## Gang of Four Design Patterns

### Creational Design Patterns

* [Abstract Factory](https://springframework.guru/gang-of-four-design-patterns/abstract-factory-design-pattern/). Allows for the creation of objects without specifying their concrete type.
* [Builder](https://springframework.guru/gang-of-four-design-patterns/builder-pattern/). Used to create complex objects.
* [Factory Method](https://springframework.guru/gang-of-four-design-patterns/factory-method-design-pattern/). Creates objects without specifying the exact class to create.
* [Prototype](https://springframework.guru/gang-of-four-design-patterns/prototype-pattern/). Creates a new object from an existing object.
* [Singleton](https://springframework.guru/gang-of-four-design-patterns/singleton-design-pattern/). Ensures only one instance of an object is created.

### Structural Design Patterns

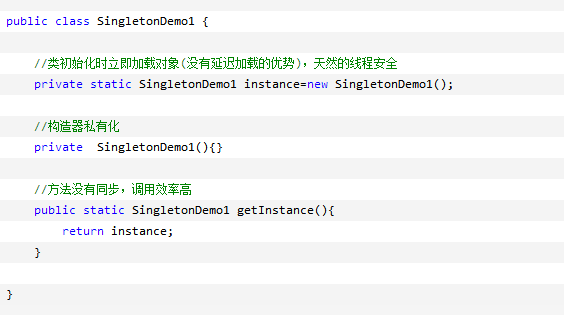
* [Adapter](https://springframework.guru/gang-of-four-design-patterns/adapter-pattern/). Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.
* [Bridge](https://springframework.guru/gang-of-four-design-patterns/bridge-pattern/). Decouples an abstraction so two classes can vary independently.
* [Composite](https://springframework.guru/gang-of-four-design-patterns/composite-pattern/). Takes a group of objects into a single object.
* [Decorator](https://springframework.guru/gang-of-four-design-patterns/decorator-pattern/). Allows for an object’s behavior to be extended dynamically at run time.
* [Facade](https://springframework.guru/gang-of-four-design-patterns/facade-pattern/). Provides a simple interface to a more complex underlying object.
* [Flyweight](https://springframework.guru/gang-of-four-design-patterns/flyweight-pattern/). Reduces the cost of complex object models.
* [Proxy](https://springframework.guru/gang-of-four-design-patterns/proxy-pattern/). Provides a placeholder interface to an underlying object to control access, reduce cost, or reduce complexity.

### Behavior Design Patterns

* [Chain of Responsibility](https://springframework.guru/gang-of-four-design-patterns/chain-of-responsibility-pattern/). Delegates commands to a chain of processing objects.
* [Command](https://springframework.guru/gang-of-four-design-patterns/command-pattern/). Creates objects which encapsulate actions and parameters.
* [Interpreter](https://springframework.guru/gang-of-four-design-patterns/interpreter-pattern/). Implements a specialized language.
* [Iterator](https://springframework.guru/gang-of-four-design-patterns/iterator-pattern/). Accesses the elements of an object sequentially without exposing its underlying representation.
* [Mediator](https://springframework.guru/gang-of-four-design-patterns/mediator-pattern/). Allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
* [Memento](https://springframework.guru/gang-of-four-design-patterns/memento-pattern/). Provides the ability to restore an object to its previous state.
* [Observer](https://springframework.guru/gang-of-four-design-patterns/observer-pattern/). Is a publish/subscribe pattern which allows a number of observer objects to see an event.
* [State](https://springframework.guru/gang-of-four-design-patterns/state-pattern/). Allows an object to alter its behavior when its internal state changes.
* [Strategy](https://springframework.guru/gang-of-four-design-patterns/strategy-pattern/). Allows one of a family of algorithms to be selected on-the-fly at run-time.
* [Template Method](https://springframework.guru/gang-of-four-design-patterns/template-method-pattern/). Defines the skeleton of an algorithm as an abstract class, allowing its sub-classes to provide concrete behavior.
* [Vistor](https://springframework.guru/gang-of-four-design-patterns/visitor-pattern/). Separates an algorithm from an object structure by moving the hierarchy of methods into one object.

# 单例模式

（1）饿汉式

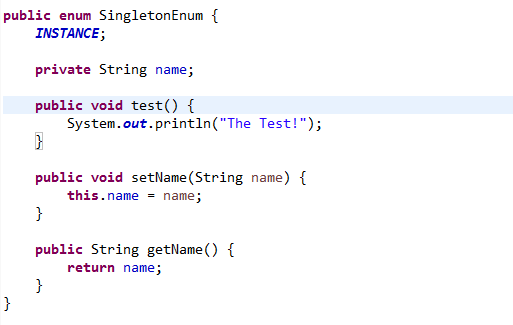


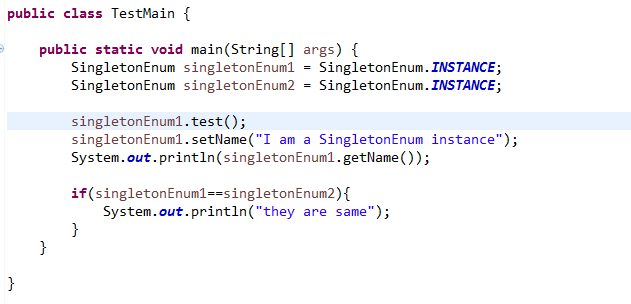
(2)懒汉式

防止反射和反序列化漏洞



（3）枚举式



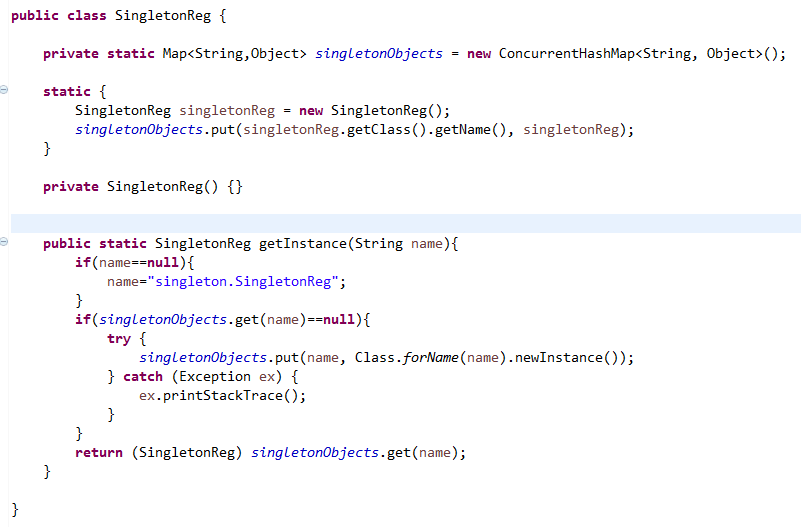


首先，我们都知道enum是由class实现的，换言之，enum可以实现很多class的内容，包括可以有member和member function，这也是我们可以用enum作为一个类来实现单例的基础。另外，由于enum是通过继承了Enum类实现的，enum结构不能够作为子类继承其他类，但是可以用来实现接口。此外，enum类也不能够被继承，在反编译中，我们会发现该类是final的。

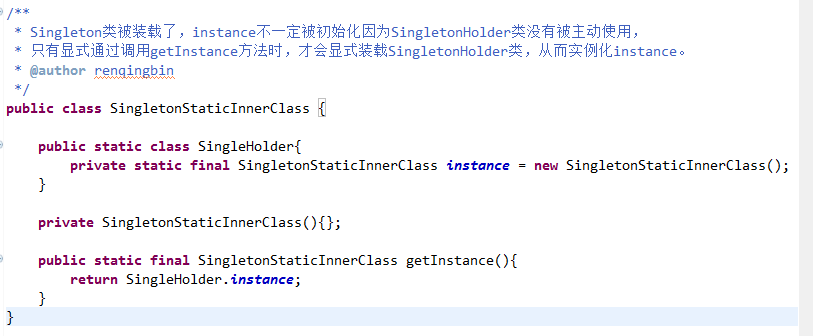
其次，enum有且仅有private的构造器，防止外部的额外构造，这恰好和单例模式吻合，也为保证单例性做了一个铺垫。这里展开说下这个private构造器，如果我们不去手写构造器，则会有一个默认的空参构造器，我们也可以通过给枚举变量参量来实现类的初始化。

（4）注册式

在 Spring 中，单例模式是十分常用的，因为 Spring Bean 默认是单例模式，即只有一个共享的实例存在，所有对这个 Bean 的请求都会返回这个唯一的实例。

****

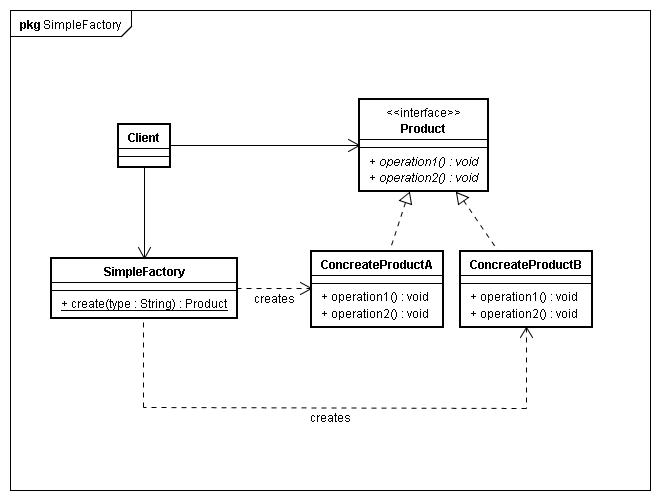
**（5）静态内部类式**

****

**简单工厂模式（Simple Factory Pattern）**

又称静态工厂方法模式（Static Factory Method）

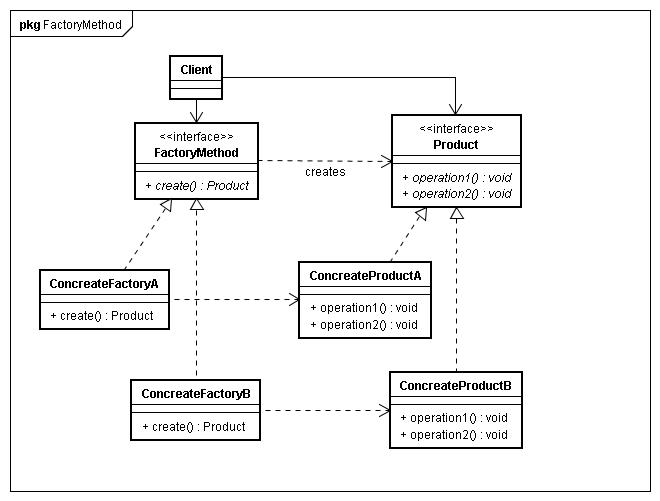
专门定义一个类对实现了同一接口的一些类进行实例的创建。实质上就是由一个工厂类根据传入的参数，动态的决定应该创建哪个具体产品类的实例。



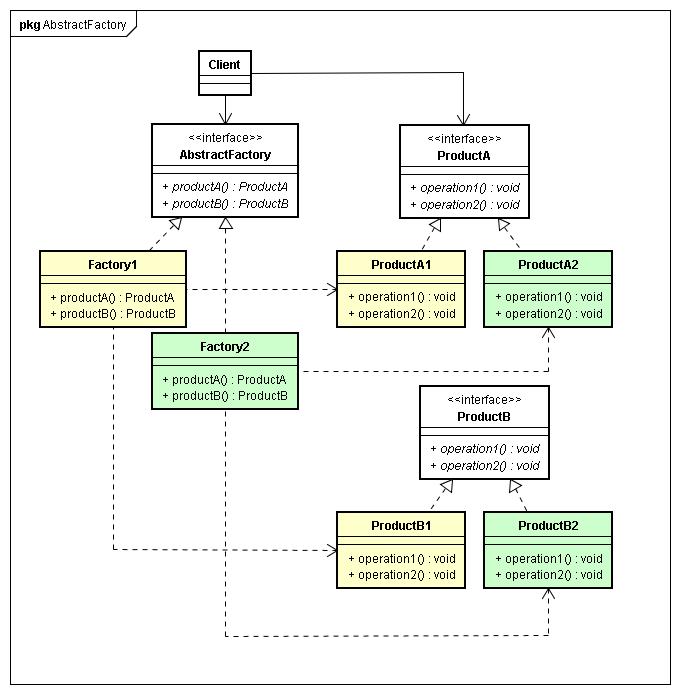
**工厂方法模式（Factory Method Pattern）**

又称工厂模式或多态工厂（Polymorphic Factory）模式

父类负责定义创建对象的公共接口，而子类负责生成具体的对象，这样的目的是将类的实例化操作延迟到子类中去完成，即由子类来决定究竟应该实例化哪个类。



**抽象工厂模式（Abstract Factory Pattern）**



# 工厂方法模式（Factory Method Design Pattern）

## 什么是工厂方法模式（What is the Factory Method Pattern? ）

In Java applications, you might be often using the new operator to create an object of a class. This is often fine for small Java programs. But when you work on large scale enterprise class applications, the amount of code to create objects will gradually increase and will become scattered across the application. If class names are hard coded in such code, the complexities of managing the code will keep increasing as you add new classes to the application. To address such concerns, you can use the factory method pattern. This pattern is a classic [Gang of Four](http://springframework.guru/gang-of-four-design-patterns/) creational design pattern that is concerned with the creation of objects in an application. As the name suggests, the factory method pattern makes use of classes that acts as factories to create objects. This pattern favors method invocation instead of making direct constructor calls to create objects.

In the factory method pattern, you provide an interface, which can be a [Java interface](https://docs.oracle.com/javase/tutorial/java/concepts/interface.html) or an [abstract class](https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html) to create objects. A factory method in the interface defers the object creation to one or more concrete subclasses at run time. The subclasses implement the factory method to select the class whose objects need to be created.

在java程序中，你会经常使用new操作来创建一个对象。对于小的java程序这通常是可以的。但是当你开发大型企业级应用程序时，大量创建对象的代码会逐渐增加，并且在应用程序中分散。如果类名在这些代码中是硬编码，那么在向应用程序添加新类时，管理代码的复杂性将不断增加。为了解决这些问题，可以使用工厂方法模式。

这个模式是经典的创建型设计模式，它关注的是在应用程序中的对象的创建。顾名思义，工厂方法模式使用类作为工厂来创建对象。此模式支持方法调用而不是直接调用构造函数来创建对象。

在工厂方法模式中，提供一个接口来创建对象，可能是java接口或者是一个抽象类。接口中的工厂方法在运行时将对象创建推迟到一个或多个具体子类。子类实现工厂方法来选择需要创建对象的类。

## 参与者（Participants in the Factory Method Pattern）

To understand how the factory method pattern works, consider a pizza store that creates pizzas. In an application, you can model the pizzas as concrete Pizza objects, such as CheesePizza, PepperoniPizza, and VeggiePizza. Just as in any pizza store, where a customer can only order a pizza and not make it, in the application you can create an abstract class, say BasePizzaFactory with a factory method to create a pizza.

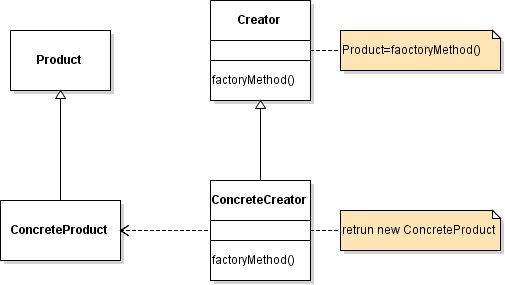
为了理解工厂方法模式是如何工作的，考虑一个做披萨的披萨店。在应用中，你可以用Pizza对象模拟披萨店，比如CheesePizza、PepperoniPizza、VeggiePizza。正如在任何披萨店，在这里客户只能订购一个披萨而不是制作，在应用里你可以创建一个抽象类，就是BasePizzaFactory 类，它带有一个工厂方法来创建Pizza。

You can then create a subclass of BasePizzaFactory, called PizzaFactory to implement the factory method. In the factory method, you can create and return a proper Pizza object. The object returned will be the proper subtype, and configured for use in your code.

The components of the factory method pattern in the context of the pizza store can be summarized as:

你可以创建一个BasePizzaFactory的子类，叫做PizzaFactory 来实现这个工厂方法。在这个工厂方法中，你可以创建并且返回一个特有的Pizza对象。返回的对象是特有的子类型，配置在你的代码中使用。在披萨店场景中工厂方法模式的组件可以概括为：

* Product (Pizza): 是一个接口或者抽象类，通过工厂方法它的子类被实例化
* ConcreteProduct (CheesePizza, PepperoniPizza, and VeggiePizza): 继承或者实现了Product 类.工厂方法实例化这些子类。
* Creator (BasePizzaFactory): 一个声明了工厂方法的接口或者抽象类，它返回了Product 类型的对象。
* ConcreteCreator (PizzaFactory): 一个实现了工厂方法的具体类，它创建并且返回一个具体ConcreteProduct对象到客户端。
* Client: 要求创建者生产一个Product.



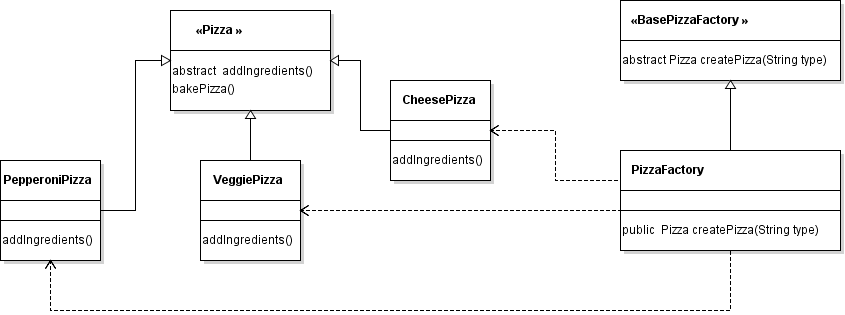
* Product (Pizza): Is an interface or an abstract class whose subclasses are instantiated by the factory method.
* ConcreteProduct (CheesePizza, PepperoniPizza, and VeggiePizza): Are the concrete subclasses that implement/extend Product. The factory method instantiates these subclasses.
* Creator (BasePizzaFactory): Is an interface or an abstract class that declares the factory method, which returns an object of type Product.
* ConcreteCreator (PizzaFactory): Is a concrete class that implements the factory method to create and return a ConcreteProduct to Client.
* Client: Asks the Creator for a Product.

