S.O.L.I.D.

SOLID (single responsibility, open-closed, Liskov substitution, interface segregation, dependency inversion)

Содержание

- Формулировки принципов
- Признаки плохого проекта
- *Принцип единственности ответственности* SRP
- Примеры
- Паттерн Фабричный метод
- Паттерн Строитель(Builder)
- Литература

SOLID –Мнемонический акроним

введённый Майклом Фэзерсом (*Michael Feathers*) для первых пяти принципов, названных Робертом Мартином-в начале 2000-х, которые означали пять основных принципов объектно-ориентированного программирования и проектирования*

* *WIKI :*https://ru.wikipedia.org/wiki/SOLiD

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.»

Признаки плохого проекта

- **Закрепощённость**: система с трудом поддается изменениям, поскольку любое минимальное изменение вызывает эффект "снежного кома", затрагивающего другие компоненты системы.
- **Неустойчивость:** в результате осуществляемых изменений система разрушается в тех местах, которые не имеют прямого отношения к непосредственно изменяемому компоненту.
- **Неподвижность:** достаточно трудно разделить систему на компоненты, которые могли бы повторно использоваться в других системах.
- **Вязкость:** сделать что-то правильно намного сложнее, чем выполнить какие-либо некорректные действия.
- **Неоправданная сложность:** проект включает инфраструктуру, применение которой не влечёт непосредственной выгоды.
- **Неопределенность:** проект трудно читать и понимать. Недостаточно четко выражено содержимое проекта.
- *И.т.д.*

Принцип единственности ответственности SRP

Формулировка SRP: **не должно быть больше одной причины для изменения класса**

Что является причиной изменения логики работы класса?

Видимо, изменение отношений между классами, введение новых требований или отмена старых.

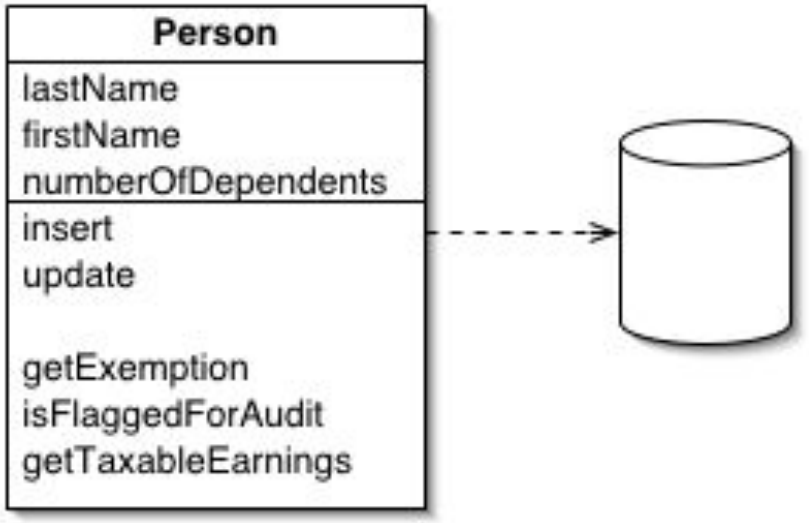
Вообще, вопрос о причине этих изменений лежит в плоскости ответственности, которую мы возложили на наш класс.

Если у объекта много ответственности, то и меняться он будет очень часто.

Таким образом, если класс имеет больше одной ответственности, то это ведет к хрупкости дизайна и ошибкам в неожиданных местах при изменениях кода.

Паттерн Active Record (Активная Запись)

Объект, который обертывает строку в таблицу или представление базы данных, инкапсулирует доступ к базе данных, и добавляет логику домена к данным.



Объект несет в себе как данные, так и поведение. Значительная часть этих данных является постоянными и должна быть сохранена в базе данных.

Active Record использует наиболее очевидный подход, ставящий логику доступа к данным в объект домена.

Таким образом, для Person реализуются чтение и запись своих данных из и в базу данных.

