【软件构造】第六章第二节 可维护的设计模式

第六章第二节 可维护的设计模式

Outline

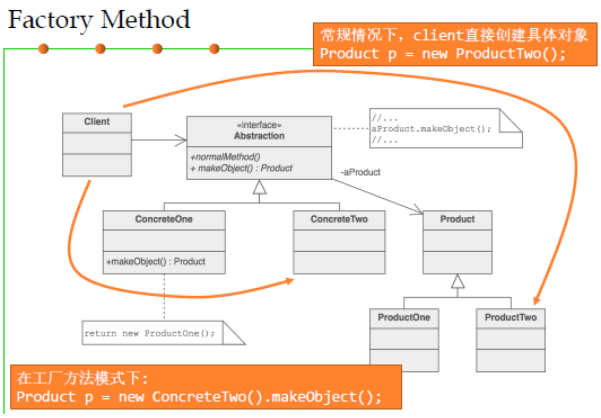
- 创造性模式：Creational patterns
  - 工厂模式（Factory Pattern）
  - 抽象工厂模式（Abstract Factory Pattern）
  - 建造者模式（Builder Pattern）
- 结构化模式：Structural patterns
  - 桥接模式（Bridge Pattern）
  - 代理模式（Proxy Pattern）
  - 组合模式（Composite Pattern）
- 行为化模式：Behavioral patterns
  - 中介者模式（Mediator Pattern）
  - 观察者模式（Observer Pattern）
  - 访问者模式（Visitor Pattern）
  - 责任链模式（Chain of Responsibility Pattern）
  - 命令模式（Command Pattern）

Notes：

创造性模式：**Creational patterns**

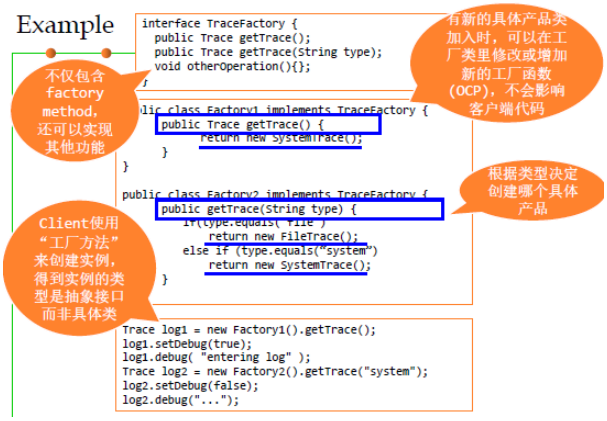
【工厂模式（**Factory Pattern**）】

- 定义：工厂方法模式也被称为虚拟构造器。当client不知道要创建哪个具体类的实例，或者不想在client代码中指明要具体创建的实例时，用工厂方法。
- 意图：定义一个用于创建对象的接口，让其子类来决定实例化哪一个类，从而使一个类的实例化延迟到其子类。
- 主要解决：主要解决接口选择的问题。
- 应用实例： 您需要一辆汽车，可以直接从工厂里面提货，而不用去管这辆汽车是怎么做出来的，以及这个汽车里面的具体实现。
- 优点：
  - 一个调用者想创建一个对象，只要知道其名称就可以了。
  - 扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。
  - 屏蔽产品的具体实现，调用者只关心产品的接口。
- 缺点：每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。这并不是什么好事。
- 满足**OCP**（Open-Closed Principle）：—对扩展的开放，对修改已有代码的封闭
- 模式：