A Client that requires a ConcreteProduct does not create any object but instead asks the Creator for it. The ConcreteCreator implements the factory method to create the object transparently from the Client. As a result the Client is not required to be aware of any ConcreteProduct and how they are created. This approach advocates the Object Oriented Programming principle “Program to an interface, not an implementation“, which leads to [polymorphism](http://springframework.guru/polymorphism-java/), a key feature of object-oriented programming. In addition, as object creation is centralized in the ConcreteCreator, any changes made to a Product or any ConcreteProduct does not affect the Client.

一个客户端要求一个具体的产品不需要创建任何对象，而是取请求创建者。ConcreteCreator实现了工厂方法透明的创建对象。作为一个结果，客户端不需要知道任何ConcreteProduct和他们是如何创建的。这种方法提倡面向对象的编程原理“程序接口，而不是实现”，这引出了多态性，面向对象编程的一个重要特征。此外，对象创建集中在ConcreteCreator类，Product和ConcreteProduct的任何改变不影响客户端。

## Applying the Factory Method Pattern

To apply the factory method pattern in the pizza store application, let us first create the abstract Pizzaclass and its subclasses.



# 抽象工厂模式（Abstract Factory Design Pattern）

The abstract factory pattern is one of the classic [Gang of Four](http://springframework.guru/gang-of-four-design-patterns/) creational design patterns used to create families of objects, where the objects of a family are designed to work together. In the abstract factory pattern you provide an interface to create families of related or dependent objects, but you do not specify the concrete classes of the objects to create. From the client point of view, it means that a client can create a family of related objects without knowing about the object definitions and their concrete class names.

It is easy to confuse the abstract factory pattern with the factory method pattern because both design patterns deal with object creation. Both the patterns advocates the [Object Oriented Programming (OOP)](https://en.wikipedia.org/wiki/Object-oriented_programming) principle “Program to an interface, not an implementation” to abstract how the objects are created. Both design patterns help in creating client code that is loosely-coupled with object creation code, but despite the similarities, and the fact that both the patterns are often used together, they do have distinct differences.

抽象工厂模式是经典的GOF创建型设计模式，来创建对象族。家族中的对象被设计用来一起工作。在抽象工厂模式中提供一个创建相关或者依赖对象家族的接口，但是不需要指定需要创建对象的具体类。从客户端的角度来看，这意味着客户端可以创建一个相关的对象族，而不需要知道对象的定义和具体的类名。

抽象工厂模式和工厂方法模式很容易混淆，因为这两种设计模式都处理对象创建。这两种模式都提倡面向对象编程（OOP）原理"Program to an interface, not an implementation"来抽象对象是如何创建的。这两种设计模式都有助于创建与对象创建代码松散耦合的客户机代码，但尽管有相似之处，而且这两种模式经常使用在一起，它们也有明显的区别。

## Abstract Factory Pattern vs Factory Method Pattern

Abstract factory adds another level of abstraction to factory method. While factory method abstracts the way objects are created, abstract factory abstracts how the factories are created. The factories in turn abstracts the way objects are created. You will often hear the abstract factory design pattern referred to as a “factory of factories“.

From implementation point of view, the key difference between the factory method and abstract factory patterns is that factory method is just a **method** to create objects of a single type, while abstract factory is an**object** to create families of objects.

抽象工厂向工厂方法增加了另一层的抽象。而工厂方法抽象了对象创建的方式，抽象工厂抽象了工厂是如何被创建的。工厂依次抽象了对象创建的方式。你会经常听到抽象工厂设计模式被称为“工厂的工厂”。

Another difference is that the factory method pattern uses [inheritance](https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html) while the abstract factory pattern uses[composition](https://en.wikipedia.org/wiki/Object_composition). We say that that factory method uses inheritance because this pattern relies on a subclass for the required object instantiation. Recall in the [Factory Method Design Pattern](http://springframework.guru/factory-method-design-pattern/) post where we created a createPizza() factory method in an abstract base class and implemented the factory method in a PizzaFactory subclass for the required Pizza object instantiation. On the other hand, the abstract factory pattern delegates responsibility to a separate object (abstract factory) dedicated to create a family of related objects. Then, through composition, the abstract factory object can be passed to the client who will use it (instead of factory method) to get the family of related objects.

另外一个不同是工厂方法模式使用继承，而抽象工厂模式使用组合。我们说工厂方法使用继承，因为该模式依赖于所需对象实例的子类。回想在工厂方法模式中，我们在抽象基类中创建了一个createPizza()工厂方法，并且为所需要的Pizza对象实例在PizzaFactory子类中实现了它。另一方面，抽象工厂模式将责任委托给一个单独的对象（抽象工厂），专门创建一个相关对象的家族。然后通过组合，抽象工厂对象被传递到客户端，客户端将使用它（而不是工厂方法）获取相关对象的家族。

## Participants in the Abstract Factory Pattern

To understand how the abstract factory pattern works, let us revisit the pizza store that we developed in the[Factory Method Design Pattern](http://springframework.guru/factory-method-design-pattern/) post. The store has seen a rapid increase in its customer base and now wants to serve their existing types of pizzas: cheese, pepperoni, and veggie in two different topping styles: Sicilian topping and Gourmet topping. Each topping style will require a different combination of products. Sicilian topping will have Goat Cheese with Tomato Sauce while Gourmet topping will have Mozzarella Cheese with California Oil Sauce. To model the new requirements of the application, we can create the concrete Cheeseproducts: GoatCheese and MozzarellaCheese and the concrete Sauce products: TomatoSauce and CaliforniaOilSauce. Now, as we do not want any client to directly instantiate the products, we will abstract the way the products are created by introducing an abstract factory. We will create a BaseToppingFactoryabstract factory class and let its two concrete subclasses: SicillianToppingFactory and GourmetToppingFactory create our products. Here, it is important to note that an abstract factory should be designed to create families of products. So we can model SicillianToppingFactory to create the family of MozzarellaCheese and TomatoSauce products and GourmetToppingFactory to create the family of GoatCheese and CaliforniaOilSauce products.

Now, let us summarize the components of the abstract factory pattern in the context of the enhanced pizza store:

* AbstractProduct (Cheese and Sauce): Is an interface or an abstract class whose subclasses are instantiated by the abstract factory objects.
* ConcreteProduct (GoatCheese, MozzarellaCheese, TomatoSauce, and CaliforniaOilSauce): Are the concrete subclasses that implement/extend AbstractProduct. The abstract factory objects instantiate these subclasses.
* AbstractFactory (BaseToppingFactory): Is an interface or an abstract class whose subclasses instantiate a family of AbstractProduct objects.
* ConcreteFactory (SicillianToppingFactory and GourmetToppingFactory): Are the concrete subclasses that implement/extend AbstractFactory. An object of this subclass instantiates a family of AbstractProduct objects.
* Client: Uses AbstractFactory to get AbstractProduct objects.

如何理解抽象工厂模式是如何工作的，让我们重温工厂方法模式中披萨店的例子。商店已经看到在其客户群的迅速增加，现在要为现有类型的比萨：奶酪披萨，香肠披萨，和蔬菜披萨增加两不同调料：Sicilian和Gourmet。每一个调料样式需要不同的产品组合。Sicilian配料需要Goat Cheese与 Tomato Sauce，而Gourmet配料需要Mozzarella Cheese和California Oil Sauce。为模拟应用程序的新需求，我们可以创建具体的Cheese产品：GoatCheese和MozzarellaCheese，还有具体的Sauce产品：TomatoSauce和CaliforniaOilSauce。现在，我们不希望任何客户直接实例化产品，我们通过引入一个抽象的工厂来抽象产品创建的过程。我们将创建一个BaseToppingFactoryabstract工厂类，让它的两个具体的子类：SicillianToppingFactory和GourmetToppingFactory创造我们的产品。在这里，重要的是要注意，一个抽象的工厂应该被设计来创建产品族。所以我们可以使用SicillianToppingFactory来创建MozzarellaCheese和TomatoSauce产品族，而使用GourmetToppingFactory来创建GoatCheese和CaliforniaOilSauce产品族。

现在，让我们在增强的比萨店实例中，总结抽象工厂模式的组成部分：

* AbstractProduct (Cheese and Sauce): 为一种产品声明接口
* ConcreteProduct (GoatCheese, MozzarellaCheese, TomatoSauce, and CaliforniaOilSauce): 定义具体工厂生成的具体产品对象，实现产品接口
* AbstractFactory (BaseToppingFactory): 声明生产抽象产品的方法
* ConcreteFactory (SicillianToppingFactory and GourmetToppingFactory): 执行生产抽象产品的方法，生成一个具体的产品。
* Client: 使用抽象产品和抽象工厂生成对象。

## Applying the Abstract Factory Pattern

To apply the abstract factory pattern in the pizza store application, let us first create the products that the factories will produce.

为了在Pizza店应用中应用抽象工厂模式，让我们首先创建工厂生产的产品。

**public** **interface** Cheese {

**void** prepareCheese();

}

**public** **class** GoatCheese **implements** Cheese {

**public** GoatCheese(){

prepareCheese();

}

@Override

**public** **void** prepareCheese() {

System.***out***.println("Prepare goat cheese ......");

}

}

**public** **class** MozzarellaCheese **implements** Cheese {

**public** MozzarellaCheese(){

prepareCheese();

}

@Override

**public** **void** prepareCheese() {

System.***out***.println("Prepare Mozzarella cheese ......");

}

}

**public** **interface** Sauce {

**void** prepareSause();

}

**public** **class** TomatoSauce **implements** Sauce {

**public** TomatoSauce(){

prepareSause();

}

@Override

**public** **void** prepareSause() {

System.***out***.println("Preparing tomato sauce..");

}

}

**public** **class** CaliforniaOilSauce **implements** Sauce {

**public** CaliforniaOilSauce(){

prepareSause();

}

@Override

**public** **void** prepareSause() {

System.***out***.println("Preparing tomato sauce..");

}

}

在上面的例子中，编写了Cheese接口，它是一个抽象的产品。接着我们写GoatCheese类和MozzarellaCheese类，这两个类是实现了Cheese接口的具体工厂。同样的，对于披萨调味汁，我们写了Sauce接口，TomatoSauce类和CaliforniaOilSauce类实现了这个接口。

接下来，我们编写创建产品的工厂。我将要开始抽象工厂的编写。

**public** **abstract** **class** BaseToppingFactory {

**public** **abstract** Cheese createCheese();

**public** **abstract** Sauce createSauce();

}

In the example above, we wrote the BaseToppingFactory abstract class, the abstract factory of our application. in the abstract factory, we declare the createCheese() and createSauce() abstract methods that return Cheese and Product objects respectively. As stated in the definition of abstract factory earlier, our abstract factory (BaseToppingFactory) is providing an “interface to create families of related or dependent objects“. The related objects here are Cheese and Sauce, both of which are together used to create toppings. The definition also states “…but you do not specify the concrete classes of the objects to create“. As you can notice in the BaseToppingFactory code, our abstract factory is not concerned with any of the concrete products: GoatCheese, MozzarellaCheese, TomatoSauce, or CaliforniaOilSauce. Let us now write the ConcreteFactory implementations.

**public** **class** SicillianToppingFactory **extends** BaseToppingFactory {

@Override

**public** Cheese createCheese() {

**return** **new** MozzarellaCheese();

}

@Override

**public** Sauce createSauce() {

**return** **new** TomatoSauce();

}

}

**public** **class** GourmetToppingFactory **extends** BaseToppingFactory {

@Override

**public** Cheese createCheese() {

**return** **new** GoatCheese();

}

@Override

**public** Sauce createSauce() {

**return** **new** CaliforniaOilSauce();

}

}

**public** **abstract** **class** Pizza {

**public** **abstract** **void** addIngredients();

**public** **void** bakePizza() {

System.***out***.println("Pizza baked at 400 for 20 minutes.");

}

}

**public** **class** CheesePizza **extends** Pizza {

BaseToppingFactory toppingFactory;

**public** CheesePizza(BaseToppingFactory toppingFactory){

**this**.toppingFactory = toppingFactory;

}

@Override

**public** **void** addIngredients() {

System.***out***.println("Preparing ingredients for cheese pizza.");

toppingFactory.createCheese();

toppingFactory.createSauce();

}

}

**public** **class** PepperoniPizza **extends** Pizza {

BaseToppingFactory toppingFactory;

**public** PepperoniPizza(BaseToppingFactory toppingFactory)

{

**this**.toppingFactory=toppingFactory;

}

@Override

**public** **void** addIngredients() {

System.***out***.println("Preparing ingredients for pepperoni pizza.");

toppingFactory.createCheese();

toppingFactory.createSauce();

}

}

**public** **class** VeggiePizza **extends** Pizza {

BaseToppingFactory toppingFactory;

**public** VeggiePizza(BaseToppingFactory toppingFactory){

**this**.toppingFactory = toppingFactory;

}

@Override

**public** **void** addIngredients() {

System.***out***.println("Preparing ingredients for veggie pizza.");

toppingFactory.createCheese();

toppingFactory.createSauce();

}

}

**public** **abstract** **class** BasePizzaFactory {

**public** **abstract** Pizza createPizza(String type);

}

**public** **class** GourmetPizzaFactory **extends** BasePizzaFactory {

@Override

**public** Pizza createPizza(String type){

Pizza pizza;

BaseToppingFactory toppingFactory= **new** GourmetToppingFactory();

**switch** (type.toLowerCase())

{

**case** "cheese":

pizza = **new** CheesePizza(toppingFactory);

**break**;

**case** "pepperoni":

pizza = **new** PepperoniPizza(toppingFactory);

**break**;

**case** "veggie":

pizza = **new** VeggiePizza(toppingFactory);

**break**;

**default**: **throw** **new** IllegalArgumentException("No such pizza.");

}

pizza.addIngredients();

pizza.bakePizza();

**return** pizza;

}

}

**public** **class** SicilianPizzaFactory **extends** BasePizzaFactory{

@Override

**public** Pizza createPizza(String type) {

Pizza pizza;

BaseToppingFactory toppingFactory = **new** SicillianToppingFactory();

**switch**(type){

**case** "cheese":

pizza = **new** CheesePizza(toppingFactory);

**break**;

**case** "pepperoni":

pizza = **new** PepperoniPizza(toppingFactory);

**break**;

**case** "veggie":

pizza = **new** VeggiePizza(toppingFactory);

**break**;

**default**: **throw** **new** IllegalArgumentException("No such pizza.");

}

pizza.addIngredients();

pizza.bakePizza();

**return** pizza;

}

}

**public** **class** GourmetPizzaFactoryTest {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

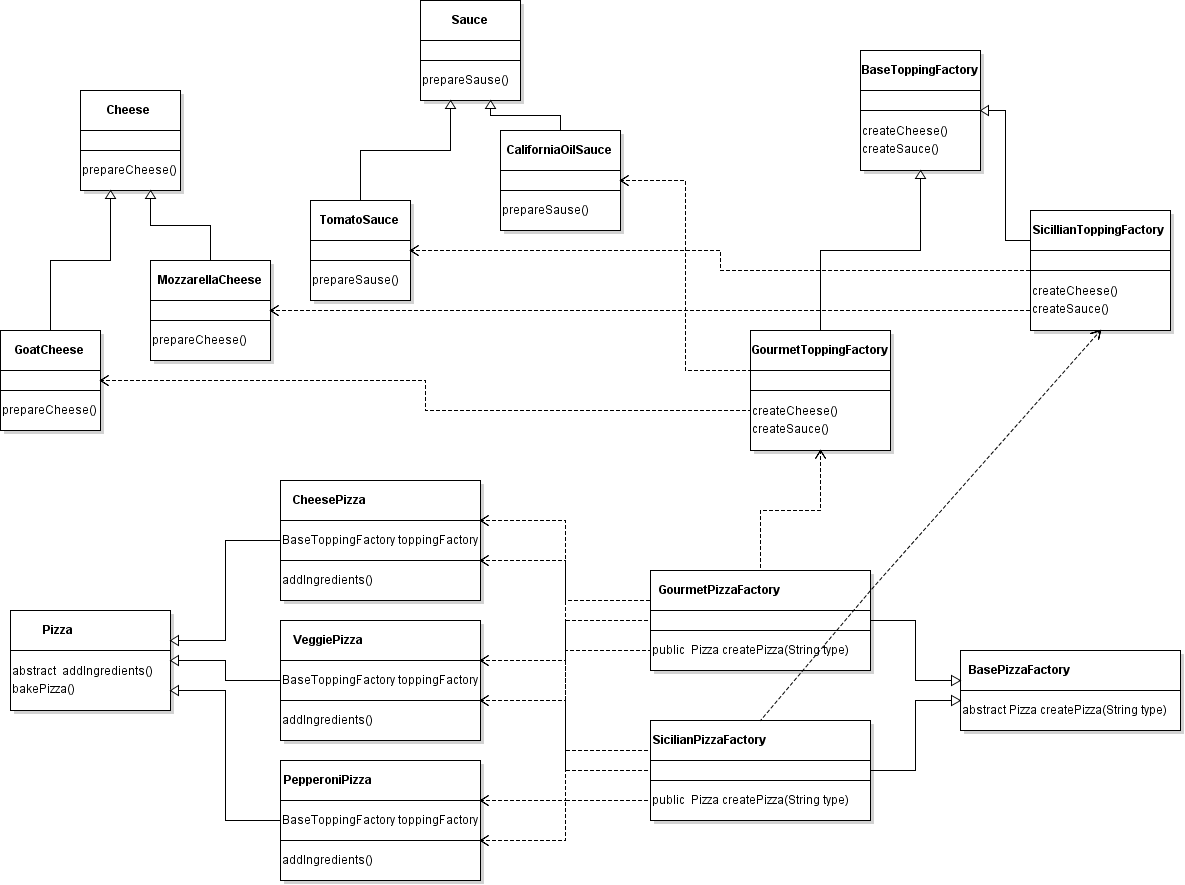
BasePizzaFactory pizzaFactory = **new** GourmetPizzaFactory();

pizzaFactory.createPizza("cheese");

pizzaFactory.createPizza("veggie");

}

}



**建造者模式（Builder Pattern）**

“Separate the construction of a complex object from its representation so that the same construction process can create different representations.”

将一个复杂对象的构建与他的表示分离，似的同样的构建过程能够创建不同的表示。

The Builder pattern is a classic [Gang of Four](http://springframework.guru/gang-of-four-design-patterns/) creational design pattern. This pattern, similar to the other creational patterns, such as [factory method](http://springframework.guru/factory-method-design-pattern/) and [abstract factory](http://springframework.guru/abstract-factory-design-pattern/), is concerned with the creation of objects. But why do we need another pattern to create objects? To answer this, let us first look at a problem scenario.