Пример использования паттерна **Active Record**

Проблема
Рассмотрим Приложение ORM.
Суть ORM состоит в том, что по таблицам базы данных, ORM генерирует бизнес-сущности.
Рассмотрим сущность пользователя - Account.
Сценарий использования выглядит так:

```
// создание пользователя
Accounts account = new Accounts();
account.AddNew();
account.Name = "Name";
account.Save();

// загрузка объекта по Id
Accounts account = new Accounts()
account.LoadByPrimaryKey(1);

// загрузка связной коллекции при обращении к свойству объекта
ArrayList<Roles> list = account.Roles;
```

Суть проблемы

Шаблон Active Record может быть успешно использован в небольших проектах с простой бизнес-логикой.

Практика показывает, что когда проект разрастается, то из-за смешанной логики внутри доменных объектов возникает много дублирования в коде и непредвиденных ошибок.

Обращения к базе данных довольно сложно проследить, когда они скрыты, например, за свойством объекта account.Roles.

В данном случае объект Account имеет несколько ответственностей:

- является объектом домена
- хранит бизнес-правила, например, связь с коллекцией ролей
- является точкой доступа к базе данных

Решение

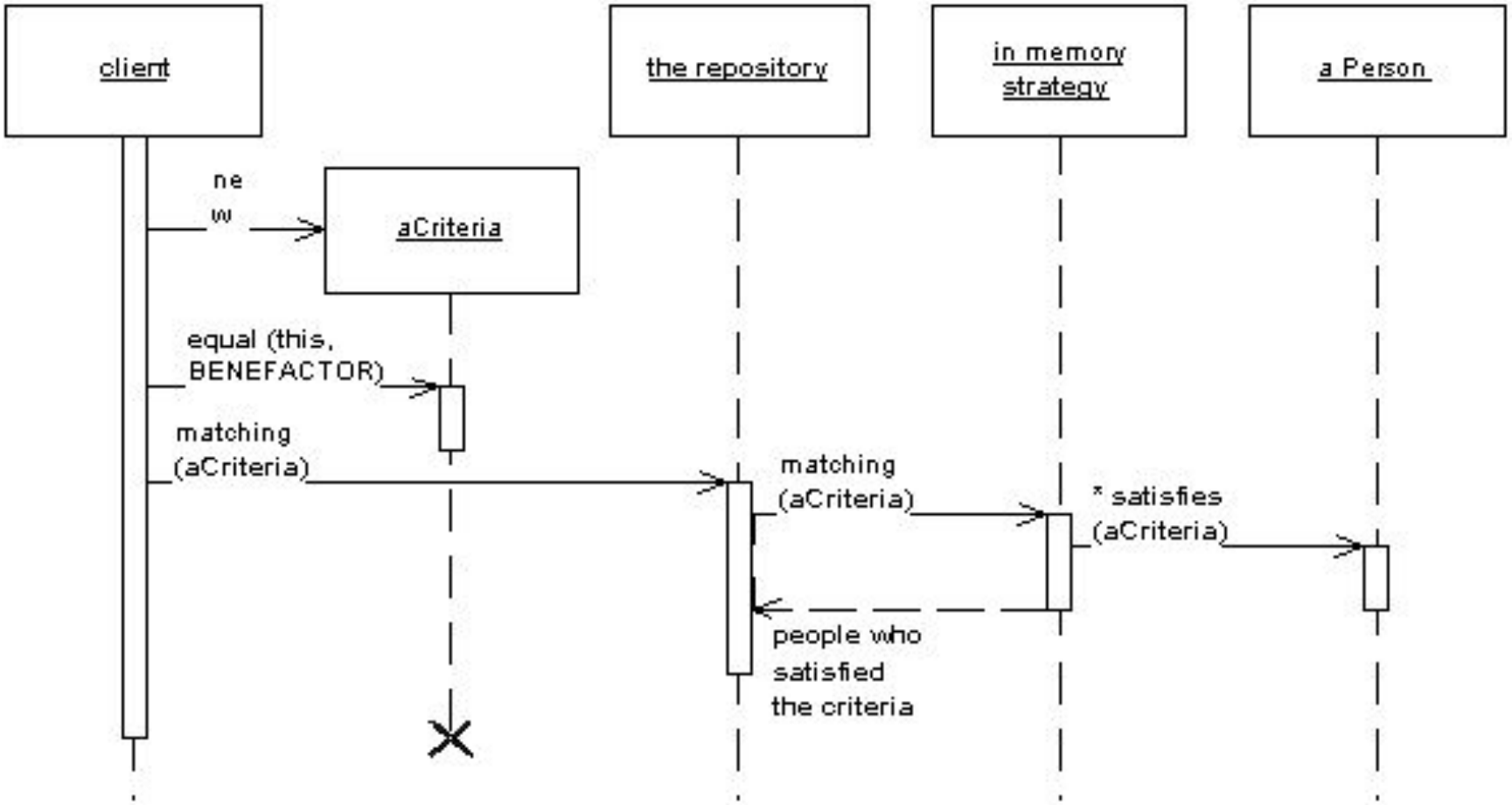
Простым и действенным выходом является использование шаблона Repository. Хранилищу AccountRepository мы оставляем работу с базой данных и получаем «чистый» доменный объект.

```
// создание пользователя
Account account = new Account();
account.Name = "Name";
accountRepository.Save(account);

// загрузка пользователя по Id
account = accountRepository.GetById(1);

// загрузка со связной коллекцией
account = accountRepository.GetById(1, new (Path[]{new
Path<Account>(Account.PrefetchPathRoles)}));
```

