

**软件工程硕士课程报告**

****

报告题目 15种设计模式小作业

学生学号 17101212475

学生姓名 豆淋淋

工程领域 软件工程

提交日期 二O一八年五月

1. 创建型
   1. 工厂方法模式
      1. 模式名称

模式名称：工厂方法模式

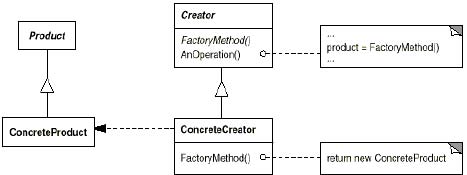
定义一个用于创建对象的接口，让子类决定实例化哪一个类，工厂方法使一个类的实例化延迟到其子类。

* + 1. 应用问题

工厂方法模式适用于一个类不能预测(预见)必须创建类的对象，类希望它的子类指定它创建的对象。工厂方法模式实现时，客户端需要决定实例化到哪一个工厂来实现一个类，选择判断问题可能还是存在，只是将其转移到了客户端去进行。将类的实例化延迟到子类中去进行。

* + 1. 解决方案

工厂方法模式的结构图：



Produce类定义工厂方法所创建的对象的接口，由ConcreteProduct类来实现，ConcreteProduct类是一个具体的产品，Creator类声明工厂方法，该方法返回一个Produce类型的对象。ConcreteCreator类重定义工厂方法以返回一个ConcreteProduct类实例。

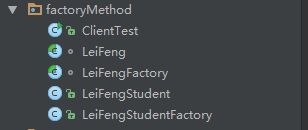
* + 1. 效果

（1）示例描述

这里举个学雷锋例子，比如有个人A要学雷锋做好事，这个人作为一个具体的产品，就相当于是结构图中的ConcreteProduct类的对象，表示一个具体的对象，而为老人服务，老人（相当于客户端）并不需要知道这个服务对象的信息，不需要知道他叫什么等，所以该具体的类ConcreteProduct需要继承一个抽象类Product，以便封装具体的类对象，并且该类中包含雷锋所具有的各种方法；但这样还不够，雷锋工厂类作为一个抽象类，相当于上图中的Creator类，这个类就像一个生产雷锋的工厂一样，返回一个学雷锋的对象，而学雷锋的大学生工厂类就是雷锋工厂类的子类，也就是上图中的ConcreteCreator类，因为学雷锋的除了大学生，可能还有其他人，所以当需求改变，我们只需要增加一个这样的类似工厂类以及一个具体的产品ConcreteProduct类就可以了。<br>

在依赖关系中，如果A依赖于B，那么一般B对象会作为A类的构造函数的返回值，指生成一个B这样的对象。

（2）目录结构



（3）代码

LeiFeng:

package designpattern.factoryMethod;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*abstract class LeiFeng {  
 public abstract void sweep();  
 public abstract void wash();  
}

LeiFengFactory:

package designpattern.factoryMethod;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*abstract class LeiFengFactory {  
 public abstract LeiFeng createLeiFeng();  
}

LeiFengStudent:

package designpattern.factoryMethod;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*public class LeiFengStudent extends LeiFeng{  
  
 @Override  
 public void sweep() {  
 System.*out*.print("LeifengStudent is sweeping");  
 }  
  
 @Override  
 public void wash() {  
 System.*out*.print("LeifengStudent is washing");  
 }  
}

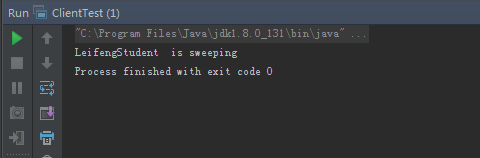
LeiFengStudentFactory:

package designpattern.factoryMethod;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*public class LeiFengStudentFactory extends LeiFengFactory{  
 @Override  
 public LeiFeng createLeiFeng() {  
 return new LeiFengStudent();  
 }  
}

ClientTest:

package designpattern.factoryMethod;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*public class ClientTest {  
 public static void main(String[] args) {  
 LeiFengFactory factory = new LeiFengStudentFactory();  
 LeiFeng me = factory.createLeiFeng();  
 me.sweep();  
 }  
}

（4）运行结果



1.2抽象工厂模式

1.2.1模式名称

模式名称：抽象工厂模式

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

1.2.2应用问题

在以下情况中使用抽象工厂模式:

