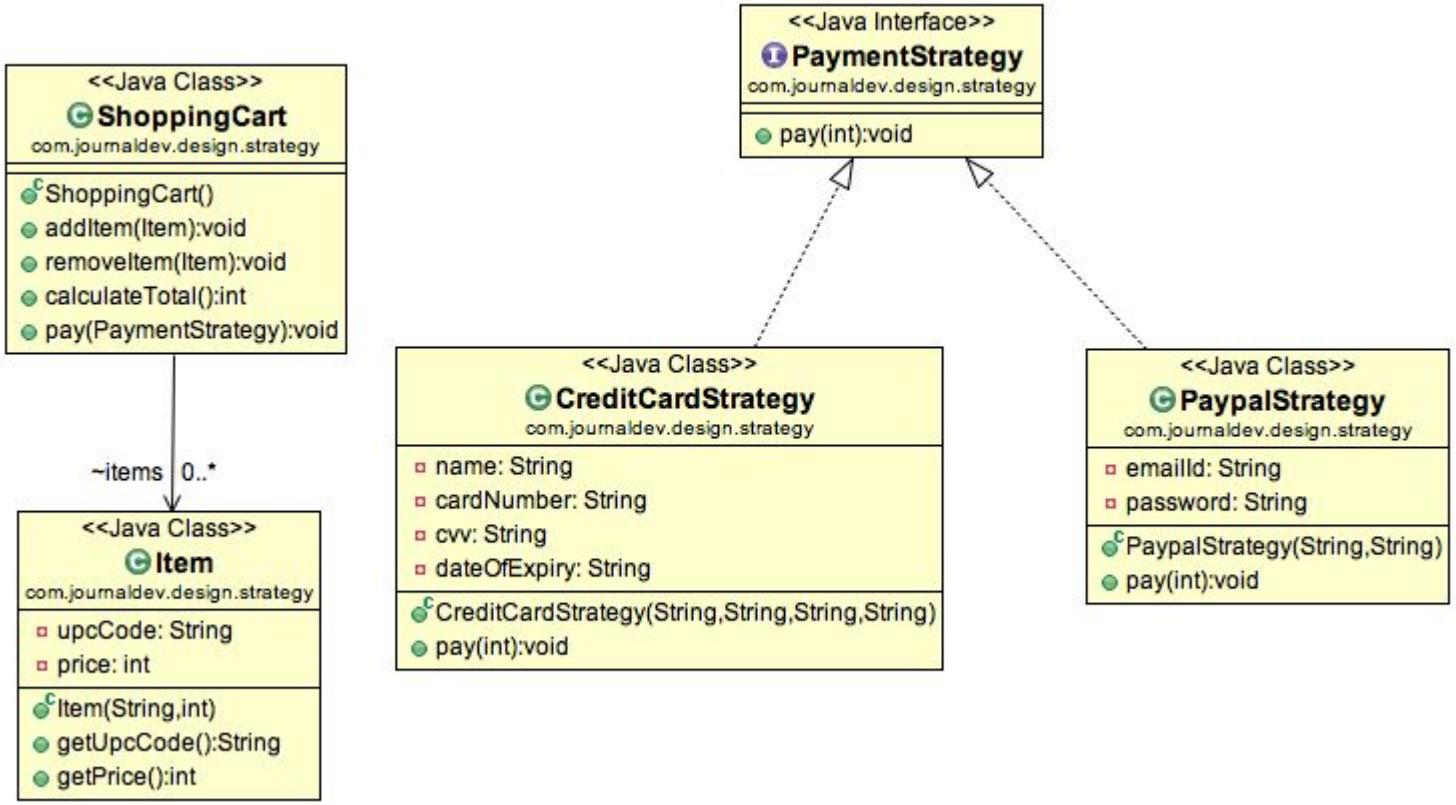
Начальные данные

```
static {
priceOnProducts.put(1, 2200);
priceOnProducts.put(2, 1850);
priceOnProducts.put(3, 1100);
 priceOnProducts.put(4, 890); }
 public static void main(String[] args) throws IOException {
while (!order.isClosed()) {
 int cost;
 String continueChoice;
 do { System.out.print("Select a product:" + "\n" + "1 - Mother board" + "\n" + "2 - CPU"
      + "\n" + "3 - HDD" + "\n" + "4 - Memory" + "\n");
 int choice = Integer.parseInt(reader.readLine());
 cost = priceOnProducts.get(choice);
 System.out.print("Count: ");
 int count = Integer.parseInt(reader.readLine());
 order.setTotalCost(cost * count);
System.out.print("You wish to continue selection? Y/N: ");
```