UML диаграмма последовательностей для паттерна Repository



Паттерн Repository (Хранилище)

- Repository посредничает между доменом и слоем отображающим данные, используя интерфейс – коллекцию для доступа к объектам домена.
- Паттерн разработан для систем со сложной моделью предметной области где есть выгоды от слоя отображения данных, который изолирует объекты домена от деталей кода доступа к базе данных.
- В таких системах может быть, стоит построить еще один уровень абстракции над отображением слоя, где сосредоточен код запроса.
- Это становится более важно, когда есть большое количество классов доменов или тяжелых запросов.
- В этих случаях, при добавлении этого слоя можно свести к минимуму дублирование логики запроса.

Паттерн Repository (Хранилище)

- Отображение хранилища является посредником между доменом и слоем данных, посредник работает в памяти с коллекцией объектов домена.
- Объекты могут быть добавлены и удалены из Repository, как набор простых объектов, и код сопоставление заложенный в Repository будут осуществлять соответствующие действия за кулисами.
- Концептуально, Repository инкапсулирует набор объектов сохраняет их в хранилище данных и реализует операции над ними, обеспечивая более объектно-ориентированное представление слоя персистентности.
- Repository также поддерживает четкое разделения и односторонней зависимости между доменом и слоями отображения данных.

Валидация данных

Например, проверка введенного адреса эл. почты, длины имени пользователя, сложности пароля и т.п.

Пример

Для валидации объекта возникает первая реализация:

```
public class Product{
   pivate int Price { get; set; }
  public Product(Price price){
          this.set(price);}
   public boolean IsValid()   {
      return Price > 0;    }}

// проверка на валидность
Product product = new Product (100);
boolean isValid = product.IsValid();
```

Такой подход является вполне оправданным в данном случае.
Код простой, тестированию поддается, дублирования логики нет.

Валидация данных

Теперь наш объект Product начал использоваться в некоем **CustomerService**, который считает валидным продукт с ценой больше 100 тыс. рублей.

Что делать?

Придется изменять наш объект продукта, например, таким образом:

```
public class Product{
   public int Price { get; set; }
   public Product(Price price){
            this.set(price);}
     public boolean IsValid(boolean isCustomerService)
   {
      if (isCustomerService == true)
         return Price > 100000;
        return Price > 0;    }}
 // используем объект продукта в новом сервисе
Product product = new Product(100);// { Price = 100 }
boolean  isValid = product.IsValid(true);
```

Решение

Стало очевидно, что при дальнейшем использовании объекта Product логика валидации его данных будет изменяться и усложняться.