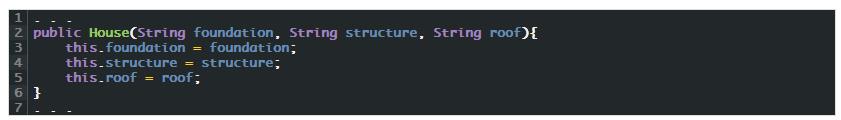
建造者模式是一个经典的GOF创建型设计模式。这个模式与其他的创建型模式（如：工厂模式、抽象工厂模式）相似，都是和对象的创建相关。为什么我们还需要另外一种模式去创建对象呢？为了回答这个问题，让我们首先看下一个问题场景。

**问题（The Problem）**

Consider that you need to build a house for a customer. Now, building a house will consist a series of steps. You will start with the foundation, then the structure, and finally the roof. But, how will you do this in the programming world? This should be simple in Java- Right? You create a House class with the required fields and initialize them through a constructor, like this.

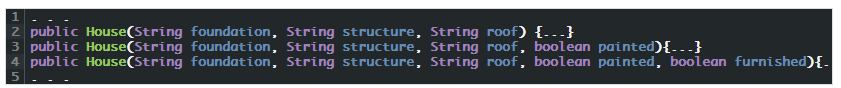
假设你需要为一个客户建造一个房子。现在，建造一个房子是由一系列步骤组成。

你由建地基开始，然后创建结构，最后封顶。但是，在编程的世界里你怎么做呢？使用Java这应该是非常简单的-对吗？你创建一个房屋类，使用必须的字段并且通过一个构造器初始化他们，像这样：



This builds the house and your customer is happy. But the next customer wants the house to be painted and furnished while another wants it only painted. You now need to revisit the House class with a set of overloaded constructors, something like this.

这样建成了房子，并且客户也高兴。但是下一个客户想建造一个喷漆和带家具的房子，而另一个客户仅想建造一个喷漆的。你需要重新改写房屋类，增加一组重载的构造器，想下面这样：



Although this will work, we have a flawed design. What we applied here is the telescopic constructor pattern, which is considered an [anti-pattern](http://c2.com/cgi/wiki?AntiPattern). Though this pattern works for simple object creation with a limited number of fields, it becomes unmanageable for complex object creation. Imagine the number of constructors you will need to create a more complex house with options for plumbing, lightning, interiors, and so on. Another major problem in this design is related to constructor calls that clients need to make. It is hard and error prone to write client code when there are multiple constructors, each with a large set of parameters. In addition, readability is a major issue with such client code.

尽管这样也能工作，但是这是一个有缺陷的设计。这里我们应用的是伸缩式的构造器模式，它被认为是一个反面模式。

尽管这种模式适用于这种有限数量字段的简单对象的创建，但是对于复杂对象的创建变得很难管理。想象下构造函数的数量如果你需要创建一个更复杂的房子包括管道的选择,灯饰,内饰,等等。这种设计的另一个主要的问题是客户端需要构造函数的调用。*而且容易出错很难编写客户机代码,当有多个构造函数,每一个都有大量的参数。此外, 这样的客户端代码可读性是一个主要问题。*

While writing such client code, you will often end up with questions, such as:

* Which constructor should I invoke?
* What will be the default values of the parameters if I don’t provide?
* Does the first boolean in the constructor represents painting or furnishing?

One solution to the telescopic constructor pattern is to follow [JavaBeans](https://en.wikipedia.org/wiki/JavaBeans) conventions by writing setter methods instead of a set of constructors to initialize the fields.

当写完客户端代码是，你会提出以下几个问题：

我应该调用那一个构造器？

如果我什么也不提供，参数的默认值是什么？

构造器中的布尔型参数代表喷漆还是装饰？

对这种可伸缩的构造模式的解决方法是按照JavaBean的定义通过写get、set方法来替换这些构造方法初始化字段。



Clients can now call the setter methods with appropriate parameters to create House objects. Also, client code are now more readable and therefore have lesser chances of errors creeping in.

Your house building business is growing and everything is going fine until a customer calls up and complains that his house collapsed during construction. On examining, you found this particular client code.

客户端可以用合适的参数调用setter方法，来创建House对象。同时客户端代码现在看来可读性有所增强，并且能够减少出错的机会。

随着房屋建筑业务的增长一切都很好，直到一个客户投诉房子在建造的时候倒塌了。通过检查，你发现了特殊的客户端代码：



As you can see, the client code tried building the roof before the structure was in place, which means that the steps to build a house was not in the correct order. Another problem is the client having an instance of the House class in an inconsistent state. This means, if a client wants to create a House object with values for all its fields then the object will not have a complete state until all the setter methods have been called.

As a result, some part of the client application might see and use a House object assuming that is already constructed while that is actually not the case.

While you might be still pondering over the existing problems on hand, imagine that a customer calls up with a requirement for a prefabricated house, another customer for a tree house, and yet another for an Igloo (a snow house). Now, here is a whole new set of problems to solve.

At this point you should consider yourself lucky because other people have faced similar problems and have come up with proven solutions. It is time to learn the classic [GoF](http://springframework.guru/gang-of-four-design-patterns/" \o "Gang of Four Design Patterns) Builder pattern.

正如你看到的客户端代码试图在结构就位之前就建造屋顶。，这意味着建造房屋的步骤不是正确的顺序。另一个问题是客户端具有不一致状态的House类的实例。也就是说，如果一个客户端想创建所有字段都有值得House对象，那么对象将不会有一个完整的状态，直到所有的setter方法已被调用。因此，一部分客户端应用可能会看到或者使用一个假设已经构建的对象，而实际上不是这样。

你可能还在考虑接下来存在的问题，假设一个客户提出了一个预制房屋的要求，另一个客户提出一个树屋的需求，而其他的可能还要求冰屋。现在，这里有一整套新的问题要解决。

在这一点上，你应该认为自己是幸运的，因为其他人都面临着类似的问题，并提出了行之有效的解决方案。是时候学习经典的GoF的建造者模式。

## 建造者模式（Builder Pattern）

### 简介（Introduction to the Builder Pattern）

The builder pattern allows you to enforce a step-by-step process to construct a complex object as a finished product. In this pattern, the step-by-step construction process remains same but the finished products can have different representations. In the context of the house building example, the step-by-step process includes the steps to create the foundation, structure, and roof followed by the steps to paint and furnish a house and these steps remain the same irrespective of the type of house to build. The finished product, which is a house, can have different representations. That is, it can be a concrete house, a prefabricated house, or a tree house.

建造者模式允许你执行一步步的处理来构建一个复杂的对象作为一个制成品。在这个模式中，一步步的创建过程保持一样但是完成的产品有不同的表示。在上面房屋构建的例子中，处理的步骤包括建造地基、架构、屋顶，然后喷漆、布置房间。这些步骤是固定的不管你是建造什么样的房屋。完成的产品（房屋），可能有不同的表示。也就是，可能是混凝土房屋、活动房屋、树屋。

### 参与者（Participants in the Builder Pattern）

To understand how the builder pattern works, let us solve the problems of our house building example. The main problem was that we expected the clients to perform the steps to construct a house and that too in the correct order. So, how will we address this in real life? We will hire a construction engineer who knows the process to construct houses.

为了理解建造者模式如何工作，让我们解决房屋建造例子中的问题，主要的问题是我们希望客户端执行步骤来构建一个房子而且也是按照正确的顺序。所以在实际的生活中我们怎么处理呢？我们会雇佣一个建筑师，它知道建造房屋的处理流程。

The second problem was that we require different types of houses, such as concrete, prefabricated, tree house, and even Igloos. So next, we will hire builders (contractors) who specializes in building specific types of houses. A builder knows how to put things together with actual building materials, components, and parts to build a particular type of house. For example, a concrete house builder knows how to build the structure of a concrete house using concrete, brick, and stone. Similarly, a prefabricated house builder knows how to build the structure of a prefabricated house using structural steels and wooden wall panels. So from now on, whenever we need a house, the construction engineer will direct a builder to build the house.

第二个问题是我们需要不同类型的房屋，比如混凝土、活动房屋、树屋乃至冰屋。接下来，我们会雇佣施工人员（contractors）它专门建造不同类型的房屋。一个建筑者知道怎样把东西放到一起与实际的建筑材，组件，零件来构建一个特定类型的房屋。例如：混凝土房屋建造者知道如何建造混凝土房屋的结构使用混凝土、砖和石头。同样活动房屋建造者知道如何建造活动房屋的结构使用钢架、木板。到现在为止，无论何时你需要一个房屋，建筑师会指挥一个建筑工人去创建这个房屋。

In our application, we can model the construction engineer by creating a ConstructionEngineer class. Then we can model the builders by first creating a HouseBuilder interface and then builder classes, such as ConcreteHouseBuilder and PrefabricatedHouseBuilder that implement the HouseBuilder interface. Here, notice that we have added a layer of abstraction by providing an interface (HouseBuilder). This is because we do not want our construction engineer to be tied with a particular builder. The construction engineer should be able to direct any builder that implements the HouseBuilder interface to build a house. This will also allow us to later add new builders without making changes to the existing application code.

在我们的应用中，可以模仿建筑师创建一个ConstructionEngineer 类。然后我们可以模仿建筑工人创建一个HouseBuilder 接口和建造者类如：ConcreteHouseBuilder 类和PrefabricatedHouseBuilder 他们继承了HouseBuilder 接口。这里注意下，我们增加了一层的抽象通过提供一个接口HouseBuilder 。这是因为我们不想我们的建筑师像一个特殊的建筑工人一样。建筑师应该能够指挥继承了HouseBuilder 的任何一个建筑工人来建造一个房屋。这样也允许我们以后增加新的建造者，而不会更改存在的应用代码。

We can now summarize the components of the builder pattern in the context of the house building example as:

* Product (House): A class that represents the product to create.
* Builder (HouseBuilder): Is an interface to build the parts of a product.
* ConcreteBuilder(ConcreteHouseBuilder and PrefabricatedHouseBuilder): Are concrete classes that implement Builder to construct and assemble parts of the product and return the finished product.
* Director (ConstructionEngineer): A class that directs a builder to perform the steps in the order that is required to build the product.

我们现在总结下建造者模式的组件：

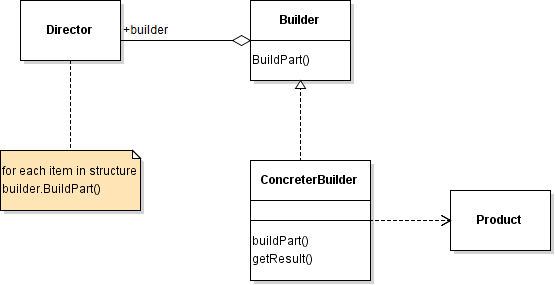
产品角色（House）:创建产品的表示

抽象建造者（HouseBuilder）：为一个产品对象的各个部件指定统一的接口

具体建造者（ConcreteHouseBuilder | PrefabricatedHouseBuilder）

指挥者（ConstructionEngineer）：指挥建造者来按照要求和步骤来创建产品

UML类图

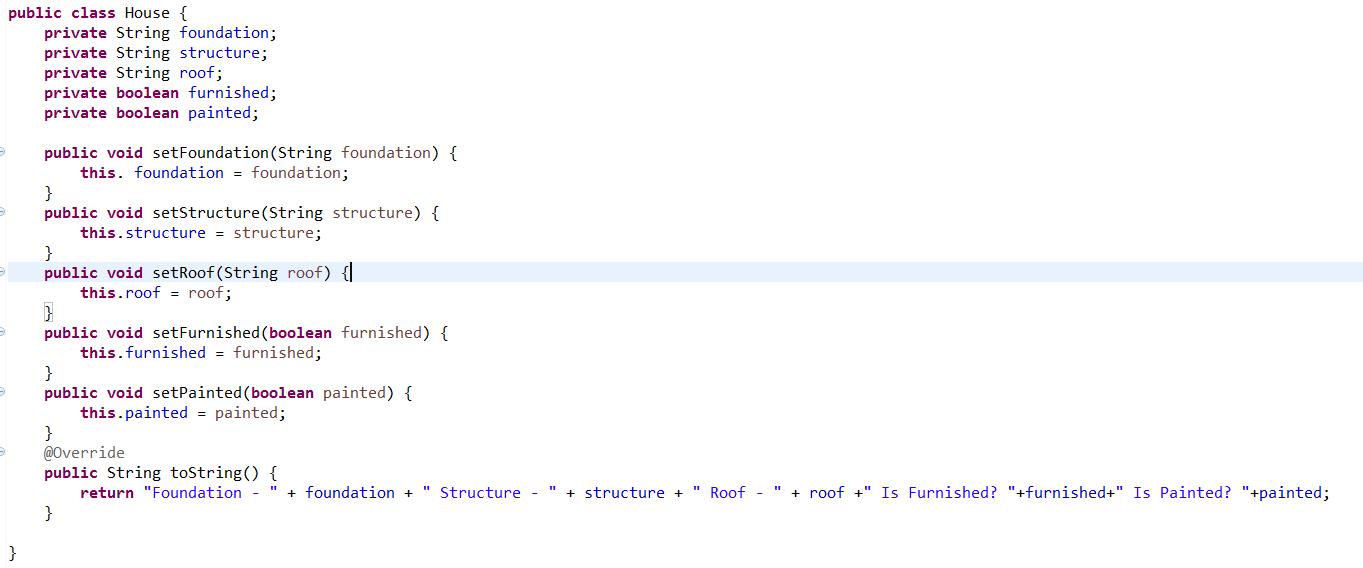


### 应用建造者模式（Applying the Builder Pattern）

To apply the builder pattern to the house building example, let us first create the product that the builders will construct.

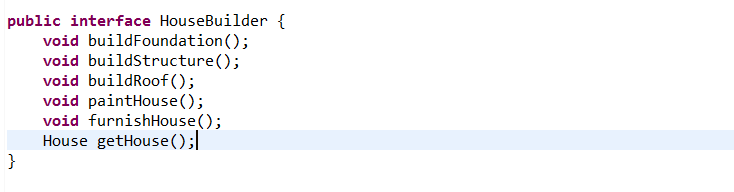
应用建造者模式

为了应用建造者模式到房屋建造的例子中，让我们先创建建造者将要建造的一个产品



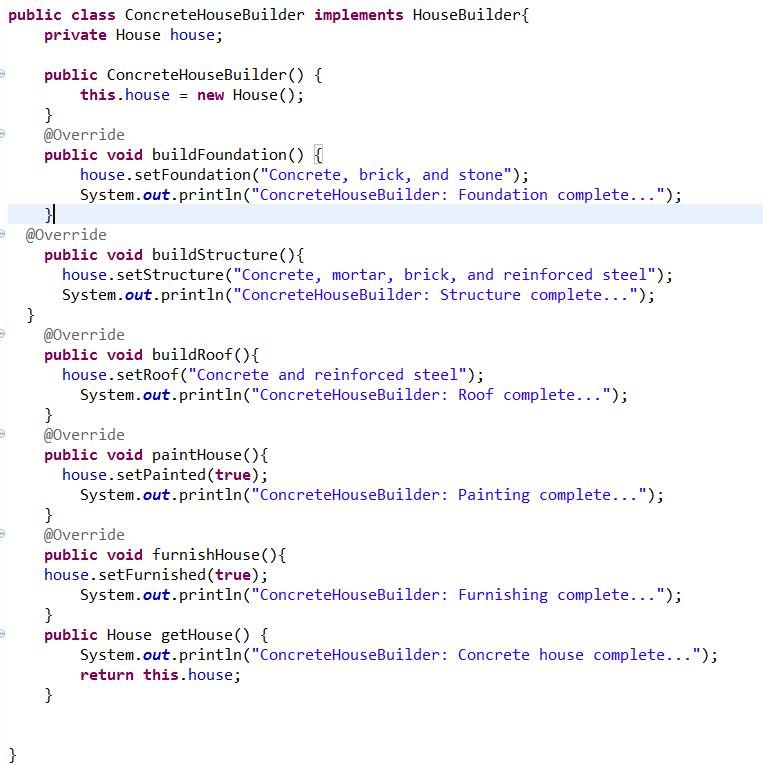
In the example above, we wrote a House class with five fields and their corresponding setter methods. Next, we will create the HouseBuilder interface, which is the Builder in the application.

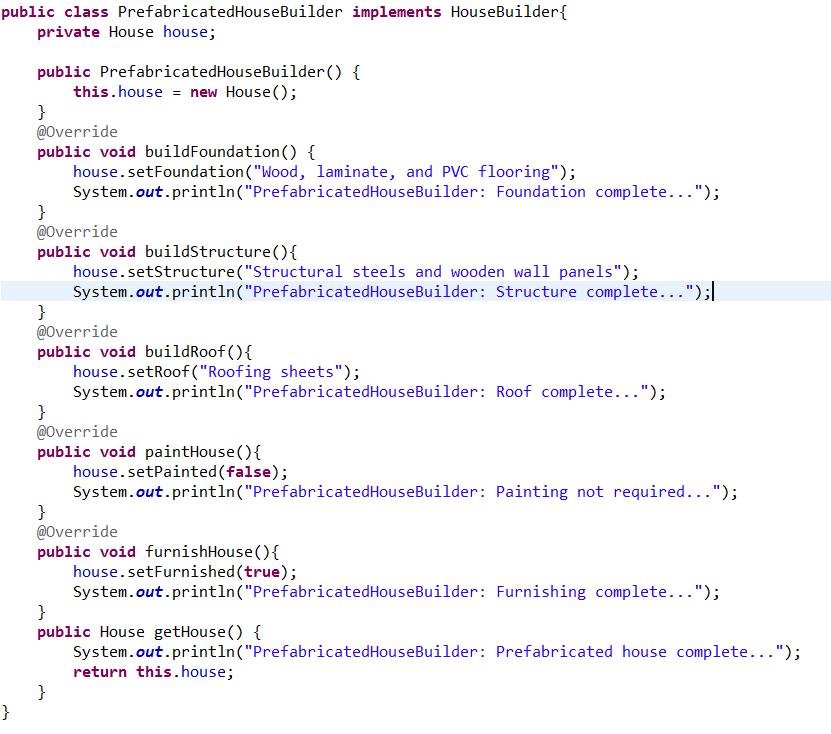
下面我们创建 HouseBuilder 接口，它是应用程序的Builder：



In the example above, we wrote the HouseBuilder interface to declare five methods to create the parts of the product (House). We also declared a getHouse() method that returns the finished product. We will provide the implementation of the methods in the concrete subclasses: ConcreteHouseBuilder and PrefabricatedHouseBuilder, which are the ConcreteBuilder components in the application.