Выбор стратегий

```
 continueChoice = reader.readLine(); }
while (continueChoice.equalsIgnoreCase("Y"));
 if (strategy == null) { System.out.println("Select a Payment Method" + "\n" + "1 - PalPay" + "\n" + "2 - Credit Card");
String paymentMethod = reader.readLine();
// Клиент создаёт различные стратегии на основании
// пользовательских данных, конфигурации и прочих параметров.
 if (paymentMethod.equals("1")) {
 strategy = new PayByPayPal();
} else if (paymentMethod.equals("2")) {
strategy = new PayByCreditCard(); }
 // Объект заказа делегирует сбор платёжных данных стратегии, т.к.
 // только стратегии знают какие данные им нужны для
 // приёма оплаты. order.processOrder(strategy); }
System.out.print("Pay " + Order.getTotalCost() + " units or Continue shopping? P/C: ");
 String proceed = reader.readLine();
if (proceed.equalsIgnoreCase("P")) {
 // И наконец, стратегия запускает приём платежа.
 if (strategy.pay(Order.getTotalCost())) {
System.out.println("Payment has succeeded"); }
else { System.out.println("FAIL! Check your data");
} order.setClosed(); } } } }
```

Диаграмма классов для паттерна стратегия



Стратегия является одним из самых **поведенческих шаблонов проектирования**

Шаблон стратегии используются, когда у нас есть несколько алгоритмов для решения конкретной задачи, и клиент выбирает фактическую реализацию, которая будет использоваться во время выполнения.

Шаблон Стратегия также известен как **политик**.

Если есть множество алгоритмов, то есть возможность клиентскому приложению, применить алгоритм, который будет использоваться в качестве параметра.

Одним из лучших примером такого рода является Collections.sort() метод, который принимает параметр Компаратор.

На основе различных реализаций Интерфейса Компаратор, объекты получаются отсортированными по разному.

Пример использования Collections.sort()

```
import java.util.*;
 public class Collectionsorting{
    public static void main(String[] args)    {
        // Create a list of strings
        ArrayList<String> al = new ArrayList<String>();
        al.add("Geeks For Geeks");
        al.add("Friends");
        al.add("Dear");
        al.add("Is");
        al.add("Superb");

        /* Collections.sort method is sorting the
        elements of ArrayList in ascending order. */
        Collections.sort(al);

        // Let us print the sorted list
        System.out.println("List after the use of" +
                           " Collection.sort() :\n" + al);    }
}
```

Предположим, что есть класс Person:

```
public class Person {
private String name;
 private Integer age;
private String country; }
```

И его список:

```
List<Person> personList = new ArrayList<Person>();
```

И нужно сортировать его по имени, иногда по возрасту, иногда по странам.

Используем интерфейс Comparator -у него есть метод compare, только он принимает не один параметр, а два.

Решением является создание следующих дополнительных классов:

```
public class NameComparator implements Comparator<Person> {
 public int compare(Person o1, Person o2) {
 return o1.getName().compareTo(o2.getName()); } }

public class AgeComparator implements Comparator<Person> {
 public int compare(Person o1, Person o2) {
return o1.getAge().compareTo(o2.getAge()); } }

 public class CountryComparator implements Comparator<Person> {
public int compare(Person o1, Person o2) {
 return o1.getCountry().compareTo(o2.getCountry()); } }
```

Как работает Collections.Sort ()?

Внутренне метод Sort вызывает метод Compare классов, которые он сортирует. Чтобы сравнить два элемента, он спрашивает «Что больше?».

Метод сравнения возвращает -1, 0 или 1, чтобы сказать, меньше ли он, равен или больше другого.

Он использует этот результат, чтобы затем определить, следует ли их поменять местами для его сортировки.