（1）一个系统应该独立于它的产品是如何创建、组合和表示的。

（2）类不能预期它必须创建的对象的类

（3）一个系统必须只使用一系列产品系列中的一个。

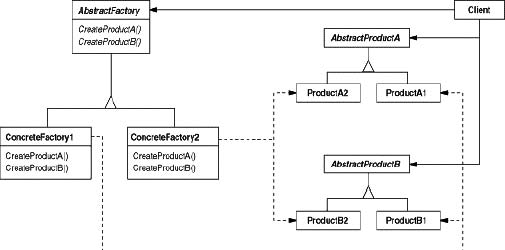
（4）一组相关的产品对象被设计成一起使用，您需要执行这个约束

（5）您希望提供一个类库的产品，而您想要显示的只是它们的接口，而不是它们的实现。

抽象工厂模式适用于要实现易于交换产品系列，由于具体的工厂类在一个应用中只需要在初始化的时候出现一次，这就使得改变一个应用的具体工厂变得很容易，它只需要改变具体工厂即可使用不同的产品配置。并且它可以使具体的创建实例过程与客户端分离，客户端是通过它们的抽象接口操纵实例，产品的具体类名也被具体工厂实现分离，不会出现在客户端。

1.2.3解决方案

抽象工厂模式的结构图如图所示：



实际上，抽象工厂模式使工厂方法模式的升级和加工，在工厂方法中，我们只有一个产品类，当有多个具体的产品类对象时，我们只需要增加继承自产品类的具体产品类以及与之对应的生产该产品的工厂类就可以了，但是如果我们需要的不仅仅是一个产品类。还是用学雷锋的例子，比如还有一个学钱学森的类，那么我们按照工厂方法的思路的话，就需要模仿原来工厂方法中的那一套中所有类再写一遍，这样做虽然也能达到效果，但是如果有N个产品类，那么显然是不高效的，所以抽象工厂应运而生。

1.2.4效果

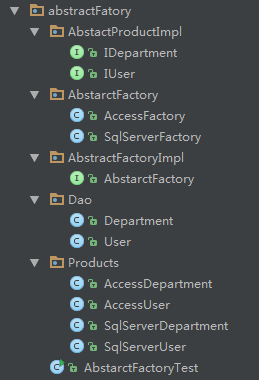
运用抽象工厂模式使得创建过程与客户端分离。

（1）示例描述

举个例子：比如大学生A，和社区志愿者B要学雷锋做好事，大学生c和社区志愿者D要学钱学森，那么A和B就作为上图中AbstarctProductA（雷锋类）的两个具体的产品ProductA1和ProduceA2，而大学生C和社区志愿者D就作为上图中AbstarctProductB(钱学森类)的两个具体的产品ProductB1和ProduceB2,至此,以上部分其实和工厂方法模式中的对应部分是一致的,而抽象工厂的意义就在于如何简化原来按照工厂方法模式的需要的多个工厂类,所以在抽象工厂方法中,有一个AbstarctFactory类来作为抽象工厂，其方法包含创建雷锋类和创建钱学森类两个方法CreateProductA和CreateProductB，这个类就像一个生产雷锋对象和钱学森对象的抽象工厂一样，返回对应的对象，而学雷锋的工厂CreateFactory1的就负责实现创建大学生类CreateProductA和社区志愿者类CreateProductB，而学钱学森的工厂CreateFactory2的就负责实现创建大学生类CreateProductA和社区志愿者类CreateProductB，这样，当还有一个类似学xx的类时，我们就只需要除了增加原先的产品部分之外，就再在抽象工厂的具体子类工厂中再增加一个工厂就可以了。

　　 这个例子虽然不太合适，但也能说明一些问题，接下来我们进行代码实现，在代码中，我们的产品相当于User和Department两个产品，而无论是对哪个产品进行操作，在不同的数据库中可能实现方式不太一样，比如这里的SQL Server和Access就是不一样的，所以针对这两个数据库下的用户表和部门表等进行增加和查找操作。

（2）示例目录结构：

（3）代码

AbstactProductImpl：

package designpattern.abstractFatory.AbstactProductImpl;  
  
import designpattern.abstractFatory.Dao.Department;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*public interface IDepartment {  
 void insert(Department department);  
 Department getDepartment(int id);  
}

package designpattern.abstractFatory.AbstactProductImpl;  
import designpattern.abstractFatory.Dao.User;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*public interface IUser {  
 void insert(User user);  
 User getUser(int id);  
}

AbstractFactory:

package designpattern.abstractFatory.AbstarctFactory;  
  
import designpattern.abstractFatory.AbstactProductImpl.IDepartment;  
import designpattern.abstractFatory.AbstactProductImpl.IUser;  
import designpattern.abstractFatory.AbstractFactoryImpl.AbstarctFactory;  
import designpattern.abstractFatory.Products.AccessDepartment;  
import designpattern.abstractFatory.Products.AccessUser;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*public class AccessFactory implements AbstarctFactory{  
 @Override  
 public IUser createUser() {  
 return new AccessUser();  
 }  
  
 @Override  
 public IDepartment createIDepartment() {  
 return new AccessDepartment();  
 }  
}