Видимо пора отдать ответственность за валидацию данных продукта другому объекту.

Причем надо сделать так, чтобы сам объект продукта не зависел от конкретной реализации его валидатора.

Получаем код:

```
public interface IProductValidator{
   bool IsValid(Product product);}
 public class ProductDefaultValidator : IProductValidator{
   public bool IsValid(Product product)
   {
      return product.Price > 0;
   }}
 public class CustomerServiceProductValidator : IProductValidator{
   public bool IsValid(Product product)
   {
      return product.Price > 100000;
   }}
```

Ур.2.1 Переписать код решения на java. Составить Тесты.

Решение

```
public class Product{
  private readonly IProductValidator validator;
    public Product() : this(new ProductDefaultValidator())  {  }

  public Product(IProductValidator validator)  {
     this.validator = validator;  }

  public int Price { get; set; }

  public bool IsValid()  {
     return validator.IsValid(this);  }
}
 // обычное использование
var product = new Product { Price = 100 };

// используем объект продукта в новом сервисе
var product = new Product (new CustomerServiceProductValidator())
 { Price = 100 };
```

God object

Предел нарушения принципа единственности ответственности – God object.

Этот объект знает и умеет делать все, что только можно.

Например, он делает запросы к базе данных, к файловой системе, общается по протоколам в сеть и содержит бизнес-логику.

В примере приведен объект, который называется ImageHelper:

```
public static class ImageHelper{
   public static void Save(Image image)   {
      // сохранение изображение в файловой системе
   }

   public static int DeleteDuplicates()   {
      // удалить из файловой системы все дублирующиеся изображения и вернуть количество удаленных
   }

   public static Image SetImageAsAccountPicture(Image image, Account account)
   {
      // запрос к базе данных для сохранения ссылки на это изображение для пользователя
   }
```

God object

```
public static Image Resize(Image image, int height, int width)
   {
      // изменение размеров изображения
   }

   public static Image InvertColors(Image image)
   {
      // изменить цвета на изображении
   }

   public static byte[] Download(Url imageUrl)
   {
      // загрузка битового массива с изображением с помощью HTTP запроса
   }

   // и т.п.
}
```

Границы ответственности у него вообще нет.
Он может сохранять в базу данных, причем знает правила назначения изображений пользователям.
Может скачивать изображения.
Знает, как хранятся файлы изображений и может работать с файловой системой.

Решение проблемы

Каждая ответственность этого класса ведет к его потенциальному изменению.

Этот класс будет очень часто менять свое поведение, что затруднит его тестирование и тестирование компонентов, которые его используют.

Такой подход снизит работоспособность системы и повысит стоимость ее сопровождения.

Решением является разделить этот класс по принципу единственности ответственности: один класс на одну ответственность:

```
public static class ImageFileManager
{
   public static void Save(Image image)
   {
      // сохранение изображение в файловой системе
   }

   public static int DeleteDuplicates()
   {
      // удалить из файловой системы все дублирующиеся изображения и вернуть количество удаленных
   }
}
```

Решение проблемы

```
public static class ImageRepository{
   public static Image SetImageAsAccountPicture(Image image, Account account)   {
      // запрос к базе данных для сохранения ссылки на это изображение для пользователя
   }}

public static class Graphics{
     public static Image Resize(Image image, int height, int width)   {
      // изменение размеров изображения
   }
     public static Image InvertColors(Image image)   {
      // изменить цвета на изображении
   }
}
  public static class ImageHttpManager{
   public static byte[] Download(Url imageUrl)   {
      // загрузка битового массива с изображением с помощью HTTP запроса
  }}
```