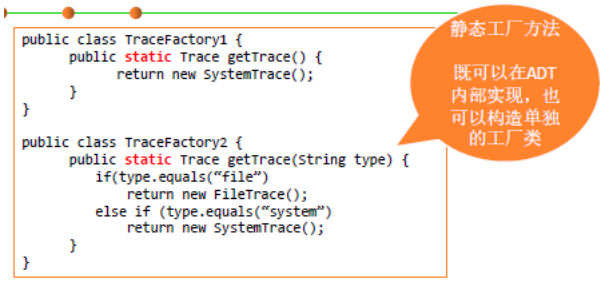
- 例子：

非静态方法：




```
interface TraceFactory {
  public Trace getTrace();
  public Trace getTrace(String type);
  void otherOperation(){};
}

public class Factory1 implements TraceFactory {
  public Trace getTrace() {
      return new SystemTrace();
  }
}

public class Factory2 implements TraceFactory {
  public Trace getTrace(String type) {
     if(type.equals("file")
        return new FileTrace();
     else if (type.equals("system")
        return new SystemTrace();
  }

Trace log1 = new Factory1().getTrace();
log1.setDebug(true);
log1.debug( "entering log" );
Trace log2 = new Factory2().getTrace("system");
log2.setDebug(false);
log2.debug("...");
```

静态方法：




```
public class TraceFactory1 {
      public static Trace getTrace() {
           return new SystemTrace();
      }
}

public class TraceFactory2 {
      public static Trace getTrace(String type) {
         if(type.equals("file")
            return new FileTrace();
         else if (type.equals("system")
            return new SystemTrace();
      }
}
```

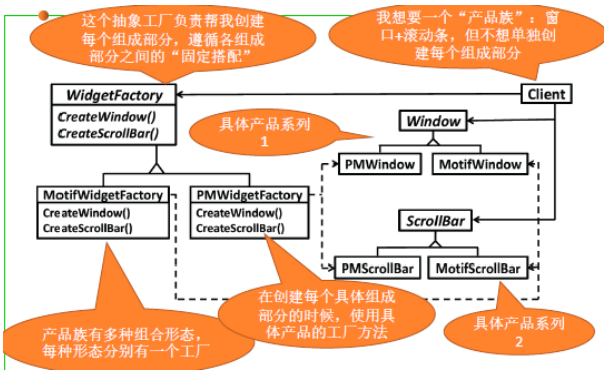
```
//... some code ...
Trace log1 = TraceFactory1.getTrace();
log1.setDebug(true);
log1.debug( "entering log" );

Trace log2 = TraceFactory2.getTrace("system");
log1.setDebug(true);
log2.debug("...");
```

【抽象工厂模式（Abstract Factory）】

- 定义：抽象工厂模式（Abstract Factory Pattern）是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。
- 在抽象工厂模式中，接口是负责创建一个相关对象的工厂，不需要显式指定它们的类。每个生成的工厂都能按照工厂模式提供对象。
- 方法：提供接口以创建一组相关/相互依赖的对象，但不需要指明其具体类。
- 用途：系统的产品有多于一个的产品族，而系统只消费其中某一族的产品时使用。例：①当一个UI，包含多个窗口控件，这些控件在不同的OS中实现不同。②当一个仓库类，要控制多个设备，这些设备的制造商各有不同，控制接口有差异
- 优点：当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

- 缺点：产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的 Creator 里加代码，又要在具体的里面加代码。
- 使用场景： 1、QQ 换皮肤，一整套一起换。 2、生成不同操作系统的程序。
- 以下面窗口滚动条为例：

客户端想要一个产品，由窗口和滚动条组成。于是可以交给一个抽象工厂来做，这个工厂负责将产品的组件组装起来成一个完整的产品。不同的产品继承这个抽象工厂接口，实现自己的工厂方法。



下面是具体的实现

Example

抽象产品接口和具体产品类

```
//AbstractProduct
public interface Window{
  public void setTitle(String s);
  public void repaint();
  public void addScrollbar(…);
}
//ConcreteProductA1
public class PMWindow
           implements Window{
  public void setTitle(){…}
  public void repaint(){…}
}

//ConcreteProductA2
public class MotifWindow
           implements Window{
  public void setTitle(){…}
  public void repaint(){…}
}
```