SqlServerFactory:

package designpattern.abstractFatory.AbstarctFactory;  
  
import designpattern.abstractFatory.AbstactProductImpl.IDepartment;  
import designpattern.abstractFatory.AbstactProductImpl.IUser;  
import designpattern.abstractFatory.AbstractFactoryImpl.AbstarctFactory;  
import designpattern.abstractFatory.Products.SqlServerDepartment;  
import designpattern.abstractFatory.Products.SqlServerUser;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*public class SqlServerFactory implements AbstarctFactory {  
 @Override  
 public IUser createUser() {  
 return new SqlServerUser();  
 }  
  
 @Override  
 public IDepartment createIDepartment() {  
 return new SqlServerDepartment();  
 }  
}

AbstractFactoryImpl:

package designpattern.abstractFatory.AbstractFactoryImpl;  
  
import designpattern.abstractFatory.AbstactProductImpl.IDepartment;  
import designpattern.abstractFatory.AbstactProductImpl.IUser;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*public interface AbstarctFactory {  
 IUser createUser();  
 IDepartment createIDepartment();  
}

Dao:

package designpattern.abstractFatory.Dao;  
  
/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/  
public class Department {  
 private String depName;  
 private int depid;  
  
 public Department(){}  
 public Department(String depName,int depid){  
 this.depName = depName;  
 this.depid = depid;  
 }  
  
 public void setDepName(String depName){  
 this.depName = depName;  
 }  
  
 public void setDepid(int depid){  
 this.depid = depid;  
 }  
  
 public String getDepName(){  
 return this.depName;  
 }  
  
 public int getDepid(){  
 return this.depid;  
 }  
}

package designpattern.abstractFatory.Dao;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*public class User {  
 private String name;  
 private int id;  
  
 public User() {  
 }  
 public User(String name ,int id) {  
 this.name = name;  
 this.id = id;  
 }  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
}

Products:

package designpattern.abstractFatory.Products;  
import designpattern.abstractFatory.Dao.Department;  
import designpattern.abstractFatory.AbstactProductImpl.\*;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*public class AccessDepartment implements IDepartment {  
 @Override  
 public void insert(Department department) {  
 System.*out*.println("AccessDepartment insert");  
  
 }  
  
 @Override  
 public Department getDepartment(int id) {  
 System.*out*.println("AccessDepartment getDpartment");  
 return null;  
 }  
}

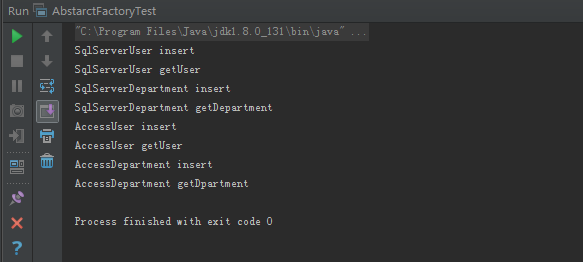
package designpattern.abstractFatory.Products;  
import designpattern.abstractFatory.Dao.User;  
import designpattern.abstractFatory.AbstactProductImpl.\*;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*public class AccessUser implements IUser {  
 @Override  
 public void insert(User user) {  
 System.*out*.println("AccessUser insert");  
   
 }  
  
 @Override  
 public User getUser(int id) {  
 System.*out*.println("AccessUser getUser");  
 return null;  
 }  
}

package designpattern.abstractFatory.Products;  
import designpattern.abstractFatory.AbstactProductImpl.IDepartment;  
import designpattern.abstractFatory.Dao.Department;  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*public class SqlServerDepartment implements IDepartment{  
  
 @Override  
 public void insert(Department department) {  
 System.*out*.println("SqlServerDepartment insert");  
 }  
  
 @Override  
 public Department getDepartment(int id) {  
 System.*out*.println("SqlServerDepartment getDepartment");  
 return null;  
 }  
}

package designpattern.abstractFatory.Products;  
import designpattern.abstractFatory.AbstactProductImpl.\*;  
import designpattern.abstractFatory.Dao.User;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*public class SqlServerUser implements IUser {  
 @Override  
 public void insert(User user) {  
 System.*out*.println("SqlServerUser insert");  
 }  
  
 @Override  
 public User getUser(int id) {  
 System.*out*.println("SqlServerUser getUser");  
 return null;  
 }  
}