接下来我们提供方法的实现在具体的子类中：ConcreteHouseBuilder and PrefabricatedHouseBuilder，它们是程序的ConcreteBuilder

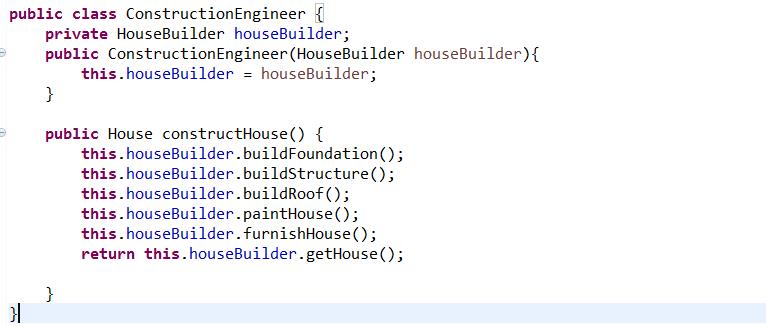




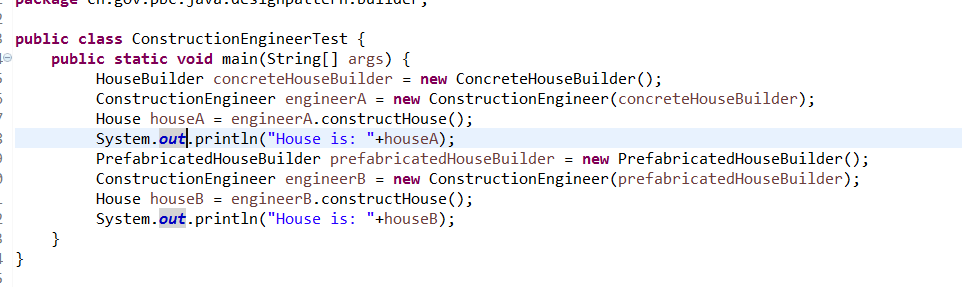
In the above examples, we first wrote the ConcreteHouseBuilder class. In the constructor of this class, we created a House object. We then implemented the methods declared in the HouseBuilder interface to create the parts of a concrete house through calls to the setter methods of the House object. Finally, we implemented the getHouse() method to return the final House object that represents a concrete house. Similarly, we wrote the PrefabricatedHouseBuilder class to create the parts of a prefabricated house and return the final House object that represents a prefabricated house.

With these two classes in place, we are almost ready to “create different representations” of a house: concrete and prefabricated. But, we are yet to define the “same construction process“. We will do it next in the ConstructionEngineer class, which is the Director in the application.

下面是ConstructionEngineer ，它是应用程序的Director



In the above example, we wrote the ConstructionEngineer class with a constructor that accepts a HouseBuilder object. In the constructHouse() method, we made a series of calls on the HouseBuilderobject in a certain order and returned the final House object to the caller. Notice that the ConstructionEngineer class is not tied to any concrete builder. Also, this class uses the same construction process in the constructHouse() method irrespective of the type of concrete builder provided to it at run time. This allows us to add new concrete builder classes without making any changes to the construction process. Now that our house building example is ready, let us write a unit test to observe the builder pattern at work.



As you can see in the example above, a client is now insulated from the object creation process. A client only needs to provide the Director a ConcreteBuilder to use. It is the responsibility of the Director to instruct the ConcreteBuilder on the construction process and the ConcreteBuilder in turn will create the finished product. Finally, the client receives the finished product from the Director.  
When you run the code above, you will see this output:

**Conclusion**

If you are familiar with the [abstract factory](http://springframework.guru/abstract-factory-design-pattern/) pattern, you might have observed that both the abstract factory and builder patterns are similar, as both can be used to abstract object creation. But there are distinct differences between the two. While abstract factory emphasizes on creating a family of related objects in one go, builder is about creating an object through a step-by-step construction process and returning the object as the final step. In short abstract factory is concerned with **what**is made, while the builder with **how**it is made. So as you go further into enterprise application development, whenever you need to create complex objects independently of the construction algorithm turn to the classic GoF Builder Pattern!

**原型模式（Prototype Pattern）**

“Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.”

使用一个原型实例创建指定类型的对象，并且通过复制这个原型创建新的对象。

**简介（Prototype Pattern: Introduction）**

The prototype pattern is a classic [Gang of Four](http://springframework.guru/gang-of-four-design-patterns/) creational pattern, and similar to the other members of the creational pattern family: [singleton](http://springframework.guru/singleton-design-pattern/) , [factory method](http://springframework.guru/factory-method-design-pattern/), [abstract factory](http://springframework.guru/abstract-factory-design-pattern/), and [builder](http://springframework.guru/builder-pattern/), prototype is also concerned with object creation, but with a difference. Using the prototype pattern, you do not create a new object for each client requesting the object. Instead, you start by creating a single object, called a prototype and make copies of it for each client requesting the object. In Java, this is achieved through object cloning, a way to make a copy of an object with the same state as the original object.

But why create copies if we can create new objects through constructor calls, which is much simpler? Most of the time you will not need to create copies of objects. But, as you move into enterprise application development where application performance is critical, you will encounter situations where construction of an object involves time consuming operations, such as network communication, database reads, and disk I/O. If a large number of such objects needs to be created, you can avoid repeating those steps for each object by initially creating a prototype and then making copies of it.

While the prototype pattern is itself simple, it is important to learn the intricacies involved in Java object cloning before you start applying the prototype pattern to applications.

原型模式是一个经典的GOF创建型模式，和其他的创建型模式（单例模式、工厂方法、抽象工厂和创建者）相似，原型模式也与对象创建有关，但是也有一个不同。使用原型模式，你不用为每一个客户端请求对象创建一个新的对象。相反，你首先创建一个对象叫做原型，并且为每一个客户端请求的对象产生一个副本。在java中，这是通过对象克隆,一种通过产生一个具有相同状态对象的副本作为原始对象的方式。

但是我们可以通过构造方法调用创建新的对象，为什么却要创建副本？那种方式更简单？大部分时间你不需要创建对象的副本，但是当你进入企业级应用程序开发，这里应用性能非常吃紧，你将会遭遇这种状况，创建一个对象涉及非常耗时的操作，比如：网络通讯，数据库读取、磁盘IO等。如果大批量的这样的对象需要被创建，你可以避免为每一个对象重复这些步骤，通过在最初创建一个原型，然后生成它的副本。

虽然原型模式本身是简单的，但是在你开始应用原型模式到你的应用之前，学习所涉及的错综复杂的Java对象克隆是非常重要的。

**对象克隆（Object Cloning: Shallow Copy Vs Deep Copy）**

Java provides the [Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html" \t "_blank" \o "Cloneable Interface) interface to mark objects that permit cloning. This interface is a marker interface and therefore does not contain any method declaration. When implemented in a class, Cloneable marks that objects of the class can be cloned. To perform cloning, you need to call the protected [clone()](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#clone--) method of the Object class through a call to super.clone().

**Note**: If an object calls super.clone() but does not implements Cloneable, the method throws an exception of type, CloneNotSupportedException.

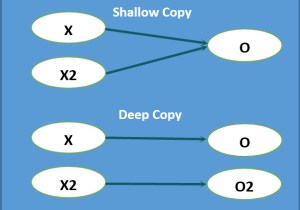
Java提供了Cloneable接口来标识运行克隆的对象。这个接口是一个标记接口，因此不包含任何的方法声明。当在类中实现时，Cloneable标识类的对象能够被克隆。为了执行克隆，你需要调用Object类的这个protected clone()方法，通过一个调用super.clone()。

注释：如果一个对象调用了super.clone()但是没有继承Cloneable，这个方法会抛出一个异常CloneNotSupportedException。

A call to super.clone() performs a shallow copy where all the fields values of the original object are copied to the new object. If a field value is a primitive type, a shallow copy copies the value of the primitive type. But, if a field value is a reference type, then only the reference is copied, and not the referred object itself. Therefore, both the original and its clone refer to the same object and if either one modifies the referred object, the modification will be visible to the other. This might result in unexpected behavior in an application. In such situation, you should perform a deep copy that makes copies of dynamically allocated memory pointed to by the reference type fields. In a deep copy, the original and the copied objects are independent of each other and therefore the objects can update their own fields without worrying about any referencing problems.

The difference between shallow and deep copy is illustrated in this figure.

对super.clone()的调用执行了一个浅拷贝，原对象的所有的字段值被复制到新的对象。如果一个字段值是基本类型，浅拷贝复制基本类型的值。但是，如果一个字段值是引用类型，只用引用被复制，而不是被引用的对象本身。因此原始的和它的克隆引用的同一对象，如果任何一个修改这个引用对象，修改对另一个都是可见的。这在应用中可能会导致产生不想看到的结果。这种情形，你应该执行一个深拷贝，复制动态分配的内存指向引用类型字段。在深拷贝中，原对象和被拷贝的对象是互相独立的，因此对象可以更新他们自己的字段，而不用担心任何引用问题。浅拷贝和深拷贝的区别见下图：

[](https://i1.wp.com/springframework.guru/wp-content/uploads/2015/04/4-23-2015-1-55-29-AM.jpg)

In the figure above, an object X references an object O. A shallow copy of X creates a new object X2 that also references object O. In contrast, a deep copy of X creates a new object X2 that references the new object O2 (copy of O).

This is a very important distinction to remember.

在上图中，对象X引用了对象O。针对X的浅拷贝创建了X2，X2同样引用了对象O。相反，针对X的深拷贝创建了X2，X2引用了一个新对象O2。

When using the prototype pattern, should we go for shallow copy or for deep copy? There is no hard and fast rule, it all depends on the requirement. If an object has only primitive fields or [immutable objects](https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html) (whose state cannot change, once created), use a shallow copy. When the object has references to other mutable objects, then either choose shallow copy or deep copy. For example, if the references are not modified anytime, avoid deep copy and go for shallow copy. But, if you know that the references will be modified and the modification might affect the intended behavior of the application, then you need to go for deep copy.

When you go for a deep copy, you will need to override theObject.clone() method in all the member classes and then recursively clone their objects. Alternatively, you can get a deep copy by serializing an object and then restoring it back through Java [object serialization](https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html). Using serialization is a simple approach, keep in mind serialization is resource intensive and can produce unexpected results if not done properly.

当使用原型模式时，我们主张使用浅拷贝还是深拷贝呢？这里没有严格的规则，主要还是依据需求。如果一个对象只有简单基础数据类型字段或者不可变对象，使用浅拷贝。当对象引用了其他可变的对象，可使用浅拷贝也可以使用深拷贝。例如，如果引用不会被修改就避免使用深拷贝而使用浅拷贝。但是如果你知道这个引用将会被更改，并且这个更改可能会影响应用程序预期的行为，那么就应该使用深拷贝。

当你使用深拷贝时，你需要在所有的成员类里重写Object.clone()方法，然后递归的去克隆他们的对象。或者你可以通过序列化一个对象然后通过Java对象序列化恢复它来获得深拷贝。使用序列化是一个简单的方式，记住序列化是资源密集型的,如果处理不当会产生意想不到的结果。

**原型模式示例（Example of the Prototype Pattern）**

### 原型模式参与者（Participants of the Prototype Pattern）

To understand how the prototype pattern works, consider a content production house that employs vendors to write content for the organization. For each project assigned to a vendor, the HR department provides the vendor a terms and conditions and a non-disclosure agreement that the vendor must accept before commencing work. The content of the agreements remains same for all vendors and an HR employee only needs to fill in the vendor name before sending an agreement to the vendor. Assuming that the agreement content are stored in a remote database, you can apply the prototype pattern to avoid making a network trip and a database read each time the HR employee needs to create an agreement.

为了了解原型模式是如何工作的，请考虑一个供应商为组织编写内容的内容制作库。对于分配给供应商的每一个项目，人力资源部门为供应商提供一个条款和条件，以及在开始工作前卖方必须接受的保密协议。协议的内容对于所有的供应商来说都是一样的，一个人力资源部的员工在向供应商发出协议之前只需要填写供应商的名字。假设协议内容存储在远程数据库中，您可以应用原型模式，避免每次人力资源人员需要创建协议时进行网络访问和数据库读取。

To implement the requirements of the agreement creation example, you can create an abstract class, named PrototypeCapableDocument that implements Cloneable. This class will act as the base class for all classes that represent agreements and that allows its objects to be cloned. Next, you can model the terms and conditions and non-disclosure agreements with the TAndC and NDAgreement classes respectively. As a non-disclosure agreement is signed by both parties, you can create an AuthorizedSignatory class to represent the signing authority of the content production house and make it a member of the NDAgreement class. Finally, you can create a DocumentPrototypeManager class that will create and store the prototype objects and return a copy whenever a client asks for an agreement. The client will receive a copy of the prototype object and the HR manager will be able to update the copy with the vendor name to create an agreement.

为了实现协议创建例子的需求，你可以创建一个抽象类叫做PrototypeCapableDocument ，这个类继承了Cloneable。这个类充当所有类的基类，这些类代表协议并且他们的对象可以被克隆。下面你可以模仿这些条款和条件和保密协议分别使用TAndC 和NDAgreement 类。随着保密协议被双方当事人签署，你可以创建一个AuthorizedSignatory 类来代表内容制作公司的签字权并且使它成为NDAgreement 类的一个成员。最后你可以创建一个DocumentPrototypeManager 类，这个类会创建存储这个原型对象并且返回一个副本不论何时客户端访问这个协议时。这个客户端将要接收一个原型对象的副本，然后人事经理将会更新这个供应商名称来创建一个协议。

We can now summarize the participants of the prototype pattern in the context of the agreement creation example as:

* Prototype (PrototypeCapableDocument): Is a Java interface or abstract class that defines the contract for classes that permits cloning of its objects.
* ConcretePrototype (TAndC andNDAgreement): Are classes that provide operations to clone its objects.
* PrototypeManager (DocumentPrototypeManager): A class that implements a registry to manage available prototypes for clients. On a client request, this class creates a copy of a prototype.
* Client: Asks the PrototypeManager for copies of prototypes.

现在我们总结下原型模式的参与者：

原型（PrototypeCapableDocument）：一个Java接口或者抽象类，它定义了类的合同，它的对象的许可克隆。

具体原型类（TAndC andNDAgreement）：提供克隆对象的操作。

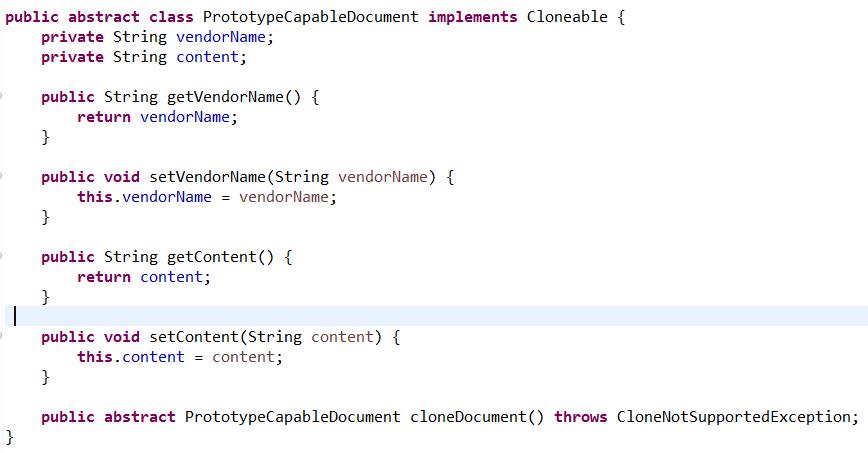
原型管理者（DocumentPrototypeManager）：一个类实现注册管理，为客户提供原型。在客户端请求的时候，这个类创建了一个原型的副本。

客户：请求这个原型管理者来进行原型的复制。

### 应用原型模式（Applying the Prototype Pattern）

To apply the prototype pattern to the agreement creation example, let us first create the prototype as an abstract class.

为了在原型创建的例子中应用原型模式，让我们首先创建这个原型作为一个抽象类。



In the example above, we wrote a PrototypeCapableDocument class that implements the Cloneableinterface. In the class, we defined the vendorName and content fields with their corresponding getter and setter methods. We also defined an abstract cloneDocument() method that subclasses will override to return clones of its objects. Next, we will create the subclass, TAndC, which is one of the ConcretePrototypecomponent in the example. **TAndC.java**

**在上面的例子中，我们写了**PrototypeCapableDocument 类实现了Cloneable接口。在这个类中定义了vendorName和content 字段，还有他们对应的getter和setter方法。我们也定义了一个抽象的方法cloneDocument，子类重写这个方法，并且返回对象的克隆。下面我们将要创建子类TAndC，它是具体原型组件其中之一。

****

In the examples above, we wrote the TAndC class that extends PrototypeCapableDocument and overrides the cloneDocument() method. In this method, we made a call to super.clone(), which will result in a shallow copy of a TAndC object at run time. We do not require a deep copy of a TAndC object because its fields, vendorName and content (inherited from PrototypeCapableDocument) are of type String, which is immutable.

Next, let’s create the NDAgreement and AuthorizedSignatory classes. At this point we need to decide on the type of cloning that we should use. NDAgreement is a concrete prototype composed of AuthorizedSignatory. If we go for shallow copy, both the prototype and its cloned objects will point to the same AuthorizedSignatory object. In our example, we want to prevent any referencing problems later by performing a deep copy operation. In order to do use a deep copy operation, we will need to ensure that AuthorizedSignatory permits itself to be cloned.

在上面的例子中，TAndC 类实现了PrototypeCapableDocument 并且重写了cloneDocumen方法。在这个方法中，调用了super.clone()，在运行期间它将会返回TAndC 对象的浅拷贝。我们不需要一个TAndC 对象的深拷贝，因为它的字段vendorName 和content （继承自父类）是字符串类型，它是不可变的。

接下来，让我们创建NDAgreement and AuthorizedSignatory类。此刻我们需要决定我们使用的克隆的类型。NDAgreement 是一个具体的原型，它由AuthorizedSignatory组成。如果我们主张浅拷贝，原型和它克隆的对象将要指向同样的AuthorizedSignatory 对象。在我们的例子中，我们想要阻止任何的引用问题稍后通过执行深拷贝。为了使用一个深拷贝操作，我们需要确保AuthorizedSignatory 允许他自己被克隆。

****

In the example above, the AuthorizedSignatory class implements Cloneable. In the class, we defined the name and designation fields of type String with their corresponding getter and setter methods. In the overridden clone() method, we made a call to super.clone() to obtain and return back a shallow copy of AuthorizedSignatory to the caller. In our example, the caller will be NDAgreement, which we will create now.