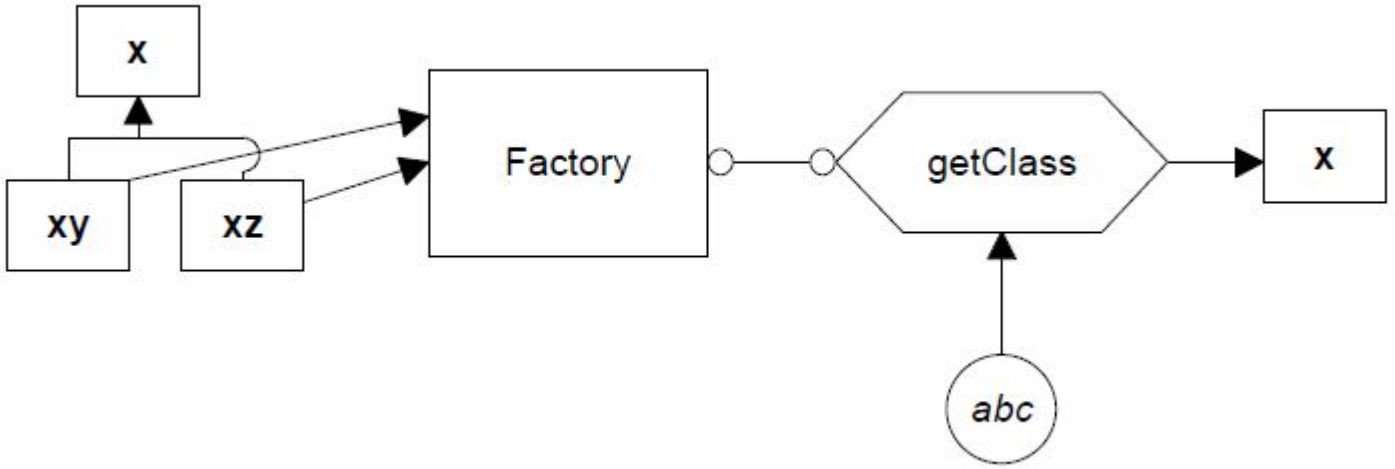
Порождающие шаблоны

- Фабрика - способ создания объекта одного из нескольких возможных классов, основываясь на представленных данных.
- Абстрактная фабрика - способ создания набора объектов классов, принадлежащих нескольким семействам (интерфейсам).
- Строитель - позволяет собрать сложный объект шаг за шагом, отделяя алгоритм сборки объекта от его представления.
- Прототип - реализация создания новых объектов через клонирование объекта-прототипа.

Фабрика

Обычно возможные классы наследуют один общий класс либо реализуют общий интерфейс и различаются между собой реализацией.

Фабрика



Классическая схема фабричного метода представлена выше.
Здесь X - базовый класс, а может быть даже интерфейс.
Классы XY и XZ по-разному реализуют интерфейс X.
Для выбора конкретной реализации используется класс Factory, где в методе getClass происходит анализ внешних параметров abc, и, основываясь на этих данных, метод возвращает объект класса X используя реализацию XY либо XZ.
Таким образом, внешняя программа не имеет представления о том, объект какого именно класса - XY или XZ - ей вернули, поскольку оба эти класса реализуют общий интерфейс X, а значит, вызов их внешних методов ничем не отличается.

Когда используется и в чем его достоинства

- Позволяет связать параллельные иерархии классов
- Когда конкретная реализация объектов данного класса X задается с помощью подклассов
- Когда заранее не известны конкретные реализации данного класса X
- Когда требуется использовать интерфейс X вместо конкретных реализующих классов
- Для скрытия конкретных классов от клиента
- Фабричные методы могут быть параметризованы
- Позволяет следовать общепринятым правилам именования, помогает реорганизовать код при необходимости

Пример Классическая фабрика(фабричный метод)

В программе создается объект некоего клиента, а затем обращаемся к его методам.

При этом существует несколько реализаций клиента - для разных операционных систем.

В таком случае удобно написать некий интерфейс **Client**, определяющий набор методов, к которым мы хотим обращаться. А конкретную реализацию этих методов для разных ОС определить в классах **ClientLinuxImpl** и **ClientWinImpl**.

Затем создать класс-фабрику, который будет принимать, **id** операционной системы и возвращать объект соответствующего клиента.

Интерфейс **Client** задает наличие всего одного метода - **getTargetOS**, который, очевидно, возвращает название операционной системы, для которой предназначен клиент.