ClientTest:

package designpattern.abstractFatory;  
import designpattern.abstractFatory.AbstactProductImpl.IDepartment;  
import designpattern.abstractFatory.AbstactProductImpl.IUser;  
import designpattern.abstractFatory.AbstarctFactory.AccessFactory;  
import designpattern.abstractFatory.AbstarctFactory.SqlServerFactory;  
import designpattern.abstractFatory.AbstractFactoryImpl.AbstarctFactory;  
import designpattern.abstractFatory.Dao.Department;  
import designpattern.abstractFatory.Dao.User;  
*/\*\*  
 \* Created by Administrator on 2018/5/20.  
 \*/*public class AbstarctFactoryTest {  
 public static void main(String[] args) {  
 User user = new User("zhangsan",001);  
 Department department = new Department("sale",002);  
  
 AbstarctFactory factory = new SqlServerFactory();  
  
 IUser sqlUser = factory.createUser();  
 sqlUser.insert(user);  
 sqlUser.getUser(001);  
  
 IDepartment sqlDepartment = factory.createIDepartment();  
 sqlDepartment.insert(department);  
 sqlDepartment.getDepartment(002);  
  
 AbstarctFactory factory1 = new AccessFactory();  
  
 IUser accessUser = factory1.createUser();  
 accessUser.insert(user);  
 accessUser.getUser(001);  
  
 IDepartment accessDepartment = factory1.createIDepartment();  
 accessDepartment.insert(department);  
 accessDepartment.getDepartment(002);  
  
 }  
}

（4）运行结果：

1.3单例模式

1.3.1 模式名称

模式名称：单例模式

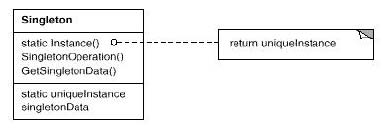
保证一个类只有一个实例或者固定个数的实例，并提供一个访问它的全局访问点。

1.3.2 应用问题

通常我们可以让全局变量使得一个对象被访问，但它不能防止你实例化多个对象。一个最好的办法就是，让类自身负责保存它的唯一实例。这个类可以保证没有其他实例可以被创建，并且它可以提供一个访问该类实例的方法。有时，我们希望在系统中只存在一个类的实例。例如，我们只需要一个窗口管理器。或者只是一个产品系列的工厂。我们需要让这个实例易于访问我们希望确保不能创建类的其他实例。

1.3.3 解决方案

单例模式的结构图：



在设计单例模式时，将构造方法为私有，这样可以保证外界类无法调用new来创建对象，在单例类中另外有一个获取实例的方法，该方法就是获得本类实例的唯一全局访问点，用来检查是否实例对象为空，如果实例不存在，则new一个实例，否则返回已经存在的实例对象。

1.3.4 效果

（1）代码：

class Singleton{  
　　private static Singleton instance;

　　private Singleton(){}  
　　public static Singleton getInstance() {  
　　if(instance == null){  
　　　　instance = new Singleton();  
　　　　}  
　　　　return instance;  
　　}  
}

1. 多线程代码

在多线程中，我们需要用加锁机制来保证单例模式线程安全，代码如下：  
　class Singleton{  
　　private static Singleton instance = null;  
　　private static Object lock = new Object();

private Singleton(){}  
　　public static Singleton getInstance() {  
　　if(instance == null){  
　　　　Synchronized(lock){  
　　　　　　if(instance == null){  
　　　　　　　　instance = new Singleton();  
　　　　　　}  
　　　　}  
　　}  
　　return instance;  
　　}  
}

1.4 建造者模式

1.4.1 模式名称

模式名称：单例模式

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

1.4.2 应用问题

建造者模式可以将产品的内部表象与产品的生成过程分割开来，从而可以使一个建造过程生成具有不同的内部表象的产品对象。如果使用了建造者模式，那么用户就只需指定需要建造的类型就可以得到它们，而具体的建造过程和细节就不需要知道了。

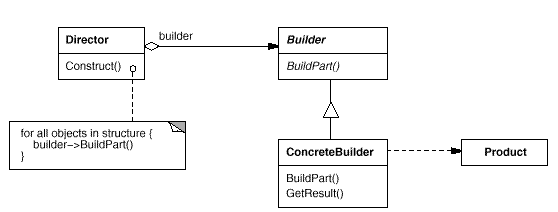
建造者模式主要用于创建一些复杂的对象，这些对象内部构建间的建造者顺序通常使稳定的，但对象内部的构建通常面临着复杂的变化。建造者模式的好处就是使得建造代码和表示代码分离，由于建造者隐藏了该产品是如何组装的，所以若需要改变一个产品的内部表示，只需要定义一个具体的建造者就可以了。也就是所谓构建与表示相分离，就是部件的生产和组装相分离。

创建复杂对象的算法应该独立于组成对象的部分以及它们是如何组装的。构造过程必须允许对构造的对象进行不同的表示。

有些程序根据显示的数据上下文动态地构造菜单。RTF(富文本格式)文档交换格式的阅读器应该能够将RTF转换为多种文本格式。读者可以将RTF文档转换为纯ASCII文本或转换为可交互编辑的文本小部件。然而,问题是,可能的转换是开放式的数量(开放式的/无限制的)。因此，在不修改阅读器的情况下添加新的转换应该很容易。