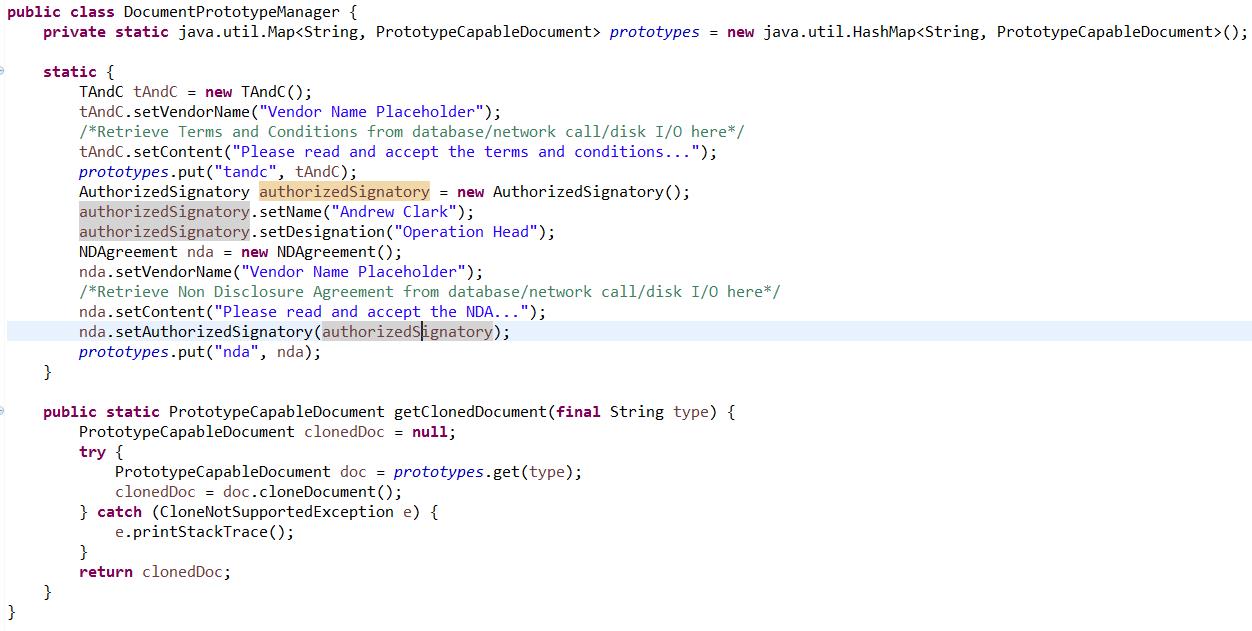
在上面的例子中，AuthorizedSignatory 类继承了Cloneable。在这个类里我们定义了名字和名称两个字符串字段还有他们的setter和getter方法。在重写的clone（）方法中调用了super.clone()，为了获得并返回AuthorizedSignatory 的一个浅拷贝到调用者。在我们的例子中调用者是NDAgreement，它将会现在被创建。



In the example above, we wrote the NDAgreement class that extends PrototypeCapableDocument. In this class, we added a field of type AuthorizedSignatory along with its getter and setter methods. In the overridden cloneDocument() method, we first made a call to super.clone() for a shallow copy of NDAgreement. Then in Line 22-23, we created a copy of AuthorizedSignatory and set it on the shallow copy of NDAgreement. Finally, we returned the NDAgreement copy to the caller.

These are the prototype classes we will use in our implementation of the prototype design pattern. Next let’s create the DocumentPrototypeManager class, which is the PrototypeManager in our example.

在上面的例子中，NDAgreement 继承了PrototypeCapableDocument。在这个类里增加了AuthorizedSignatory 类型的字段和它的getter、setter方法。在重写的cloneDocument方法中调用了super.clone()实现了NDAgreement的浅拷贝，在22-23行创建了AuthorizedSignatory 的副本，然后设置它到NDAgreement的浅拷贝。最后返回了NDAgreement 的副本到调用者。这是在我们实现的原型模式中使用的类。下面我们创建DocumentPrototypeManager 类，在我们的例子中它是原型管理者。

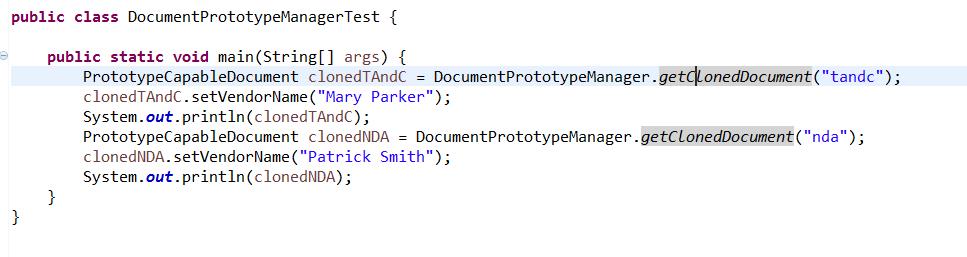


In the example above, we wrote the DocumentPrototypeManager class. In this class, we created an object of type Map that acts as a registry of the prototypes. We then used a static initialization block to create and add the prototypes (TAndC and NDAgreement) to the Map. Finally, we added a getClonedDocument() static method. This method based on the parameter retrieves a prototype from the Map, calls the cloneDocument()  method on the prototype for a copy of it, and returns the copy to the caller.

最后我们增加getClonedDocumen()静态方法。这个方法基于参数从Map中取回一个原型，在原型上调用cloneDocument()方法来拷贝它，然后返回这个副本到调用者。

### 测试原型模式（Testing the Prototype Pattern）

Let’s now write a unit test and observe the prototype pattern at work. **DocumentPrototypeManagerTest.java**

****

## 结论（Conclusion）

The prototype pattern is a simple pattern but a common drawback to using it is with the complexities involved in making object copies. When an object refers other objects, some of which in turn refer other objects. While this is true to some extent, there are circumstances where the benefits that prototype provide far outweigh the inherent complexities of object copying. Especially in enterprise application development, you will encounter situations where copying an object can be significantly efficient than creating a new object. In such situations, use the prototype pattern. It is after all, it is a classic [GoF design pattern](http://springframework.guru/gang-of-four-design-patterns/" \o "Gang of Four Design Patterns).

原型模式是一个简单的模式，但是使用它的一个缺点是涉及对象复制的复杂性。当对象引用其他对象时，其中一些对象反过来引用其他对象。虽然这在某种程度上是真实的，但在某些情况下，原型提供的好处远远超过了对象复制的固有复杂性。特别是在企业应用程序开发中，您会遇到复制对象的情况比创建新对象的效率显著。在这种情况下，使用原型模式。它毕竟是一个经典的GoF设计模式。

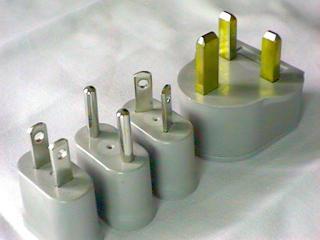
# 适配器模式（Adapter Pattern）

“Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.”

**将一个接口转化成客户想要的另一种接口。适配器使接口不兼容的类可以一起工作**

## 简介（Adapter Pattern: Introduction）

We use adapters in our daily lives. The moment you plugin your mobile handset or your laptop to a socket for charging, an adapter is at work. What the adapter does is makes the socket that produces 120 V (or 220 V for European standard) and the mobile device that requires 4 V work together. Similarly, by using the adapter pattern in the programming world, you can make incompatible interfaces work together.

[](https://i2.wp.com/springframework.guru/wp-content/uploads/2015/06/adapter-plugs-radio-shack.jpg?ssl=1)

The adapter pattern is one of the classic [Gang of Four](http://springframework.guru/gang-of-four-design-patterns/) structural patterns – A set of patterns whose objective is to identify how to realize relationships between classes and objects in a simple way. This pattern is typically used when an incompatible module needs to be integrated with an existing module without making any source code modifications.

## 参与者（Participants of the Adapter Pattern）

To understand how the adapter pattern works, consider an existing text formatting application comprising of a TextFormattable interface and a NewLineFormatter implementation class. A client provides a string to format with a call to the formatText(String text) method declared in the interface. The implementation class returns a new string by replacing the periods with new line characters.

As an enhancement, the application now needs to support CSV formatting so that the text can be read and edited in a spreadsheet software. The new feature has been implemented and delivered by an external vendor as a CsvFormattable interface along with a CsvFormatter implementation class.

To integrate the new feature in the existing application, several issues require addressing. Primarily, the interface against which the new requirements are implemented has a different structure with different methods, and this is not what the clients of the application expect.

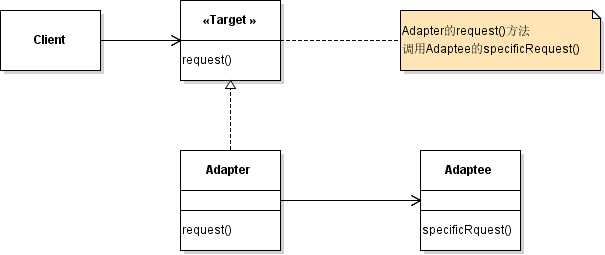
One approach to address such incompatibilities is to ask the vendor to supply a new version with a compatible interface. Another is to update the structure of the existing interface to accommodate the new requirement. Both the approaches will not only result in major rework but also carry the risk of violating the [SOLID](https://springframework.guru/solid-principles-object-oriented-programming/" \t "_blank" \o "SOLID Principles Of Object Oriented Programming)programming principles, specifically [Single Responsibility Principle](https://springframework.guru/single-responsibility-principle/) and [Interface Segregation Principle](https://springframework.guru/interface-segregation-principle/). In such situations, the adapter pattern comes to the rescue.

Using the adapter pattern, you can create a class, say CsvAdapterImpl that will act as an adapter to make both the incompatible interfaces (TextFormattable and CsvFormattable) work together.

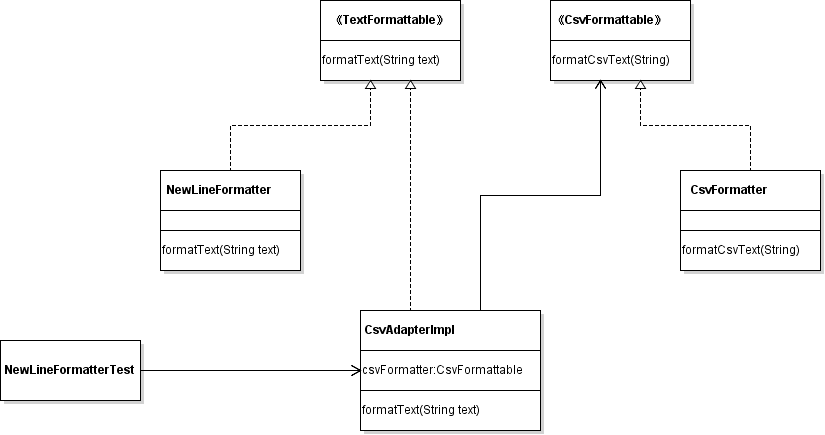
There are two variants of adapters: Object adapter that relies on composition and class adapter that relies on inheritance. As Java does not support multiple inheritance, you have to use object adapter when there are multiple classes that the adapter needs to address. Also, the approach to [favors composition over inheritance](https://en.wikipedia.org/wiki/Composition_over_inheritance)should be the driving factor for using object adapters in Java.

We can summarize the participants of the adapter pattern in the context of the text formatting example, as:

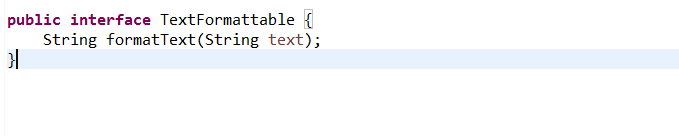
* Target (TextFormattable): The existing interface that clients communicate with.
* Adaptee (CsvFormattable): The new incompatible interface that needs adapting.
* Adapter (CsvAdapterImpl): A class that adapts the Adaptee to the Target.
* Client: Communicates with the Target.

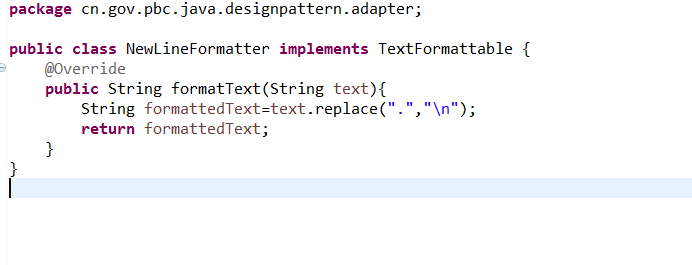
****

## 应用适配器模式（Applying the Adapter Pattern）

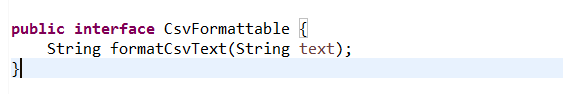


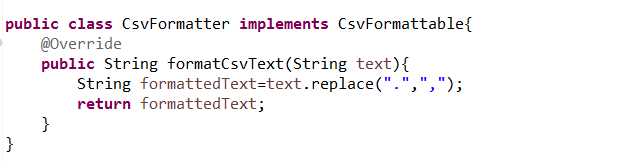
To apply the adapter pattern to the text formatting example, let’s look at the existing Target interface and its implementation class.



 In the example above, the TextFormattable interface declares a single formatText() method that the NewLineFormatter class overrides to return a string formatted with new line characters.

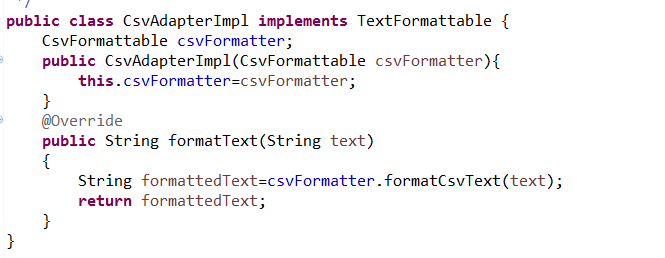
Next, let’s look at the Adaptee interface and its implementation class.



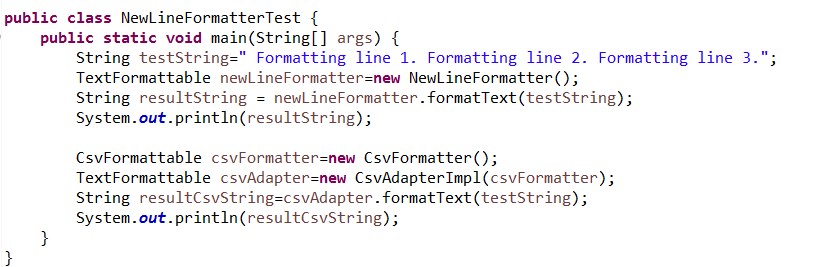


The CsvFormattable interface declares a formatCsvText() method that the CsvFormatter class overrides to return a string formatted as comma separated values.

Coming to the main part, we will now write the Adapter class.



In the CsvAdapterImpl class above, we implemented the TextFormattable interface, which is the Target. We then declared the Adaptee type (CsvFormattable) as a field and initialized it in the constructor. In the overridden formatText() method, we made a call to the formatCsvText() method, and returned back the CSV formatted string to the caller. Let’s now write a unit test for the example.



In Line 17 of the test code above, we called the formatText() method of TextFormattable to format text without using the adapter. It is in Line 20 – Line 22 where we used the adapter. We created a CsvAdapterImplobject passing a CsvFormatter object to its constructor. We then called the formatText() method, which at runtime got forwarded to a call to formatCsvTex() on CsvFormatter.

## 结论（Conclusion）

Some programmers argue that the Adapter pattern is a workaround for a system, which was not well designed keeping into considerations new possibilities. While this is true to some extent, we cannot expect an enterprise application, which will often have a pluggable architecture, to be designed considering all components that might get added in future.  
In enterprise application development, it is likely that you might need to hook in other libraries, APIs, and “off the shelf” components, and if they are not aligned with the existing system, put the adapter pattern to use. After all, being a core GOF pattern, it is a tested and proven solution used over a long period of time.

## 在Spring 中的应用（Adapter Pattern in the Spring Framework）

When doing [Enterprise Application Development](https://springframework.guru/using-the-spring-framework-for-enterprise-application-development/) with the Spring Framework, you will be using adapters built into the framework. [Spring Integration](http://docs.spring.io/spring-integration/reference/html/overview.html) uses JMS adapters to send and receive JMS messages and JDBC adapters to convert messages to database queries and result sets back to messages.  
Spring also uses the adapter design pattern to handle load-time-weaving used in [Aspect-Oriented Programming (AOP)](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html). An adapter is used to inject AspectJ’s aspects to bytecode during class loading done by the servlet container.

# 桥接模式（Bridge Pattern）

“Decouple an abstraction from its implementation so that the two can vary independently.”

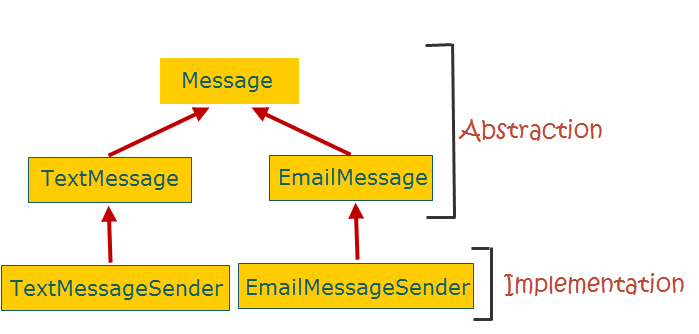
**使抽象部分和实现部分分离，使得它们可以独立的变化。**

## The Bridge Pattern: Introduction

We use abstraction to decouple client code from implementations, and the usual way is to use inheritance. We define an interface or an abstract class and create inheritance hierarchies from it, one for each of the several possible implementations. Although at first look this approach appears logical and nothing wrong in it, abstractions through inheritance isn’t always flexible. When we use inheritance, we are permanently binding the implementation to the abstraction. As a result, any change made to one affects the other. A more flexible way is to separate the abstraction and the implementation, and this is where the bridge pattern comes in.

Similar to the other patterns of the classic [Gang of Four](http://springframework.guru/gang-of-four-design-patterns/) structural pattern family, the objective of the bridge pattern is to identify how to realize relationships between classes and objects in a simple way. The bridge pattern does it by separating the abstraction and the implementation in separate class hierarchies. The bridge between the class hierarchies is achieved through composition.