Интерфейс Client

задает наличие всего одного метода - getTargetOS, который, возвращает название операционной системы, для которой предназначен клиент:

```
package org.test.factory.ex;
public interface Client {
public String getTargetOS();
}
```

Реализации клиентов

Клиент Linux:

```
package org.test.factory.ex;
public class ClientLinuxImpl implements Client{
@Override
public String getTargetOS() {
return "Gentoo Linux";
}
}
```

Клиент Windows

```
package org.test.factory.ex;
public class ClientWinImpl implements Client{
@Override
public String getTargetOS() {
return "Windows Vista";
}
}
```

Код фабрики

в которой метод getClient получает id операционной системы и возвращает объект соответствующего класса, либо null если id не соответсвует ни одной ОС:

```
package org.test.factory.ex;
public class Factory {
public Client getClient(String currentOS){
if(currentOS.equals("win"))
return new ClientWinImpl();
else if(currentOS.equals("linux"))
return new ClientLinuxImpl();
return null;
}
}
```

Для теста всех примеров класс RunTestFactory

```
package org.test.factory.ex;
public class RunTestFactory {
public static void main(String[] args){
log(">Start test for Factory pattern"); //тут будут вызываться тесты
}
public static void log(String msg){ //А это для вывода сообщений в консоль, для удобства
if(msg==null)
msg = "null";
System.out.println(msg);
}
}
```

класс RunTestFactoryEx

```
package org.test.factory.ex;
import package org.test.factory.ex.RunTestFactory;
public class RunTestFactoryEx extends RunTestFactory{
public static void main(String[] args){
log("----> Start Example 1");

String currentOS = args[0];//получаем id ОС

Factory factory = new Factory(); //Инициализируем фабрику

log(" OS id: [" + currentOS + "], creating client");

Client client = factory.getClient(currentOS);//Инициализируем клиент типа Client

if(client!=null){//Фабрика нашла подходящий клиент, хорошо
log(" Client created for OS:");
log(client.getTargetOS());//Конкретная реализация метода здесь не видна, что дает нам свободу
}else{
log(" No client!");
}
log("<---- Finish Example "); } }
```

Паттерн Строитель(Builder)

- Как известно, шаблон Фабрика создает объект одного из нескольких подклассов в зависимости от полученных параметров.
- Но часто объекты могут быть сложными и их создание требует выполнения целого набора операций по их "сборке" из простых объектов.
- При этом может потребоваться использовать разные реализации этих простых объектов или алгоритм сборки целевого объекта может быть различным.
- Для отделения построения объекта от деталей его конструкции применяется шаблон **Строитель (Builder).**
- В шаблоне Строитель мы инкапсулируем, скрываем алгоритм сборки объекта и выбора реализаций его частей внутри класса конкретного строителя, реализующего базовый интерфейс строителя.
- В зависимости от того, какой именно алгоритм сборки нам требуется, мы выбираем конкретного строителя.
- Инициализация строителя, вызов его методов и получение целевого объекта обычно описывают в классе Директора (Director).

Когда используется Builder и в чем его достоинства

- Позволяет изменять внутреннее представление конечного продукта.
- Скрывает детали строительства.
- Каждый конкретный строитель независим от других строителей и остальной части кода: проще добавлять новых строителей, выше модульность вашего приложения.
- Поскольку строитель собирает конечный продукт шаг за шагом, вы лучше контролируете каждый из продуктов.
- Строитель похож на абстрактную фабрику тем, что также возвращает классы, использующие наборы методов и объектов.
- Но главное отличие строителя от абстрактной фабрики в том, что строитель собирает объект поэтапно.

Пример построения дома House

Дом может быть одноэтажным или двухэтажным:

```
package org. builder;
public class House {
 private int stores; //количество этажей
public House(){
stores = 0; //дом не построен, 0 этажей
}
 public void setStores(int newLevel){ //в процессе строительства мы будем увеличивать количество этажей
log("setting stores to: " + newLevel);
stores = newLevel; }
 public int getStores(){
 return stores; }
public void buildBase(){//Построить фундамент
 log("Build base"); }

 public void buildWalls(Window window){// Построить стены
       log("Build walls with window: " + window.getWindowType());     }
private void log(String msg){ System.out.println(msg); }

public void buildFloor(){//Построить пол
log("Build floor"); }
public void buildRoof(){//Построить крышу
 log("Build roof"); }
  }
```

Методы класса House

House -класс с набором методов, каждый из которых может быть использован в процессе сборки объекта.