1.4.3 解决方案

建造者模式的结构图：



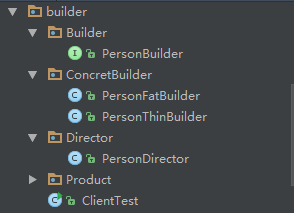
Builder类是为创建一个Product对象的各个部件指定的抽象接口。而ConcreteBuilder就是具体的部件的建造者，定义如何建造这些部件。Product就是具体的产品角色了，负责展示一个具体的产品对象，ConcreteBuilder类也就是建造者依赖于该类，没有该具体产品类，那么建造者类就没有存在的意义，而最重要的Director类就是指挥者，它是构建一个使用Builder接口的对象，根据用户的需求来构建具体的部件，也就是指挥到底建造的那些部件的特征是什么样的。在该类的指挥构建的方法中一般参数为Builder的类对象，用来指挥创建哪个部分。注意：在Builder类中除了定义各个部分之外，一般还会定义一个获得产品的一个方法，在ConcreteBuilder类实现继承自Builder的所有方法，其中获得产品的方法中返回其属性，其属性就是产品类对象。

1.4.4 效果

（1）示例描述

举个例子：现在我们需要造小人，但这些人有胖的，瘦的，高的，矮的。那么小人包含头，身体，左胳膊，右胳膊，左腿，右腿等部件，指挥者Director就是用来指定是瘦的，还是胖的，还是其他，而Builder就是抽象出都由哪些部件，并将生产这些部件的这些方法抽象，在具体的建造者ConcreteBuilder中去建造对应的指挥者指挥的类型的部件，比如高的小人，而Product就是将高的创建者创建的符合高的小人的部件拿过来进行组装，并且展示。

（2）示例目录结构



（3）代码

PersonBuilder:

import designpattern.builder.Product.PersonProduct;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public interface PersonBuilder {  
 void builderHead();  
 void builderbody();  
 void builderLeftArm();  
 void buliderRightArm();  
 void builderLeftLeg();  
 void builderRightLeg();  
 PersonProduct getPerson();  
}

PersonFatBuilder:

package designpattern.builder.ConcretBuilder;  
  
import designpattern.builder.Builder.PersonBuilder;  
import designpattern.builder.Product.PersonProduct;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public class PersonFatBuilder implements PersonBuilder {  
  
 PersonProduct person = new PersonProduct();  
  
 @Override  
 public void builderHead() {  
 person.addPart("fathead");  
 }  
  
 @Override  
 public void builderbody() {  
 person.addPart("fatbody");  
 }  
  
 @Override  
 public void builderLeftArm() {  
 person.addPart("fatleftArm");  
 }  
  
 @Override  
 public void buliderRightArm() {  
 person.addPart("fatrightArm");  
 }  
  
 @Override  
 public void builderLeftLeg() {  
 person.addPart("fatleftLeg");  
 }  
  
 @Override  
 public void builderRightLeg() {  
 person.addPart("fatRightLeg");  
 }  
  
 @Override  
 public PersonProduct getPerson(){  
 return person;  
 }  
}

PersonThinBuilder:

package designpattern.builder.ConcretBuilder;  
  
import designpattern.builder.Builder.PersonBuilder;  
import designpattern.builder.Product.PersonProduct;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public class PersonThinBuilder implements PersonBuilder{  
  
 PersonProduct person = new PersonProduct();  
 @Override  
 public void builderHead() {  
 person.addPart("thinHead");  
 }  
  
 @Override  
 public void builderbody() {  
 person.addPart("thinBody");  
 }  
  
 @Override  
 public void builderLeftArm() {  
 person.addPart("thinLeftArm");  
 }  
  
 @Override  
 public void buliderRightArm() {  
 person.addPart("thinRightArm");  
 }  
  
 @Override  
 public void builderLeftLeg() {  
 person.addPart("thinLeftLeg");  
 }  
  
 @Override  
 public void builderRightLeg() {  
 person.addPart("thinRightLeg");  
 }  
  