抽象工厂接口和具体工厂类

```
//AbstractFactory
public interface AbstractWidgetFactory{
  public Window createWindow();
  public Scrollbar createScrollbar();
}
//ConcreteFactory1
public class WidgetFactory1{
  public Window createWindow(){
    return new MSWindow();
  }
  public Scrollbar createScrollbar(){A}
}
//ConcreteFactory2
public class WidgetFactory2{
  public Window createWindow(){
    return new MotifWindow();
  }
  public Scrollbar createScrollbar(){B}
}
```

第一个具体产品的工厂方法

第二个具体产品的工厂方法

Example

辅助类

Delagate到抽象工厂类

```
public class GUIBuilder{
  public void buildWindow(AbstractWidgetFactory widgetFactory){
    Window window = widgetFactory.createWindow();
    Scrollbar scrollbar = widgetFactory.createScrollbar();
    window.setTitle("New Window");
    window.addScrollbar(scrollbar);
  }
}

Client
GUIBuilder builder = new GUIBuilder();
AbstractWidgetFactory widgetFactory = null;

if(“Motif”)
   widgetFactory  = new WidgetFactory2();
else
   widgetFactory  = new WidgetFactory1();

builder.buildWindow(widgetFactory);
```

使用抽象工厂类分别创建两个具体产品

把创建的两个具体产品实例组合起来（模式之外）

定义抽象工厂接口的实例

根据要创建的“组合产品”的类型，构建不同的抽象工厂子类

具体构建

抽象工厂类型创建的不是一个完整产品，而是“产品族”（遵循 固定搭配规则的多类产品的实例），得到的结果是：多个不同产品的 object，各产品创建过程对client可见，但“搭配”不能改变。

本质上，Abstract Factory是把多类产品的factory method组合在一起。

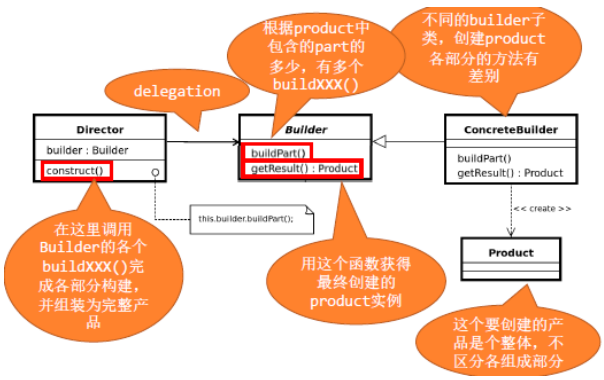
【建造者模式（Builder Pattern）】

- 定义：建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象。该 Builder 类是独立于其他对象的。
- 方法：创建复杂对象，包含多个组成部分

- 意图：将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。
- 关键代码：建造者：创建和提供实例，导演：管理建造出来的实例的依赖关系。
- 应用实例： 1、去肯德基，汉堡、可乐、薯条、炸鸡翅等是不变的，而其组合是经常变化的，生成出所谓的"套餐"。

2、JAVA 中的 StringBuilder。

- 优点： 1、建造者独立，易扩展。 2、便于控制细节风险。
- 缺点： 1、产品必须有共同点，范围有限制。 2、如内部变化复杂，会有很多的建造类。
- 模式图：



- 例子： 我们需要一个**Pizza**的产品，该**Pizza**产品的**part**是以三个属性的形式体现，其**builder**就相当于给三个属性赋值（也可更复杂）。

```
/* "Product" */
class Pizza {
    private String dough = "";
    private String sauce = "";
    private String topping = "";

    public void setDough(String dough) {
        this.dough = dough;
    }
    public void setSauce(String sauce) {
        this.sauce = sauce;
    }
    public void setTopping(String topping) {
        this.topping = topping;
    }
}
```

```
/* "Abstract Builder" */
abstract class PizzaBuilder {
    protected Pizza pizza;
    public Pizza getPizza() {
        return pizza;
    }
    public void createNewPizzaProduct() {
        pizza = new Pizza();
    }
    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
}
```