Нам понадобится метод для строительства фундамента, а также стен, крыши и т.п.

Метод для строительства стен требует передачи объекта типа Window, и для разных типов домов мы должны использовать разные типы окон, реализующих базовый интерфейс Window:

```
package org.builder;
public interface Window {
    public String getWindowType(); }
```

Это может быть простое окно SimpleWindow:

```
package org.builder;
public class SimpleWindow implements Window{
 @Override public String getWindowType() {
 return "Simple window"; } }
```

Или сложное ComplexWindow:

```
package org. builder;
public class ComplexWindow implements Window{
@Override public String getWindowType() { return "Complex window"; } }
```

Интерфейс для строителя в абстрактном классе Builder

```
package org.builder;
public abstract class Builder {
 protected House house;
public abstract House buildHouse();
}
```

Алгоритм сборки дома внутри метода buildHouse.

Пишем строителя простого одноэтажного дома RanchBuilderImpl:

```
package org.builder;
 public class RanchBuilderImpl extends Builder{
public RanchBuilderImpl(){
house = new House(); }
 @Override
public House buildHouse() {
 house.buildBase(); //фундамент
house.buildFloor(); //пол
 Window simpleWindow = new SimpleWindow(); //инициализируем простое окно
house.buildWalls(simpleWindow); //строим стены
house.buildRoof(); //крыша
house.setStores(1); //готов 1 этаж
return house; //Возвращаем собранный объект
} }
```

Здесь мы строим одноэтажный дом, используя именно то окно, которое нам нужно сейчас и вызывая методы согласно конкретному алгоритму.

Строитель двухэтажного дома MultiStoreyBuilderImpl

```
package org. builder;
public class MultiStoreyBuilderImpl extends Builder{
 public MultiStoreyBuilderImpl(){
house = new House(); }
 @Override
 public House buildHouse() {
house.buildBase();//фундамент
 house.buildFloor();//пол 1 этажа
Window complexWindow = new ComplexWindow(); house.buildWalls(complexWindow); //стены
 house.buildFloor(); //пол второго этажа
house.setStores(1); //1 этаж готов
complexWindow = new ComplexWindow();
house.buildWalls(complexWindow); //стены
house.buildRoof(); //крыша
house.setStores(2); //2 этаж готов
return house; //возвращаем готовый объект
 } }
```

Здесь мы используем другую реализацию окна и другой алгоритм.

Директор для управления строителем

```
package org.builder;
public class Director {
private Builder builder;
public Director(int level){ if(level > 1)//Выбираем строителя
builder = new MultiStoreyBuilderImpl();
else builder = new RanchBuilderImpl(); }
public House buildHouse(){
return builder.buildHouse(); //заставляем строителя строить нам дом
} }
```

В зависимости от количества этажей, мы инициализируем разных строителей и далее обращаемся к ним.

Конечно, вариантов директора можно написать несколько, логику можно из конструктора вообще убрать.

К тому же выбор конкретного строителя можно вынести в отдельную фабрику.

Тест

```
package org.builder;
import java.util.logging.Level;
 import java.util.logging.Logger;
 public class RunTest {
 public static void main(String[] args){
 log("building ranch house--->");
 Director director = new Director(1);
House house = director.buildHouse();
 log("building multistorey house--->");
 director = new Director(2);
 house = director.buildHouse(); }
private static void log(String msg){
 System.out.println(msg);
} }
```

Литература

1. **Джимми Нильссон Применение DDD и шаблонов проектирования. Проблемно-ориентированное проектирование приложений с примерами на C# и .NET, 2008 OZON.ru**
2. http://www.technerium.ru/node/23