 @Override  
 public PersonProduct getPerson(){  
 return person;  
 }  
}

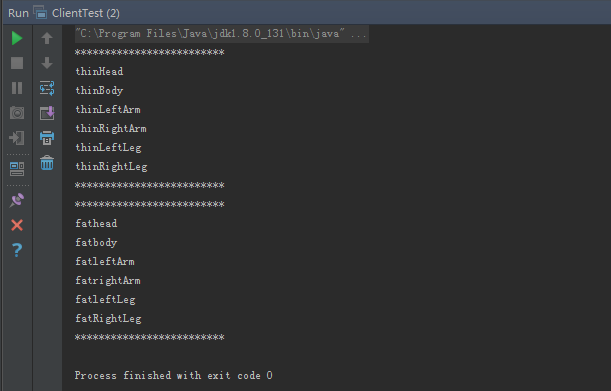
PersonDirector:

package designpattern.builder.Director;  
import designpattern.builder.Builder.PersonBuilder;  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public class PersonDirector {  
 public PersonDirector(){}  
 public void create(PersonBuilder builder){  
 builder.builderHead();  
 builder.builderbody();  
 builder.builderLeftArm();  
 builder.buliderRightArm();  
 builder.builderLeftLeg();  
 builder.builderRightLeg();  
 }  
}

ClientTest:

package designpattern.builder;  
  
import designpattern.builder.Builder.PersonBuilder;  
import designpattern.builder.ConcretBuilder.PersonFatBuilder;  
import designpattern.builder.ConcretBuilder.PersonThinBuilder;  
import designpattern.builder.Director.PersonDirector;  
import designpattern.builder.Product.PersonProduct;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public class ClientTest {  
 public static void main(String[] args) {  
 PersonDirector director = new PersonDirector();  
 PersonBuilder person = new PersonThinBuilder();  
 director.create(person);  
 PersonProduct thinperson = person.getPerson();  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 thinperson.show();  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
  
 PersonBuilder person1 = new PersonFatBuilder();  
 director.create(person1);  
 PersonProduct fatperson = person1.getPerson();  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 fatperson.show();  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 }  
}

（4）运行结果



1.5 原型模式

1.5.1 模式名称

模式名称：原型模式

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。原型模型其实就是从一个对象再创建另外一个可定制的对象，而且不需要知道任何创建的细节。

1.5.2 应用问题

设计一个独立于对象如何创建、组合和表示的系统非常有用

（1）动态对象规范(规范化)

指定所需的通用类，但将细节延迟到执行时

（2）不需要“工厂”类的类层次结构

是否可以避免构建与产品的类层次结构并行的工厂类层次结构

（3）类实例中很少有状态的不同组合

可以更方便地为每个状态安装原型并克隆它们，而不是每次都以适当的状态手工实例化类。

当构造和初始化一个对象比简单地克隆一个现有对象更复杂或更耗时时，这是很有用的。

（4）具体类对象的动态属性规范

克隆的具体产品对象将拥有原型实例状态的完整副本。

客户端还可以在创建时指定属性更改，初始化方法参数:“copy with changes capability”

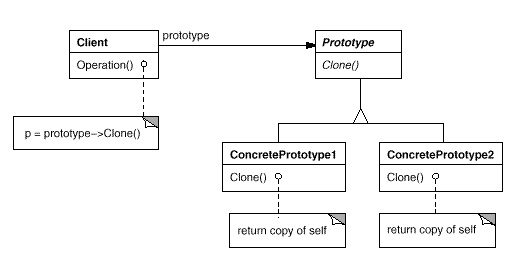
（5）允许为产品类自动扩展类继承树

自动适应(适应)具体类的子类

在clone()方法中，创建逻辑总是附加到具体的product类，不是在单独的混凝土“工厂”类

1.5.3 解决方案

原型模式的结构图：



在原型模型中，最主要的就是clone()函数，它的本意是为具体的类创建一个共同的抽象类，抽象类中包含clone()方法，而具体的类继承了这个抽象类，自然也需要实现这个抽象类中的clone（）方法了，这样就达到了创建一个可定制的对象，而不需要知道任何创建细节的目的。

　　一般在初始化的信息不发生变化的情况下，克隆是最好的办法，这既掩藏了对象创建的细节，又能对性能大大的提高。

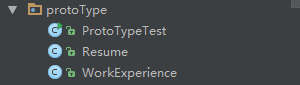
　　在java中提供了Cloneable接口，就相当于java已经为我们封装定义好了抽象方法，我们只需要定义具体的类来实现该抽象类，并且重写其中的抽象方法clone()

1.5.4 效果

（1）示例描述

在代码示例中展示了一个及其简陋的简历的原型模型的实现。在该例子中，有一个简历类，该类实现了java定义的Cloneable抽象接口，所以重写clone()方法，注意我们可以根据自己的需求来重写，这里可以进行浅拷贝和深拷贝，需要我们自己根据需求去重写，除此之后，还有一个“工作经历”的类，该类其实是简历类的一部分，也就是和简历类是组合关系，而一般在聚合或者组合关系中，一般都是将作为部分的类的对象作为整体的类的一个属性的。