Builder抽象类有三个抽象方法，分别创建三个part

```
/* "ConcreteBuilder 1" */
class SpicyPizzaBuilder extends PizzaBuilder {
    public void buildDough() {
        pizza.setDough("pan baked");
    }
    public void buildSauce() {
        pizza.setSauce("hot");
    }
    public void buildTopping() {
        pizza.setTopping("pepperoni+salami");
    }
}
```

具体的builder子类中，override三个抽象方法，分别构建三个parts

两个不同的builder子类，构建三个parts的内容有差异

```
/* "ConcreteBuilder 2" */
class HawaiianPizzaBuilder extends PizzaBuilder {
    public void buildDough() {
        pizza.setDough("cross");
    }
    public void buildSauce() {
        pizza.setSauce("mild");
    }
    public void buildTopping() {
        pizza.setTopping("ham+pineapple");
    }
}
```

Example

Director中定义要delegate的builder

```
/* "Director" */
class Waiter {
    private PizzaBuilder pizzaBuilder;
    public void setPizzaBuilder(PizzaBuilder pb) {
        pizzaBuilder = pb;
    }
    public Pizza getPizza() {
        return pizzaBuilder.getPizza();
    }
    public void constructPizza() {
        pizzaBuilder.createNewPizzaProduct();
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
}
```

得到最终builder的结果

在这里具体构建各部分

```
/* A customer ordering a pizza. */
public class PizzaBuilderDemo {
    public static void main(String[] args) {
        Waiter waiter = new Waiter();
        PizzaBuilder hawaiianPizzabuilder = new HawaiianPizzaBuilder();
        PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

        waiter.setPizzaBuilder( hawaiianPizzabuilder );
        waiter.constructPizza();
        Pizza pizza = waiter.getPizza();
    }
}
```

Client使用不同的builder子类创建不同的产品

建立waiter和builder之间的delegation关系

Bingo！构建细节完全对client隐藏

比较：工厂抽象模式（**Factory method**）和构造器模式（**Builder method**）

Abstract Factory创建的不是一个完整产品，而是“产品族”（遵循固定搭配规则的多类产品实例），得到的结果是：多个不同产品的实例object，各产品创建过程对client可见，但“搭配”不能改变。

Builder Factory创建的是一个完整的产品，有多个部分组成，client不需了解每个部分是怎么创建、各个部分怎么组合，最终得到一个产品的完整 object 。

比较：模板方法模式（**Template method**）和构造器模式（**Builder method**）

- Template Method：a behavioral pattern 目标是为了复用算法的公共结构（次序）。
  - 定义了一个操作中算法的骨架(steps)，而将具体步骤的实现延迟到子类中， 从而复用算法的结构并可重新定义算法某些特定步骤的实现逻辑。
  - 复用算法骨架，强调步骤的次序
  - 子类override算法步骤
- Builder Factory：a creationalpattern 目标是“创建复杂对象”，灵活扩展
  - 将一个复杂对象的构造方法与对象内部的具体表示分离出来，同样的构造方法可以建立不同的表现。
  - 不强调复杂对象内部各部分的“次序”
  - 子类override复杂对象内部各部分的“创建”
  - 适应变化：通过派生新的builder来构造新的对象（即新的内部表示），OCP

结构化模式：**Structual patterns**

【桥接模式（**Bridge Pattern**）】

【代理模式（**Proxy Pattern**）】

【组合模式（**Composite Pattern**）】

行为化模式：**Behavioral patterns**

【中介者模式（**Mediator Pattern**）】

【观察者模式（**Observer Pattern**）】

【访问者模式（**Visitor Pattern**）】

【责任链模式（**Chain of Responsibility Pattern**）】

【命令模式（**Command Pattern**）】