## Participants of the Bridge Pattern

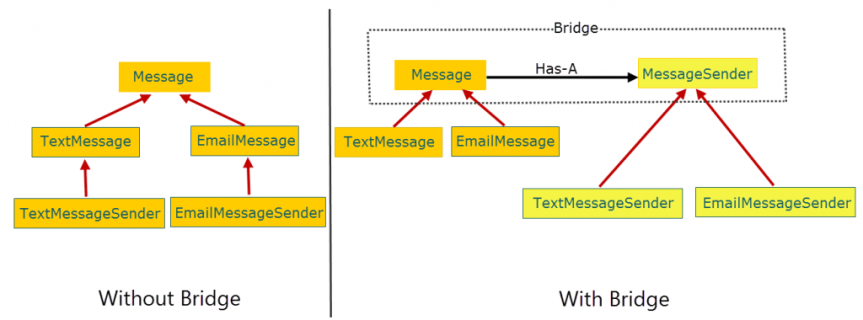
To understand how the bridge pattern works, consider a messaging application that clients can use to send different types of messages, such as a text or an email message. The most intuitive approach is to first create an interface or an abstract base class, Message. Next, we create the derived classes: TextMessage and EmailMessage. Finally, to send messages, we create two message sender classes: TextMessageSender that extends TextMessage and EmailMessageSender that extends EmailMessage. This is how our inheritance hierarchy looks like.  
[](https://i0.wp.com/springframework.guru/wp-content/uploads/2015/06/Bridge01.png?ssl=1)

At first sight there appears nothing wrong in the design above. But if you look deep you will notice that, the abstraction part- the part that clients interact with, and the implementation part- the part that provides the core functionality of sending messages, are tightly integrated. Our design relies on inheritance and one inherent disadvantage is that it breaks encapsulation. As a developer of the EmailMessageSender subclass, you have to know about the internals of the EmailMessage superclass, which means the encapsulation in the superclass is broken.

Our design is also fragile. As an example, if we change the implementation to allow clients to optionally encrypt message before sending, we will need to update the abstraction part to make the encryption functionality available to clients.

Another issue is reusability. If we want to reuse only the implementation (message sending) part in some other application, we will have to take along the abstraction part as extra baggage.

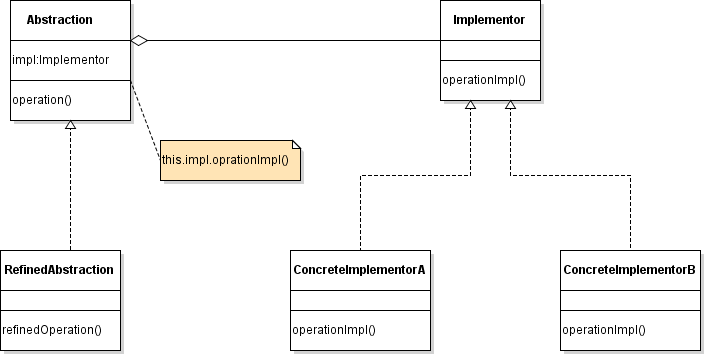
The bridge pattern addresses all such issues by separating the abstraction and implementation into two class hierarchies. This figure shows the design without and with the bridge pattern.

[](https://i2.wp.com/springframework.guru/wp-content/uploads/2015/06/Bridge02.png?ssl=1)

With the bridge pattern, the abstraction maintains a **Has-A** relationship with the implementation instead of a **IS-A** relationship. The **Has-A** relationship is achieved through composition where the abstraction maintains a reference of the implementation and forwards client requests to it.

Let’s summarize the participants of the bridge pattern in the context of the messaging example:

* Abstraction (Message): Is the interface implemented as an abstract class that clients communicates with.
* RefinedAbstraction (TextMessage and EmailMessage): Are classes that implement or extend Abstraction.
* Implementor (MessageSender): Is the interface of the implementation class hierarchy.
* ConcreteImplementor(TextMessageSender and EmailMessageSender): Are concrete subclasses that implements Implementor.

****

## Applying the Bridge Pattern

To apply the bridge pattern to the messaging example, let’s write the abstraction class hierarchy.

## Conclusion

The bridge pattern looks a lot like the [adapter pattern](https://springframework.guru/adapter-pattern/) and is a common cause of confusion. However, while the adapter pattern helps two incompatible interfaces work together, the bridge pattern helps decouple the abstraction and implementation by creating two separate class hierarchies. Also, as stated by GOF “Adapter makes things work after they’re designed; Bridge makes them work before they are.“.

So, when doing [Enterprise Application Development](https://springframework.guru/using-the-spring-framework-for-enterprise-application-development/) with the Spring Framework, if you think about creating deep class hierarchies for different implementations, consider separating the low-level implementations into a separate class hierarchy connected with a bridge. Doing so will make your code more flexible and less fragile to changes. Also, with the bridge pattern in place, it will be much easier to [unit test](https://springframework.guru/unit-testing-junit-part-1/) your code. Without the bridge pattern, in order to test implementation classes, you need their super classes. With the bridge pattern, you can test them independently, and then create mock objects of the implementation classes to test the refined abstraction classes.

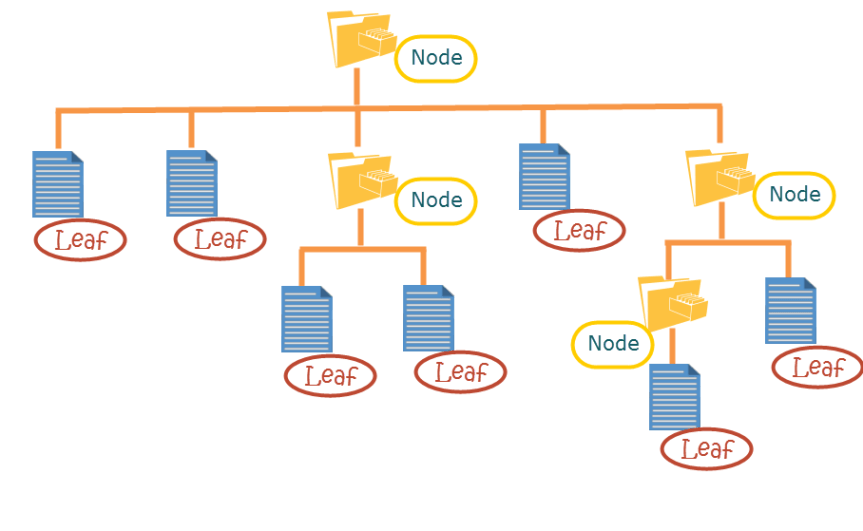
# 组合模式（Composite Pattern）

“Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.”

**组合多个对象形成树形结构以表示整体-部分的结构层次。组合对单个对象和组合对象的使用具有一致性。**

## 简介（Composite Pattern: Introduction）

As a programmer you will deal with hierarchical trees of objects at some point or other. Hierarchical tree structures can come in different flavors, and one can be a tree of components (think as objects) that can be either leaf or node. A leaf is an object that doesn’t have children, while a node does. A node can have one or more leaves or other nodes. This is called recursive composition and can be best illustrated through a file system directory structure.

[](https://i2.wp.com/springframework.guru/wp-content/uploads/2015/07/HierarchialTreeStructure.png?ssl=1)

A challenge in creating such [hierarchical tree structures](https://en.wikipedia.org/wiki/Tree_structure) is to provide clients a uniform way to access and manipulate objects of the tree. Clients should remain unaware whether any operation is being done on a leaf or a node, and this is where the composite design pattern comes in. As an example, composite design pattern can ensure that the process to add or delete a directory (node) and a file (leaf) remains the same for a user.

The composite pattern is part of the classic [Gang of Four](http://springframework.guru/gang-of-four-design-patterns/) structural pattern group. Using this pattern, you can create hierarchical object trees in a uniform manner without going through complexities, such as object casting, type evaluations, and conditional checks to see if one object is independent or contains other objects.

## 参与者（Participants of the Composite Pattern）

To understand how the composite pattern works, consider a shopping store that provides a catalog to help users browse products before buying. Initially, the shopping store had few products manufactured in-house. With expansion, gradually several products got added, some of which are purchased from other manufactures. For different categories of products, sub catalogs were created, and some sub catalogs further had their own sub catalogs. The requirement here is to organize the products and sub catalogs efficiently in a single main catalog. For such requirements, we have the composite pattern.

Let’s review what the GoF authors say about the composite pattern.

* “Compose objects into tree structures to represent part-whole hierarchies”: A part-whole hierarchy is composed of smaller individual objects called **Parts** and larger objects called **Wholes** that are aggregation of **Parts**. What the pattern says is- for part-whole hierarchies, create tree structures to represent relationships between the **Parts** and **Wholes**.
* “Composite lets clients treat individual objects and compositions of objects uniformly”: This means that a client should be able to apply the same operations over both aggregation of objects (**Wholes**) and individual objects (**Parts**).

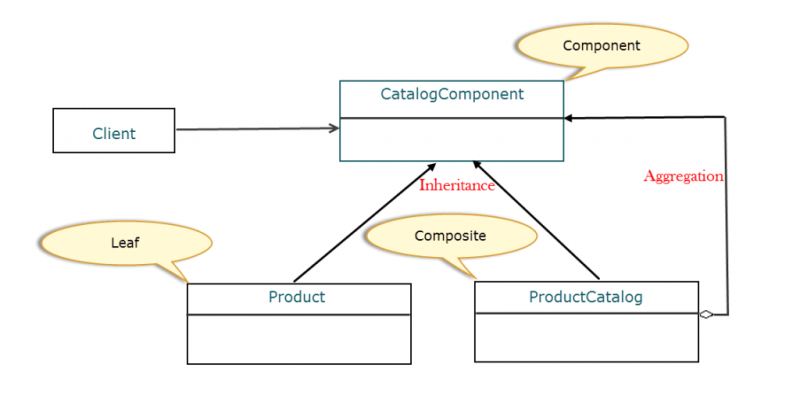
To apply the composite pattern to the catalog example, we can create a tree structure (upside down) and model the root of the tree as an abstract class, CatalogComponent. This class defines the behavior of both individual and composite objects and acts as an interface of the tree to clients. Note that we could have also gone for a Java interface instead, but in the context of our example, that would force classes down the tree to provide implementation of methods not relevant to them. For example, a getProductDiscount() method for a product is not relevant to a catalog, and we don’t expect catalog to implement it.

Next, we need to create a class to model leaves, Product and one to model nodes, ProductCatalog.

Let’s look how our classes map to the participants of the composite pattern:

* **Component** (CatalogComponent): An abstract base class for the objects in the tree structure. This class defines the default behavior for all objects and behaviors to access and manage child components in the tree.
* **Leaf**(Product): Is a class that extends **Component** to represent leaves in the tree structure that does not have any children.
* **Composite** (ProductCatalog): Is a class that extends **Component** to represent nodes (contain children) in the tree structure. This class stores **Leaf** components and implements the behaviors defined in**Component** to access and manage child components. As mentioned earlier, child components can be one or more **Leaf** or other **Composite** components.
* **Client**: Interacts with **Component** to access and manipulate objects in the composition.

This is how the class hierarchy structure of the catalog example will be.

[](https://i1.wp.com/springframework.guru/wp-content/uploads/2015/07/ClassDiagram.png?ssl=1)

As you can see in the figure above, Product and ProductCatalog extend CatalogComponent. So, the relation here is inheritance. ProductCatalog can contain instances of Product and occasionally other ProductCatalog, both of which are CatalogComponent. This relation is called aggregation.

## 应用组合模式（Applying the Composite Pattern）

To apply the composite pattern to the catalog example, let’s write the abstract base class- the **Component**.

**轻量级模式（Flyweight Pattern）**

*“Use sharing to support large numbers of fine-grained objects efficiently.”*

[Design Patterns: Elements of Reusable Object-Oriented Software](https://www.amazon.com/gp/product/0201633612/ref=as_li_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0201633612&linkCode=as2&tag=triatcraft-20&linkId=XRGUDJCGWC6AJNZM)

**简介（Flyweight Pattern: Introduction）**

As a Java programmer you’ve probably heard to “*Think in terms of objects*”. Designing objects to the lowest levels of system “granularity” promote flexibility in the application. But, as a good programmer you also need to think about the performance of the application, in terms of the amount of resources required to run it. When you are dealing with a large number of objects, you need to be concerned about the resources the objects will consume. Each will need memory, and creating each will need CPU cycles.

Imagine you need 1000 car objects in an online car racing game, and the car objects vary only in their current position on the race track. Instead of creating 1000 car objects and adding more as users join in, you can create a single car object with common data and a client object to maintain the state (position of a car). The Flyweight Pattern reduces repeated data, thus reduces memory consumption when dealing with large numbers of objects.

作为一个java程序员，你可能听说过“考虑对象”。在应用中设计对象为最低层次的系统“粒度”来提升灵活性。但是，作为一个好的程序员，你还需要考虑应用程序的性能，根据运行它所需的资源量。当您处理大量对象时，您需要关注对象将要消耗的资源。创建每个对象需要多少内存和 CPU。

假设在在线赛车游戏中你需要1000个汽车对象，并且车的对象仅在其当前位置的赛道上变化。随着用户加入与其创建1000个车对象，不如创建一个单一的存储公共数据的汽车对象和一个客户端对象来保存状态（汽车的位置）。轻量级模式减少了重复数据，从而减少了内存消耗来当处理大量对象。

The Flyweight pattern is a part of the classic [Gang of Four](http://springframework.guru/gang-of-four-design-patterns/) structural pattern family. We have already learned about some of the other patterns of this family: [Adapter](https://springframework.guru/adapter-pattern/), [Bridge](https://springframework.guru/bridge-pattern/), [Composite](https://springframework.guru/composite-pattern-2/), and [Decorator](https://springframework.guru/decorator-pattern/). The Flyweight pattern, similar to those patterns is concerned with relationship between classes and objects, but through a whole new concept of object sharing. What this pattern says is that if an application requires a large number of fine-grained objects, share them instead of repeated instantiation. The object you share is referred as a flyweight.

To become familiar with the Flyweight pattern, a concept to understand is how the internal state of a flyweight is represented. You can categorize the internal state into:

* **Intrinsic data**: Information that is stored in the flyweight object. The information is independent of the flyweight’s context, thereby making it sharable. While applying the pattern, you take all of the objects that have the same intrinsic data and replace them with a single flyweight.
* **Extrinsic data**: Information that depends on the flyweight’s context and hence cannot be shared. Client objects stores and computes extrinsic data and passes it to the flyweight when it needs it.

We will have a closer look at both the preceding types when we cover the participants of the Flyweight pattern, in the context of an example, next.

内部数据：被存储在轻量级对象中的信息。这种信息不依赖于轻量级的上下文，从而使它能够共享。当使用这种模式时，你把所有具有相同内部数据的对象取代为一个单独的轻量级的对象。

外部数据：依赖于与轻量级对象的上下文，并且不能被共享的信息。客户端对象存储并且计算外部数据，并且在需要的时候传递到轻量级对象。

**参与者（Participants of the Flyweight Pattern）**

Continuing with the car racing example, consider we need two types of cars: Midget and Sprint race cars. We can create an abstract base class **RaceCar** with three fields:

* **name** of type **String** for the name of the car
* **speed** of type **int** for the maximum car speed
* **horsepower** of type **int** for the engine power

Objects of a concrete subclass, such as **FlyweightMidgetCar** will store the same set of values for the preceding fields.

This means, these values can be shared between various **FlyweightMidgetCar** objects, and so we consider them as **intrinsic data**. This is common data that will not change and can be shared with other objects.

Consider in the base class, we have a **moveCar(int currentX, int currentY, int newX ,int newY)** method that we override in the **FlyweightMidgetCar** subclass. This method defines the position of each **FlyweightMidgetCar**object at a given time, and hence cannot be shared. So we consider the car position passed to **moveCar()** as extrinsic data.

Now that we have categorized the internal states of **FlyweightMidgetCar** into intrinsic and extrinsic data, we can make it a flyweight object. Similarly, we can make **FlyweightSprintCar** or any other object that extend**RaceCar** as flyweight objects.

继续上面赛车的例子，考虑我们需要两种类型的赛车：Midget and Sprint。我们可以创建一个抽象的基类RaceCar，它包括了三个属性: name,speed,horsepower

具体子类的对象，比如[FlyweightMidgetCar]会为上面的属性存储相同的一组值。意思就是，值可以在不同的FlyweightMidgetCar对象之间共享。这就是公共数据，它们不会变化并且能够被不同的对象共享。

考虑在基类里有一个方法[**moveCar(int currentX, int currentY, int newX ,int newY)** ],我们在子类**FlyweightMidgetCar** 中重写了。这个方法定义了每一个FlyweightMidgetCar对象在指定时刻的位置，因此不能被共享。所以我们认为被传递到moveCar()中的位置数据是外部数据。

现在我们把FlyweightMidgetCar的内部状态进行了分类为内部数据和外部数据，我们可以使它成为一个轻量级对象。同样，我们也可以使**FlyweightSprintCar** 或者其他继承于RaceCar的对象成为轻量级对象。

To manage our flyweight objects, we need a factory – a class that uses a pool (implemented as some data structure) to store flyweight objects. When client request a flyweight object for the first time, the factory instantiates a flyweight, initializes it with its intrinsic data, and puts it in the pool, before returning the flyweight to the client. For subsequent requests, the factory retrieves the flyweight from the pool and returns it to the client. We will name the class **CarFactory**.

We will also need a client class, **RaceCarClient** that will retrieve a flyweight (already initialized with its intrinsic data) from the factory and pass the extrinsic data to the flyweight. In the context of our example, the client will manage and pass the position (extrinsic data) of a car by calling the **moveCar(int currentX, int currentY, int newX ,int newY)** method of the flyweight.

为了管理我们的轻量级对象，需要一个工厂---使用池（实现了一些数据结构）来存储轻量级对象的一个类。当客户端第一次请求一个轻量级对象时，工厂实例化一个轻量级对象，初始化内部数据，并且在返回客户端之前把它放到池中。针对

随后的请求，工厂从池中取出轻量级对象并且返回到客户端。我们命名这个类为CarFactory。

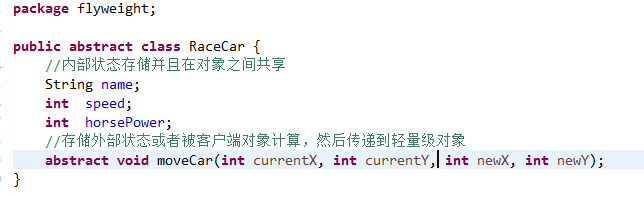
我们还需要一个客户端类**RaceCarClient** ，它会从工厂中得到一个轻量级对象（已经初始化带有内部数据）并且传递外部数据到这个对象。在我们的例子中，客户端会管理和传递汽车的位置（外部数据）通过调用轻量级对象的**moveCar()方法。**

We can now categorize the participants of the Flyweight pattern as:

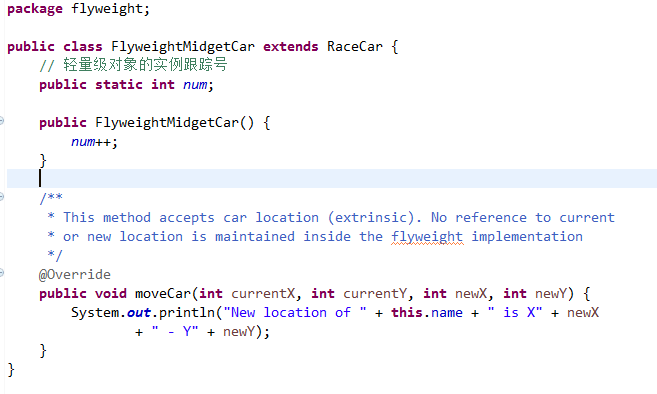
* **Flyweight** (**RaceCar**): Abstract class or interface for flyweight objects. Declares method through which flyweights can receive and act on extrinsic state. Although **Flyweight** enables sharing, it is not mandatory that all **Flyweight** subclasses must be shared.
* **ConcreteFlyweight**(**FlyweightMidgetCar** and **FlyweightSprintCar**): Extends/Implements **Flyweight** to represent flyweight objects that can be shared.
* **FlyweightFactory**(**CarFactory**): Creates and manages flyweight objects. When a client requests a flyweight object, **FlyweightFactory** provides an existing one or creates a new one, if it does not exists.
* **Client**(**RaceCarClient**): Requests **FlyweightFactory** for a flyweight object, and then computes and passes the extrinsic data to it.