（2）示例目录结构



（3）代码

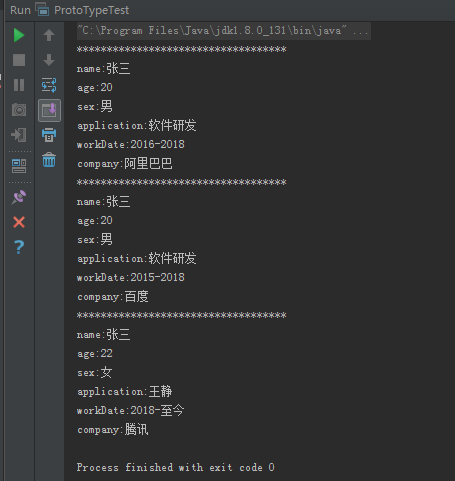
WorkExperience:

package designpattern.protoType;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public class WorkExperience implements Cloneable{  
 private String workDate;  
 private String company;  
  
 public WorkExperience(){}  
  
 public String getWorkDate() {  
 return workDate;  
 }  
  
 public void setWorkDate(String workDate) {  
 this.workDate = workDate;  
 }  
  
 public String getCompany() {  
 return company;  
 }  
  
 public void setCompany(String company) {  
 this.company = company;  
 }  
  
 public Object clone(){  
 return (Object)new WorkExperience();  
 }  
  
}

Resume:

package designpattern.protoType;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public class Resume implements Cloneable{  
 private String name;  
 private String sex;  
 private int age;  
 private String application;  
 //组合关系中，WorkExperience类是Resume的一部分，所以将工作经历类对象作为简历类的一个属性  
 private WorkExperience work;  
  
 Resume(String name){  
 this.work = new WorkExperience();  
 this.name = name;  
 }  
 Resume (WorkExperience work){  
 this.work =(WorkExperience)work.clone();  
 }  
  
 //设置个人信息  
 public void setPersonalInfo(String sex, int age, String application){  
 this.age = age;  
 this.sex = sex;  
 this.application = application;  
 }  
  
 //设置工作经历  
 public void setWorkExperience(String workDate,String company){  
 this.work.setWorkDate(workDate);  
 this.work.setCompany(company);  
 }  
  
 //展示简历  
 public void diaplay(){  
 System.*out*.println("name:"+this.name);  
 System.*out*.println("age:"+this.age);  
 System.*out*.println("sex:"+this.sex);  
 System.*out*.println("application:"+this.application);  
 System.*out*.println("workDate:"+this.work.getWorkDate());  
 System.*out*.println("company:"+this.work.getCompany());  
 }  
  
 public Object clone(){  
 //此处为深拷贝  
 Resume obj = new Resume(this.work);  
 obj.name = this.name;  
 obj.age = this.age;  
 obj.sex = this.sex;  
 obj.application = this.application;  
 return obj;  
 }  
}

（4）运行结果



1. 结构型

2.1 装饰模式

2.1.1 模式名称

模式名称：装饰模式

动态地给一个对象添加一个额外的职责，就增加功能来说，装饰模式比生成子类更加灵活。

2.1.2 应用问题

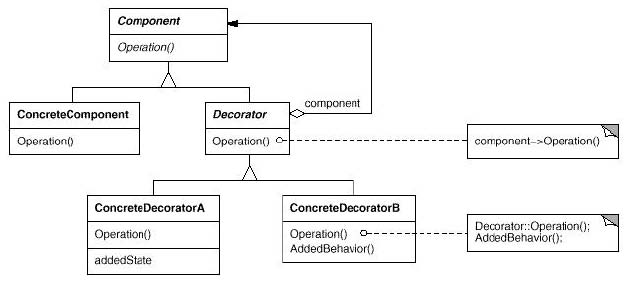
使用装饰模式，在不影响其他对象的情况下动态地向单个对象添加职责。

当通过子类扩展是不切实际的。有时，大量的独立扩展是可能的，并且会产生大量的子类来支持每个组合。或者类定义可能是隐藏的，或者子类不可用。

装饰模式是为已有的功能动态地添加更多的功能的一种方式，当系统需要新的功能时，是向已有的旧类中添加新的代码，这些新加的代码通常装饰了原有类的核心职责或者主要行为。但问题是在主类中加入了新的字段，新的方法和逻辑，从而增加了主类的复杂度，而这些新加入的东西仅仅是为了满足一些只在某种特定情况下才会执行的特殊行为的需要，而装饰模式就提供了一个很好的解决方案，它把每一个要装饰的功能放在单独的类中，并让个类包装它所需要装饰的对象，因此，当需要执行特殊的行为时，客户代码就可以在运行时根据需要有选择的，按顺序地使用装饰功能包装对象了。