Затем список можно отсортировать следующим образом:

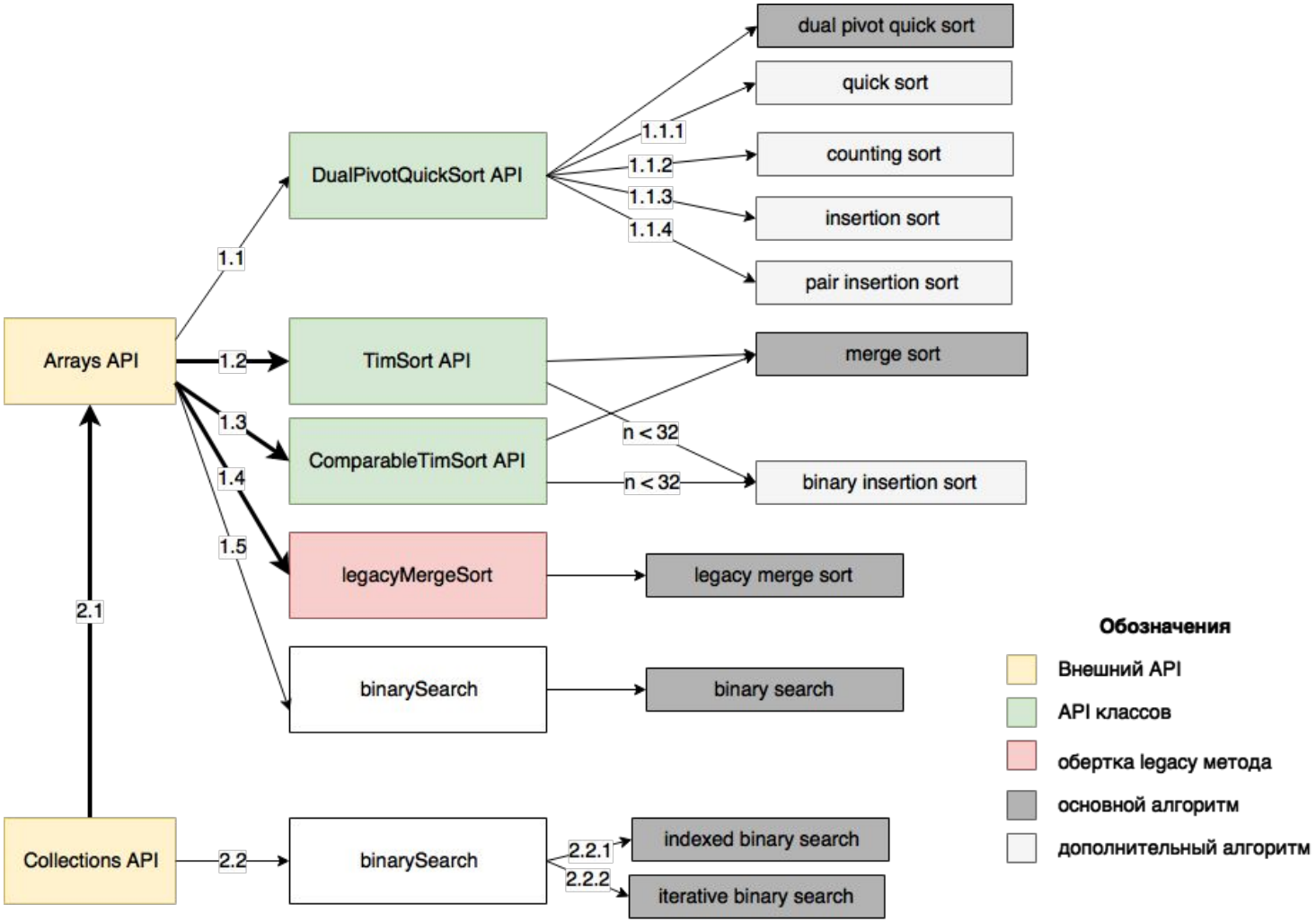
Вызываем метод**Collections.sort**, передаем туда список объектов и еще специальный объект во втором параметре, который реализует интерфейс **Comparator** и говорит, как правильно сравнивать пары объектов в процессе сортировки.

```
Collections.sort(personList, new NameComparator());
Collections.sort(personList, new AgeComparator());
Collections.sort(personList, new CountryComparator());
```

Способ Java 8 - это использовать List.sort следующим образом:

```
personList.sort(Comparator.comparing(Person::getName));
```