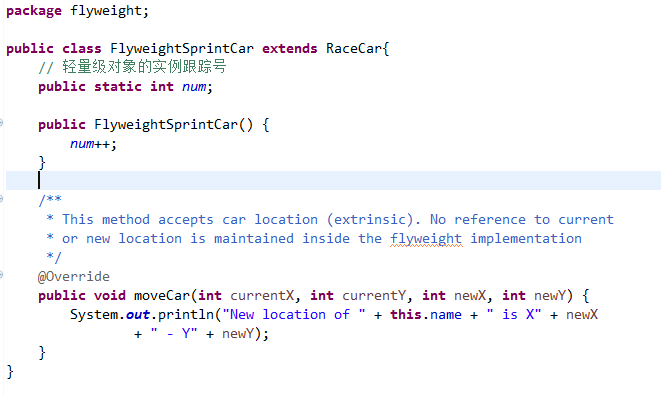
**应用轻量级模式（Applying the Flyweight Pattern）**

To apply the flyweight pattern to our example, let’s first write the **Flyweight**.



In the RaceCar abstract class, we wrote the fields to hold the intrinsic data and an abstract moveCar()method through which the client will pass extrinsic data to flyweight objects. Notice how the extrinsic data are values that will not change. The horsepower of the car is not a value that will change while application is in use. Our race cars can have different horsepower values, but once set, this is not something what will be changing. Let’s write the concrete FlyweightMidgetCar and FlyweightSprintCar classes.

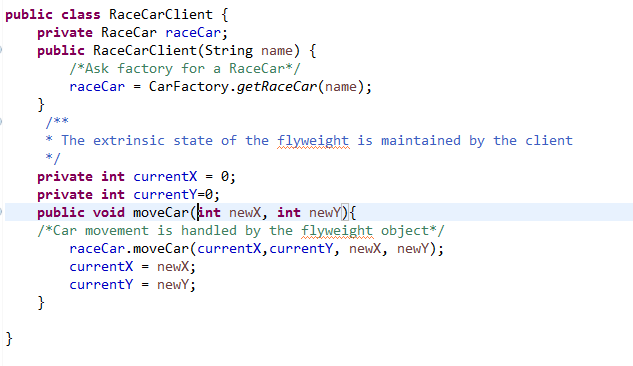




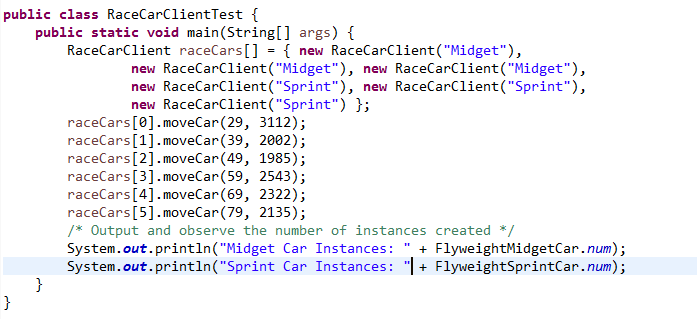
In both the classes above, we maintained a counter through their constructor that will later enable us to check the number of flyweight objects created. We have also overridden the moveCar() method to output the new location of a car. **NOTE** – Notice how the moveCar()  method does not store state in the flyweight object.  Let’s now write the CarFactory class – the **FlyweightFactory**.



In the CarFactory class, we created a HashMap to store flyweight (RaceCar) objects and written a getRaceCar() method- a [factory method](https://springframework.guru/factory-method-design-pattern/) that returns a flyweight object. In the factory method, we checked the HashMap for the client specified key. If an entry is present, we returned the corresponding flyweight object. If an entry is not present, we used a switch statement to create and initialize a flyweight object, which can either be FlyweightMidgetCar or FlyweightSprintCar, based on the client specified value. We then stored the object in the HashMap, and finally returned the object to the client.



In the client class above, we have a reference to a flyweight object and initialized it in the constructor through a call to the getRaceCar() factory method. Once we got the shared flyweight object, we calculated the current position (extrinsic data) and passed it to the flyweight object through a call to the moveCar() method of the object (Line 16). Notice how the client code is responsible for maintaining the state of the individual object. If state was not maintained in the client object, but in the flyweight object, the changes would be seen by all objects having a reference to the flyweight object. Clearly this is not what we want. Things like location will change for each client object, while things like horsepower do not change and can be shared across the client objects. Let’s write some test code to see how this behaves in actual use.



In the test code we created an array of RaceCarClient with six RaceCarClient instances and called the moveCar() method on each instance. We also printed out the number of FlyweightMidgetCar and FlyweightSprintCar instances created.

## 结论（Conclusion）

The flyweight pattern might appear complex to some, and I agree it is one of the lesser used design patterns. Its probably used even less as computers have become more and more powerful. But when you’re dealing with large scale systems, the Flyweight Pattern is something to consider. It can help save system resources. There’s a lot of buzz about ‘Big Data’ in the industry. What happens when you’re processing big data?? You may see a use case for the Flyweight Pattern.

When you are developing [Enterprise Applications with the Spring Framework](https://springframework.guru/using-the-spring-framework-for-enterprise-application-development/), if you are in situations where you have a large amount of objects sharing some common state and consuming large amount of memory, the Flyweight pattern may be exactly what you are looking for.

# 策略模式（Strategy Pattern）

“Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

定义一系列算法，将每一个算法封装起来，并使它们可互换。策略允许算法独立于使用它的客户端。

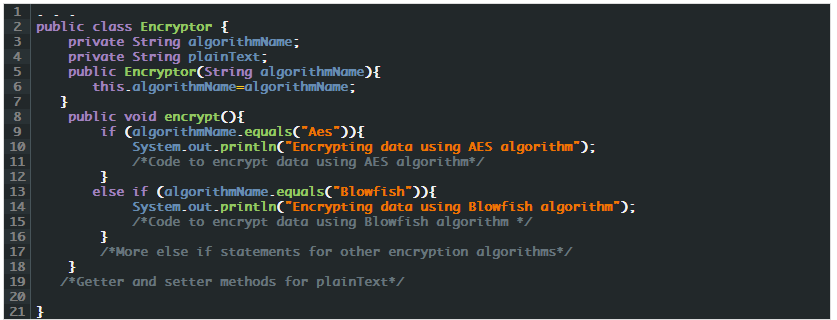
The Behavioral pattern family of the [Gang of Four](http://springframework.guru/gang-of-four-design-patterns/) design patterns address responsibilities of objects in an application and how they communicate between them at runtime. The Behavioral patterns that I already wrote in this series of the GoF patterns are the [Command](https://springframework.guru/gang-of-four-design-patterns/command-pattern/), [Chain of Responsibility](https://springframework.guru/gang-of-four-design-patterns/chain-of-responsibility-pattern/), [Iterator](https://springframework.guru/gang-of-four-design-patterns/iterator-pattern/), [Mediator](https://springframework.guru/gang-of-four-design-patterns/mediator-pattern/), [Interpreter](https://springframework.guru/gang-of-four-design-patterns/interpreter-pattern/),[Memento](https://springframework.guru/gang-of-four-design-patterns/memento-pattern/), [Observer](https://springframework.guru/gang-of-four-design-patterns/observer-pattern/) and [State](https://springframework.guru/gang-of-four-design-patterns/state-pattern/) patterns. In this post, I will discuss the Strategy Pattern – one of the most fundamental design pattern that all programmers should possess in their design toolkit. The intent of the Strategy Pattern, as mentioned above, suggests that this pattern is applicable when you have multiple algorithms and you want to treat them as independent objects that can be interchanged dynamically at runtime to achieve high [cohesion](https://en.wikipedia.org/wiki/Cohesion_%28computer_science%29) and loose [coupling](https://en.wikipedia.org/wiki/Coupling_%28computer_programming%29) in your application.

GOF设计模式中的行为模式家族处理应用中对象的职责和他们在运行时如何通信。这些行为模式在之前的已经写过了，他们是命令模式、责任链模式、迭代器模式、中介者模式、解释器模式、观察者模式、备忘录模式和状态模式。在这个章节将要讨论策略模式，最基础的设计模式之一，所有的开发者在他们的开发包中应该具备。正如上面提到的，策略模式的含义是，当你有多个算法并且你想作为独立的对象对待它们，这些对象可以在运行时动态的互换来达到应用程序的高内聚和松耦合时，建议你使用策略模式。

## 简介（Strategy Pattern: Introduction）

In enterprise applications, you will often have objects that use multiple algorithms to implement some business requirements. A common example is a number sorting class that supports multiple sorting algorithms, such as bubble sort, merge sort, and quick sort. Similarly, a file compression class can support different compression algorithm, such as ZIP, GZIP, LZ4, or even a custom compression algorithm. Another example can be a data encryption class that encrypts data using different encryption algorithms, such as AES, TripleDES, and Blowfish. Typically, programmers tend to bundle all the algorithm logic in the host class, resulting in a monolithic class with multiple switch case or conditional statements. The following example shows the structure of one such class that supports multiple algorithms to encrypt data.

在企业应用程序中，通常会有使用多个算法实现某些业务需求的对象。一个常见的例子是支持多种排序算法的数字排序类，如冒泡排序、归并排序和快速排序。同样的，一个文件压缩类可以支持不同的压缩算法，如ZIP，GZIP，lz4，甚至一个自定义的压缩算法。另一个例子是一个数据加密类，对数据进行加密，使用不同的加密算法，如AES，TripleDES和密码学。通常情况下，程序员往往把所有的算法逻辑写在宿主类，导致在一个类中有多个开关或条件判断语句。下面的示例展示了一个这样的类的结构，该类支持多个算法对数据进行加密。



The above Encryptor class has conditional statements for different encryption algorithms. At runtime, the code loops through the statements to perform encryption based on the client specified algorithm. The result is a tightly coupled and rigid software that is difficult-to-change. You can imagine the consequences if you try to implement a new encryption algorithm, say TripleDES or a custom encryption algorithm. You’d have to open and modify the **Encryptor** class. Also, if an existing algorithm needs to be changed, the **Encryptor** class will again require modification. As you can see, our **Encryptor** class is a clear violation of the [Open Closed](https://springframework.guru/principles-of-object-oriented-design/open-closed-principle/) principle – one of the [SOLID](https://springframework.guru/solid-principles-object-oriented-programming/) design principles. As per the principle, new functionality should be added by writing new code, rather than modifying existing code. The violation occurred because we did not follow the fundamental tenet of Object-Oriented (OO) programming practice that states “encapsulate what varies”.

上面的加密类有不同的加密算法的条件语句。在运行时，代码依次通过语句判断执行客户端指定算法进行加密。这是一个紧耦合和死板的软件，是很难改变。你能想象到的后果，如果你想实现一个新的加密算法，比如说TripleDES或自定义的加密算法。你会打开并修改加密类。同样，如果一个现有的算法需要改变，加密类将再次需要修改。你可以看到，我们的加密类是明显违反了开闭原则–面向对象的设计原则。按照原则，新的功能应该通过编写新的代码来增加，而不是修改现有代码。发生违反原则的情况是因为我们没有遵循面向对象（OO）编程实践的基本原则，即“封装变化”。

Such pitfalls in enterprise applications result in rippling effects across the application making the application fragile, and you can avoid them by using the Strategy pattern. Using the Strategy pattern, we define a set of related algorithm and encapsulate them in classes separated from the host class (**Encryptor**). Clients can choose the algorithm to use at run time. By doing so, we can easily add a new algorithm or remove an existing one without modifying the other existing algorithms or the host class. Each of the algorithm classes adhere to the [Single Responsibility](https://springframework.guru/principles-of-object-oriented-design/single-responsibility-principle/) principle, another SOLID principle as they will only be concerned with encrypting data with a specific algorithm, which is currently lacking in our **Encryptor** class. In addition, with smaller algorithm classes, unit testing becomes easier to focus on testing one particular situation.

企业应用程序中的这些缺陷会导致整个应用程序的涟漪效应，使应用程序变得脆弱，并且可以通过使用策略模式避免它们。使用策略模式，我们定义了一组相关的算法，将它们封装在与主类（**Encryptor**）分开的类中。客户端可以选择在运行时使用的算法。通过这样做，我们可以轻松地添加一个新的算法或删除而不用修改其他现有的算法或主机类。每个算法类坚持单一责任原则，另一个面向对象的原则，他们只关注一个特定的算法对数据进行加密，这是目前在我们**Encryptor** 类中所缺乏的。此外，较小的算法类，单元测试变得更容易专注于测试一个特定的情况。

## 参与者（Participants of the Strategy Pattern）

Using our example of data encryption, we will first implement the interface that will be used by all of the different encryption algorithm-specific classes. Let’s name the interface **EncryptionStrategy** and name the algorithm specific classes **AesEncryptionStrategy** and **BlowfishEncryptionStrategy**. Ultimately, these are our strategies.

We will next refactor the **Encryptor** class to remove all conditional statements and delegate any encryption request to an algorithm-specific class that the client specifies.

We can now summarize the participants of the strategy pattern as:

* **Strategy** (**EncryptionStrategy**): Is an interface common to all supported algorithm-specific classes.
* **ConcreteStrategy** (**AesEncryptionStrategy** and **BlowfishEncryptionStrategy**): Implements the algorithm using the **Strategy** interface.
* **Context** (**Encryptor**): Provides the interface to client for encrypting data. The **Context** maintains a reference to a **Strategy** object and is instantiated and initialized by clients with a **ConcreteStrategy** object.

使用我们的数据加密的例子，我们将首先实现的接口，接口将用于所有不同的加密算法特定的类。让我们命名接口为**EncryptionStrategy** ，命名算法的具体类**AesEncryptionStrategy** 和**BlowfishEncryptionStrategy**。最终，这些是我们的策略。

接下来我们将重构加密类，删除所有条件语句和委托任何加密算法的客户指定的具体要求。

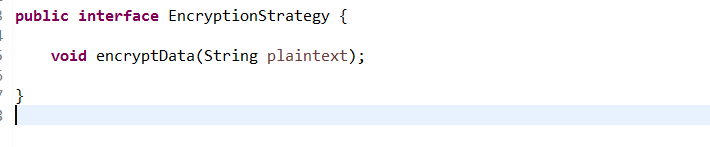
现在我们可以将策略模式的参与者归纳为：

* 策略类（encryptionstrategy）：支持所有特定算法的接口类。
* 具体策略类（AesEncryptionStrategy和blowfishencryptionstrategy）：实现算法使用策略接口。
* 上下文（**Encryptor**）为客户端加密数据提供接口。上下文维护一个对**Strategy** 对象的引用、通过客户端使用**ConcreteStrategy** 实例化和初始化一个对象。

## 应用策略模式（Applying the Strategy Pattern）

Let’s now apply the Strategy pattern to implement the same requirement done for the **Encryptor** class we wrote earlier. We will start with the **Strategy** interface and then move to the **ConcreteStrategy** classes.

**让我们使用策略模式来实现上面Encryptor** 类实现的同样的需求。**我们从Strategy** 接口开始，然后转移到**ConcreteStrategy** 类

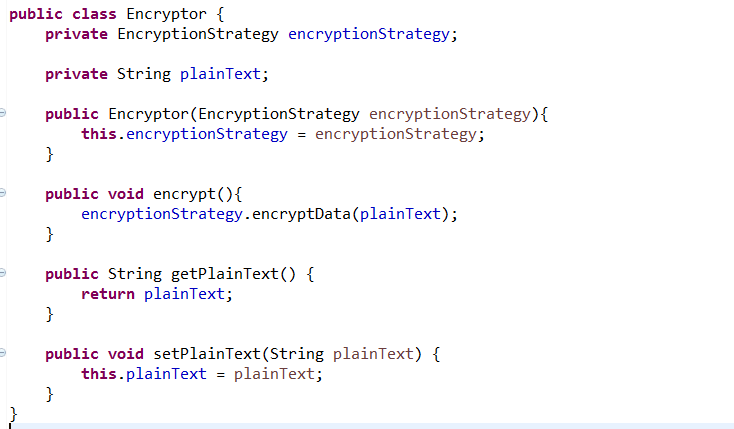






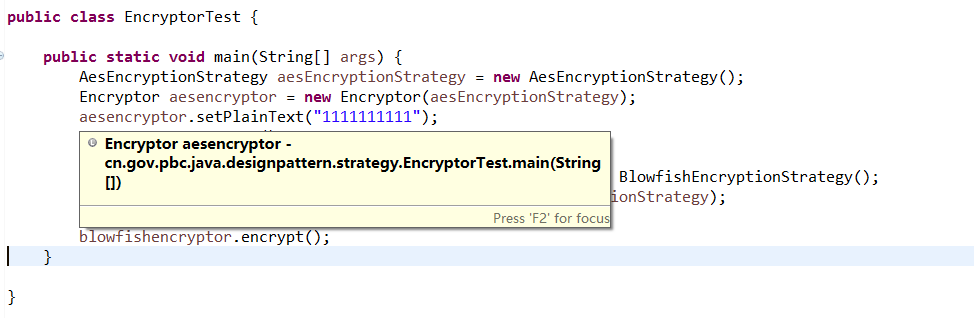
In the EncryptionStrategy interface we wrote above, we declared a single encryptData() method that both the AesEncryptionStrategy and BlowfishEncryptionStrategy implements. In the AesEncryptionStrategy class, we used the AES encryption algorithm to symmetrically encrypt the string passed to encryptData(). We performed the encryption using the cryptography classes of the javax.crypto package. After encryption, we printed out the encrypted string. We performed the same function in the BlowfishEncryptionStrategy class, but this time using the Blowfish encryption algorithm.