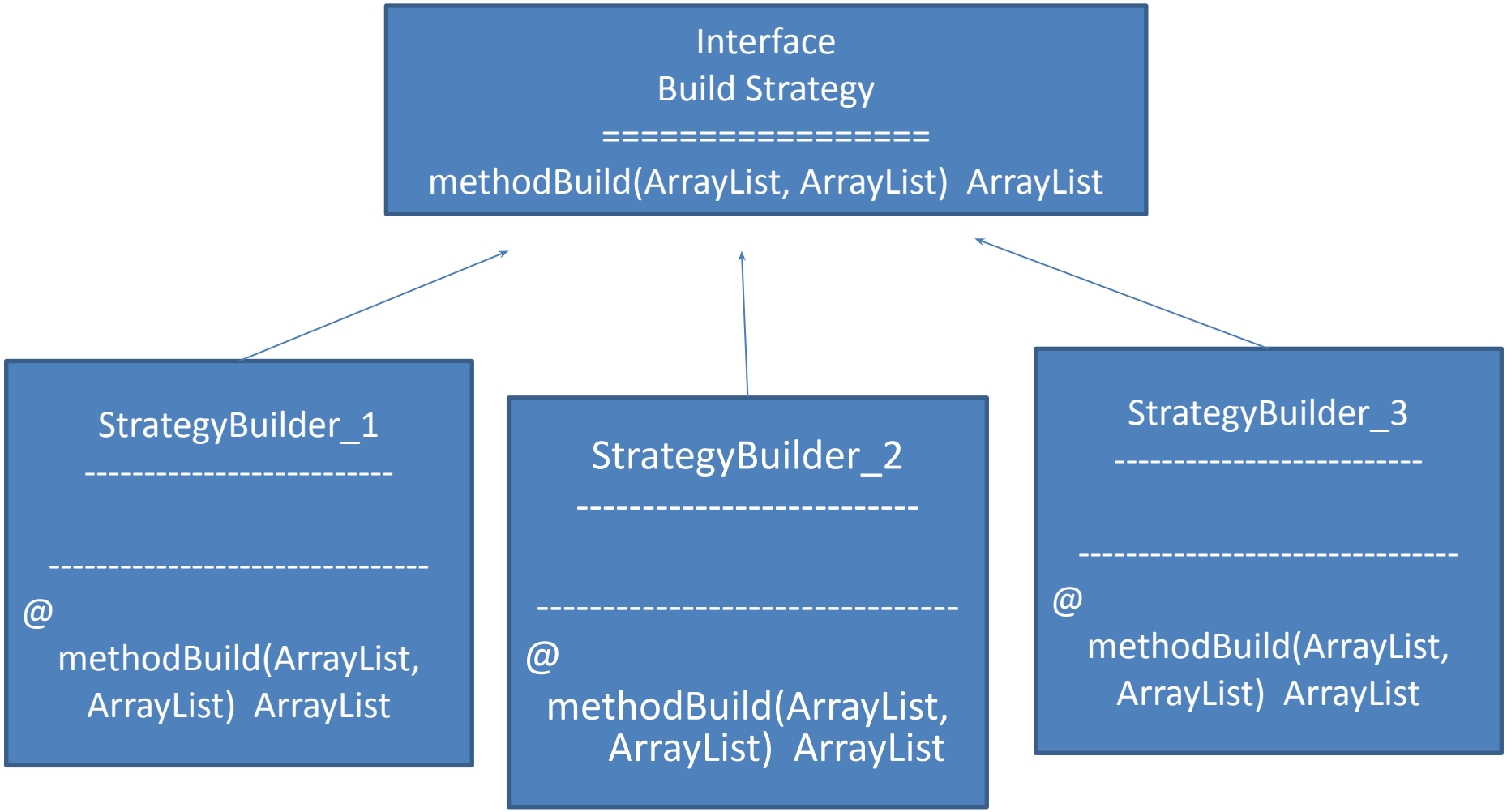
Основные и дополнительные алгоритмы Arrays, Collections



Сводная таблица по алгоритмам Arrays, Collections

Номер	Условие перехода	Алгоритм	Сложность*
1.1	default	dual pivot quick sort	O(n log n)
1.1.1	n < 286	quick sort	O(n log n)
1.1.2	29 < n (byte), 3200 < n (short, char)	counting sort	O(n+k)
1.1.3, 1.1.4	n < 47	insertion sort, pair insertion sort	O(n)
1.2, 1.3	default	merge sort	O(n log n)
1.2.1, 1.3.1	n < 32	binary insertion sort	O(n)
1.4	выставлен ключ VM	legacy merge sort	O(n log n)
1.5	default	binary search	
2.2.1	список с произвольным доступом	indexed binary search	O(log n)
2.2.2	список с последовательным доступом, 5000 < n	iterative binary search	O(log n)

Упражнение использовать паттерн Стратегия вместо паттерна Строитель



Литература

https://refactoring.guru/ru/design-patterns