Next, we will write the Context – The **Encryptor** class.



Notice that our Encryptor class now doesn’t have any conditional statements. It maintains a reference to Strategy, which is initialized through the constructor. Notice, we don’t have any reference to any**ConcreteStrategy** (**AesEncryptionStrategy** and **BlowfishEncryptionStrategy**) classes. We are adhering to the OOP programming practice of “*Program to an interface, not an implementation*”. So, if we later want to accommodate a new encryption algorithm, say Triple DES, all we need to do is create a new **ConcreteStrategy**class, say **TripleDesEncryptionStrategy**. This class will implement our **EncryptionStrategy** interface and we are good to go without modifying our **Encryptor** class. Our Encryptor class is now open for extension and closed for modification – It’s now following the Open Close principle. The Encryptor class also contains a encrypt() method that clients will call to perform encryption. Instead of putting in any encryption logic in this method, we delegated any call to this method to the associated algorithm-specific object.

Let’s write a test class for our example.



In the EncryptorTest class above, from Line 15 – 18 we created an AesEncryptionStrategy object and passed it to the constructor while instantiating the Encryptor class. We then called the setPlainText()method of Encryptor to set the plain text to encrypt and then called the encrypt() method to perform the encryption using AES. From Line 20 – Line 23, we performed the same steps but this time, we switched the encryption algorithm to Blowfish by using the BlowfishEncryptionStrategy class.

## 总结（Summary）

To many, the Strategy and State patterns appear similar. It’s true that the structure of both the patterns are similar. It’s the intent that differs – that is, they solve different problems. The State pattern aims to facilitate state transition while the aim of the Strategy pattern is to change the behavior of a class by changing internal algorithm at runtime without modifying the class itself.

There is a lot of debate around the use of the Strategy Pattern with Spring. Often you’ll see the Strategy Pattern used in conjunction with Dependency Injection, where Springs IoC container is making the choice of which strategy to use. Different data sources as a great example. Using a H2 data source for local development is one strategy. Using MySQL for production is another strategy. Which one is to use at runtime is up to the Spring IoC container.

Spring 3, introduced a type [conversion factory](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/validation.html#core-convert). In this, you provide a type converter, which implements the Converter interface. At run time, your code can ask the converter factory for the proper converter. This sure sounds like the Strategy Pattern, doesn’t it?

The Strategy Pattern is one of those GoF patterns you’ll often encounter without realizing it’s a classic GoF pattern. Its a pattern that will get used simply by practicing widely accepted OO development principles.

# 中介者模式（Mediator Pattern）

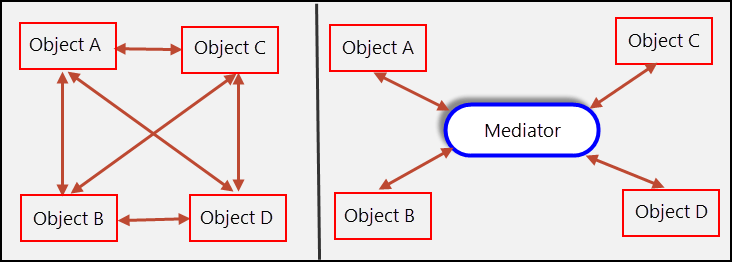
“Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.”

[Design Patterns: Elements of Reusable Object-Oriented Software](https://www.amazon.com/gp/product/0201633612/ref=as_li_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0201633612&linkCode=as2&tag=triatcraft-20&linkId=XRGUDJCGWC6AJNZM)

The Mediator Pattern, which is similar to the [Command](https://springframework.guru/command-pattern/), [Chain of Responsibility](https://springframework.guru/chain-of-responsibility-pattern/), and [Iterator](https://springframework.guru/iterator-pattern/) patterns, are part of the Behavioral pattern family of the [Gang of Four](http://springframework.guru/gang-of-four-design-patterns/) design patterns. Behavioral patterns address responsibilities of objects in an application and how they communicate between them. The Mediator pattern allows the loose coupling between set of objects in an application by handling the interactions between the objects.

## Behavioral Patterns: Introduction

Well-designed enterprise applications are composed of lightweight objects with specific responsibilities divided between them in accordance to the [Single Responsibility](https://springframework.guru/single-responsibility-principle/)principle, one of the [SOLID](https://springframework.guru/solid-principles-object-oriented-programming/) design principles. However, the benefits of an application composed of a large number of small objects comes with a challenge – the communication between them. Objects need to communicate to perform business requirements, and as the number of objects grow, connections between the objects required can quickly become unmanageable. In addition objects communicating between them needs to know about each other – they are all tightly coupled that violates the basics of the [SOLID](https://springframework.guru/solid-principles-object-oriented-programming/) principles. The Mediator pattern is a proven solution to address such recurring challenges and the problems arising out of them.

The Mediator pattern says that instead of allowing a set of objects to directly interact between them, define an object (mediator) that will handle the interactions. What the mediator essentially says to such set of objects is “talk with me instead of talking among yourselves”. This figure conceptually shows how objects interact without and with a mediator.  
[](https://i1.wp.com/springframework.guru/wp-content/uploads/2015/11/Mediator.png?ssl=1)

The Mediator pattern, as shown in the above figure employs a mediator object to enable other objects of the application to interact without knowing each other’s identities. The mediator also encapsulates a protocol that objects can follow.

It is the mediator object that encapsulates all interaction logic between objects in a system. Therefore, if an existing object is updated with new interaction rules or a new object is added to the system, it is only the mediator object that you need to update. In the absence of the mediator, you would need to update all the corresponding objects that which to interact. Through the use of the Mediator Pattern your code becomes more encapsulated, thus changes are not as extensive.

Let’s now identify the participants of the Mediator Pattern.

## Participants of the Mediator Pattern

Imagine a war zone where armed units are moving into the enemy’s territory. Armed units can include soldier, tank, grenadier, and sniper units. The strategy being employed is that whenever one unit attacks, other units should stop attacking and take cover. To do so, the unit that is currently attacking needs to notify the other units.

In the programming world, you can easily model this requirement by creating classes for each armed unit. In each class, whenever its object is about to start attacking, you can implement the logic to notify objects of the other classes. Now, imagine that a new unit joins in. The consequence – all the existing classes need to be updated. In the worst case, imagine that the battle tactics change so that both the soldier and sniper units can now attack simultaneously. Again, the consequence is a lot of changes to the code base. We can address such challenges, similarly as done in real life. Place a Commander in a base camp that will act as the mediator. All units, instead of communicating between themselves will communicate with the mediator. The mediator based on the notifications received from some units can send request to one or more other units to perform actions as requirements demand.

Let’s model the mediator with a **Commander** interface and a concrete **CommanderImpl** subclass of the interface. In the **Commander** interface, we can declare methods to send messages to objects representing armed units and also methods that armed unit’s objects can communicate with the **Commander**. In the**CommanderImpl** subclass, we will maintain reference to the objects representing armed units and override the methods of **Commander**. For our example, let’s create an interface as **ArmedUnit** whose implementing classes will represent specific armed units. We will create two such concrete classes: **SoldierUnit** and **TankUnit**, and each of them will hold references to **Commander**.

With our example, let’s look at the participants of the Mediator pattern.

* **Mediator** (**Commander**): Is an interface that declares methods for communicating with **Colleague** objects.
* **ConcreteMediator** (**CommanderImpl**): Implements **Mediator**. This class maintains and coordinates**Colleague** objects.
* **Colleague**(**SoldierUnit** and **TankUnit**): Communicates with its **Mediator** when their state changes and responds to requests from the **Mediator**.

## Applying the Mediator Pattern

Going ahead with our example on armed units, let’s apply the Mediator pattern to it. We will start with the**Mediator** interface followed by the **ConcreteMediator** classes.

**单一职责原则**

**Factory** – Spring uses factory pattern to create objects of beans using Application Context reference

// Spring uses factory pattern to create instances of the objects  
BeanFactory factory = new XmlBeanFactory(new FileSystemResource("spring.xml"));  
Triangle triangle = (Triangle) factory.getBean("triangle");  
triangle.draw();

**Proxy** – used heavily in AOP, and remoting.

**Singleton** – by default, beans defined in spring config file (xml) are only created once. No matter how many calls were made using getBean() method, it will always have only one bean. This is because, by default all beans in spring are singletons.  
This can be overridden by using Prototype bean scope.Then spring will create a new bean object for every request.

**Model View Controller** – The advantage with Spring MVC is that your controllers are POJOs as opposed to being servlets. This makes for easier testing of controllers. One thing to note is that the controller is only required to return a logical view name, and the view selection is left to a separate ViewResolver. This makes it easier to reuse controllers for different view technologies.

**Front Controller** – Spring provides DispatcherServlet to ensure an incoming request gets dispatched to your controllers.

**View Helper** – Spring has a number of custom JSP tags, and velocity macros, to assist in separating code from presentation in views.

**Template method** – used extensively to deal with boilerplate repeated code (such as closing connections cleanly, etc..). For example JdbcTemplate, JmsTemplate, JpaTemplate.

# 命令模式（Command Pattern）

“Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.”

[Design Patterns: Elements of Reusable Object-Oriented Software](https://www.amazon.com/gp/product/0201633612/ref=as_li_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0201633612&linkCode=as2&tag=triatcraft-20&linkId=XRGUDJCGWC6AJNZM)

The Command Pattern is one of the 11 design patterns of the [Gang of Four](http://springframework.guru/gang-of-four-design-patterns/) Behavioral pattern family. We’ve already learned about the [Chain of Responsibility](https://springframework.guru/chain-of-responsibility-pattern/) pattern, and similar to it and the other patterns of the family, the Command Pattern addresses responsibilities of objects in an application and how they communicate between them.

## 简介（Command Pattern: Introduction）

Communication between objects in enterprise applications starts from an object sending a request to another object. The object receiving the request can either process the request and send back a response or forward the request to another object for processing. Typically requests are made by the invoker (the object making the request) through method calls on the object that processes the request, which we will refer as the receiver. As a result, the invoker and the receiver are tightly coupled. This violates the [SOLID](https://springframework.guru/solid-principles-object-oriented-programming/) design principles that advocates loosely coupled components to ensure that changes made to one component does not affect the other components of the application.

The Command Pattern is a proven solution that addresses such recurring problems of tightly coupled invoker and receiver components in applications. This pattern states that requests should be encapsulated as objects that like any other objects can be stored and passed around the application. Requests encapsulated as objects are known as commands.

企业应用程序中对象之间的通信从一个将请求发送到另一个对象的对象开始。接收请求的对象可以处理请求并将响应发送回或将请求转发给另一个对象进行处理。通常请求被调用是通过调用处理请求对象的方法，我们将其称为接收者。因此，该调用者和接收者是紧耦合。这违反了坚实的设计原则，主张松耦合的组件，以确保对一个组件的变化不影响应用程序的其他组件。

命令模式是一种行之有效的解决方案，这个方案解决了应用程序中调用者和接受者重复出现的紧密耦合的问题。此模式规定请求应封装为对象，可以像其他对象一样被存储并传递到应用程序周围。被封装为对象的请求称为命令。

In the command pattern, the invoker issues commands without knowing anything about the receiver. In fact the invoker issuing the command doesn’t even know what operation will be carried out on issuing a command. Let’s look at it from programming point of view.

A command object basically has a **execute()** method and an optional **undo()** method. The **execute()** method executes some operation on the receiver while the **undo()** method reverses the current operation. The implementation of the operation is done by the receiver. The invoker only sets itself up with a command object and invokes a command by calling the **execute()** method. The invoker does not know what the **execute()**method will do.

Imagine that the invoker is a switch, When the invoker invokes a command by calling the **execute()** method of a command object, it might turn on a light or even start a generator. What this essentially means is that the same invoker (switch) can be used to invoke commands to perform actions on different receivers (devices). We only need to create the appropriate command object and set it on the invoker. As apparent, by applying the Command Pattern we can take reusability to another level and this is made possible due to the loose coupling between the invoker and receiver and the command object that acts as the interface between them.

在命令模式中，调用者发出命令，但是不用知道任何关于接受者的信息。实际上调用者发出命令甚至不知道什么操作将要被执行。下面让我们从程序的视角来看下。

一个命令对象基本上都有一execute()方法和一个可选的undo()方法。execute()方法在接受者上执行一些操作，而undo()撤销当前的操作。操作的实现是接受者做的。调用者本身只设置一个命令对象，然后通过execute()方法来调用这个命令。调用者并不知道execute()方法将要做什么。

想象这个调用是一个开关，当调用者通过调用命令对象的execute()方法调用命令时，它可能会打开一盏灯或甚至启动发电机。这基本上意味着相同的调用（开关）可以用来调用命令来执行不同的接收者的行动（设备）。我们只需要创建相应的命令对象并设置它的调用程序。显然，采用命令模式我们可以把可重带到另一个水平，这也使得调用者和接受者之间的松耦合变为可能，命令对象充当了他们之间的接口。

## 参与者（Participants of the Command Pattern）

Before drilling into the participants of the Command Pattern and their roles, let’s start with an analogy of remote controlled toys.

Imagine a toy manufacturing company, which recently decided to manufacture remote controlled toys. They started with a remote controlled car. The remote control has three buttons: On, Off, and Undo. The remote control is programmed so that the On button moves the car, the Off button stops it, and the Undo button reverses the current action of the car. The remote controlled car was a success and the management next decided to manufacture a remote controlled rotating top. Again, along with the top, a new remote control programmed to start and stop rotating and reverse the current action of the top was manufactured. Riding on the success of remote controlled toys, the management decided to manufacture 200 varieties of remote controlled toys each with a remote control programmed differently to operate its corresponding toy. Imagine yourself in the programming team. You need to create 200 different programs for the remote controls each designed to operate on a specific toy. Apparently a bad design decision because of the inability to reuse code. The reason is also apparent. Each remote control is tightly coupled with the toy it operates on. What we need is a remote control that is programmed to operate on all the toys, and here is where the Command Pattern comes to the rescue.

Let’s look how we can apply the pattern as a solution. The toys are the receivers and perform actions. Let us model two receivers as **Car** and **RotatingTop**. These classes have behaviors in form of methods to perform actions. For example, the **Car** receiver will have methods to move and stop and the **RotatingTop** will have methods to start and stop rotating.

We will next model the commands that will trigger the actions on the receivers. We start with a **CommandBase**interface with two methods **execute()** and **undo()**. Each concrete sub classes of **CommandBase** represents a command to trigger an action on the receiver, and to do so it maintains a reference to the receiver whose action it will trigger. For the **Car** and **RotatingTop** receivers, let us model the concrete command classes as**CarMoveCommand**, **CarStopCommand**, **TopRotateCommand**, and **TopStopRotateCommand**. Finally we will model the invoker as a **RemoteControl** class. This class is not aware of any concrete command class. Whenever one of its button is pressed, the method handling the press event is injected with an appropriate command object at runtime and the invoker calls the **execute()** method on it.

In the context of the remote controlled toys example, let’s summarize the participants of the Command Pattern.

* **Command** (**CommandBase**): Is an interface for executing an action.
* **ConcreteCommand** (**CarMoveCommand**, **CarStopCommand**, **TopRotateCommand**, and**TopStopRotateCommand**): Are concrete classes that implements **Command** and defines the **execute()**and **undo()** methods to communicate with receivers for performing an action and undoing it respectively.
* **Invoker**(**RemoteControl**): Asks **Command** to carry out the action.
* **Receiver** (**Car** and **RotatingTop**): Performs the action based on the command it receives.
* **Client**: Creates a **ConcreteCommand** object and sets its receiver.

The following figure shows the relationship between the participants.

# 访问者模式（Visitor Pattern）

“Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.”

说明一个操作执行于一个对象结构的成员中，访问者模式让你定义一个类的新操作而无须改变它操作的这些成员类。

The Behavioral pattern family of the [Gang of Four](http://springframework.guru/gang-of-four-design-patterns/) design patterns address responsibilities of objects in an application and how they communicate between them at runtime. The other GoF Behavioral patterns that I wrote in this series of GoF patterns are:

* [Chain of Responsibility](https://springframework.guru/gang-of-four-design-patterns/chain-of-responsibility-pattern/): Delegates commands to a chain of processing objects.
* [Command](https://springframework.guru/gang-of-four-design-patterns/command-pattern/): Creates objects which encapsulate actions and parameters.
* [Interpreter](https://springframework.guru/gang-of-four-design-patterns/interpreter-pattern/): Implements a specialized language.
* [Iterator](https://springframework.guru/gang-of-four-design-patterns/iterator-pattern/): Accesses the elements of an object sequentially without exposing its underlying representation.
* [Mediator](https://springframework.guru/gang-of-four-design-patterns/mediator-pattern/): Allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
* [Memento](https://springframework.guru/gang-of-four-design-patterns/memento-pattern/): Provides the ability to restore an object to its previous state.
* [Observer](https://springframework.guru/gang-of-four-design-patterns/observer-pattern/): Is a publish/subscribe pattern which allows a number of observer objects to see an event.
* [State:](https://springframework.guru/gang-of-four-design-patterns/state-pattern/) Allows an object to alter its behavior when it’s internal state changes.
* [Strategy](https://springframework.guru/gang-of-four-design-patterns/strategy-pattern/): Allows one of a family of algorithms to be selected on-the-fly at runtime.
* [Template Method](https://springframework.guru/gang-of-four-design-patterns/template-method-pattern/): Defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.

The final Behavioral pattern that I will discuss in this post is the Visitor pattern – A pattern that decouples the algorithm from an object structure on which it operates.

## Visitor Pattern: Introduction

When going into enterprise application development, you will be working more and more with object structures. Such structures can range from a collection of objects, object inheritance trees, to complex structures comprising of a composite implemented using the [Composite](https://springframework.guru/gang-of-four-design-patterns/composite-pattern/) structural pattern. As you start working, you will be adding operations to the elements of such structures and distributing the operations across the other elements of the structure.

This complexity can quickly lead to a messy system that’s hard to understand, maintain, and change. Imagine, you or some other programmers later need to change the class of one such element to address some new requirements. Initially, understanding the code itself is a big challenge. Think in terms of understanding a class with over thousand lines of code. I’ve seen this type of class too many times in legacy code. One large class, with just one public method, and over one thousand lines of code.

It’s extremely time consuming to just understand what the class is trying to do. The smallest of changes need to be delicately thought out to ensure you’re not breaking things.

Let’s start with an example of a mail client configurator application. The requirements state that the application should allow users to configure and use the open source Opera and Squirell mail clients in Windows and Mac environments. Sounds simple – So let’s start coding by creating an interface containing the operations of the mail clients and the subclasses, one each for the mail clients.

This is how our interface looks like.