2.1.3 解决方案

装饰模式的结构图：



如上图所示：对于Component类来说，是无需知道Decorator的存在的。

在Decorator类中有一个setComponent方法来对对象进行包装，这样每一个装饰对象的实现就和如何使用这个对象分离了，每个装饰对象只关心自己的功能，不需要关心如何被添加到对象链当中。如果只有一个ConcreteComponent类而没有抽象的Component类，那么Decorator类可以是ConcreteComponent的一个子类，同理，如果只有一个ConcreteDecorator类，那么就没有必要建立一个单独的Decorator类，而可以把Decorator和ConcreteDecorator的责任合并成一个类。

装饰模式的优点就是把类中的装饰功能从类中移除，这样可以简化原有的类，有效的把类的核心职责和装饰功能区分开，而且可以除去相关类中的重复的装饰逻辑。

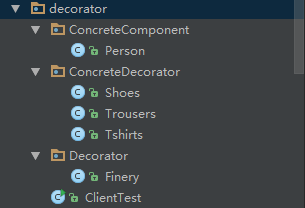
　　装饰模式和建造者模式的区别在于，建造者模式要求建造的模式时稳定的。

2.1.4 效果

（1）示例描述

举个例子：比如现在我们要为人穿衣服，也许刚开始我们可以写一个person类，之后在类中设置方法穿上衣，穿裤子，穿外套，穿鞋等，但是如果有一个“超人”呢，理论上可以改person类，但是这就违背了开闭原则，所以设计一个person类，一个服饰类，已经继承服饰类的各个具体的“衣服”。但是这样做虽然做到了分离，但是在客户端，需要创建一个person对象，再用对象调用穿具体衣服的方法，但是这样无疑“像是再=在大众面前穿衣服一样暴露”，所以我们需要装饰模式。Person类相当于是ConcreteComponent类，服饰类相当于是装饰类Decorator，具体的衣服类就是各个ConcreteDecorator类。

（2）示例目录



1. 代码

Person:

package designpattern.decorator.ConcreteComponent;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/22.  
 \*/*public class Person {  
 private String name;  
  
 public Person(){}  
 public Person(String name){  
 this.name = name;  
 }  
 public void show(){  
 System.*out*.print("decorate "+this.name+" : ");  
 }  
  
}

Shoes:

package designpattern.decorator.ConcreteDecorator;  
import designpattern.decorator.Decorator.Finery;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/22.  
 \*/*public class Shoes extends Finery{  
  
 public void show(){  
 super.show();  
 System.*out*.print(" shoes ");  
  
 }  
}

Trousers:

package designpattern.decorator.ConcreteDecorator;  
import designpattern.decorator.Decorator.Finery;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/22.  
 \*/*public class Trousers extends Finery{  
 public void show(){  
 super.show();  
 System.*out*.print(" Trousers ");  
 }  
}

Tshirts:

package designpattern.decorator.ConcreteDecorator;  
import designpattern.decorator.Decorator.Finery;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/22.  
 \*/*public class Tshirts extends Finery{  
 public void show(){  
 super.show();  
 System.*out*.print(" Tshirts ");  
 }  
}

Finery:

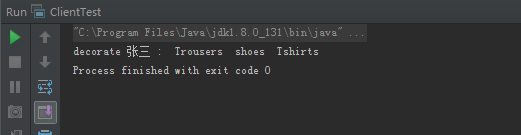
package designpattern.decorator.Decorator;  
import designpattern.decorator.ConcreteComponent.Person;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/22.  
 \*/*public class Finery extends Person{  
 protected Person component;  
  
 //打扮  
 public void decorate(Person component){  
 this.component = component;  
 }  
 public void show(){  
 if(component !=null){  
 component.show();  
 }  
 }  
}

ClientTest:

package designpattern.decorator;  
  
import designpattern.decorator.ConcreteComponent.Person;  
import designpattern.decorator.ConcreteDecorator.Shoes;  
import designpattern.decorator.ConcreteDecorator.Trousers;  
import designpattern.decorator.ConcreteDecorator.Tshirts;

*/\*\*  
 \* Created by Administrator on 2018/5/22.  
 \*/*public class ClientTest {  
 public static void main(String[] args) {  
 Person person = new Person("张三");  
 Tshirts tshirts = new Tshirts();  
 Shoes shoes = new Shoes();  
 Trousers trousers = new Trousers();  
  
 trousers.decorate(person);  
 shoes.decorate(trousers);  
 tshirts.decorate(shoes);  
 tshirts.show();  
 }  
}

（4）运行结果



2.2 代理模式

2.2.1 模式名称

模式名称：代理模式

为其他对象提供一种代理以控制这个对象的访问。

2.2.2 应用问题

如果需要一个更复杂的(复杂的)参考，代理可以提供一个简单的指针或引用对象来进行简单的参考。

（1）.远程代理：也就是为一个对象在不同的地址空间提供局部代表。这样可以隐藏一个对象存在于不同地址空间的事实。

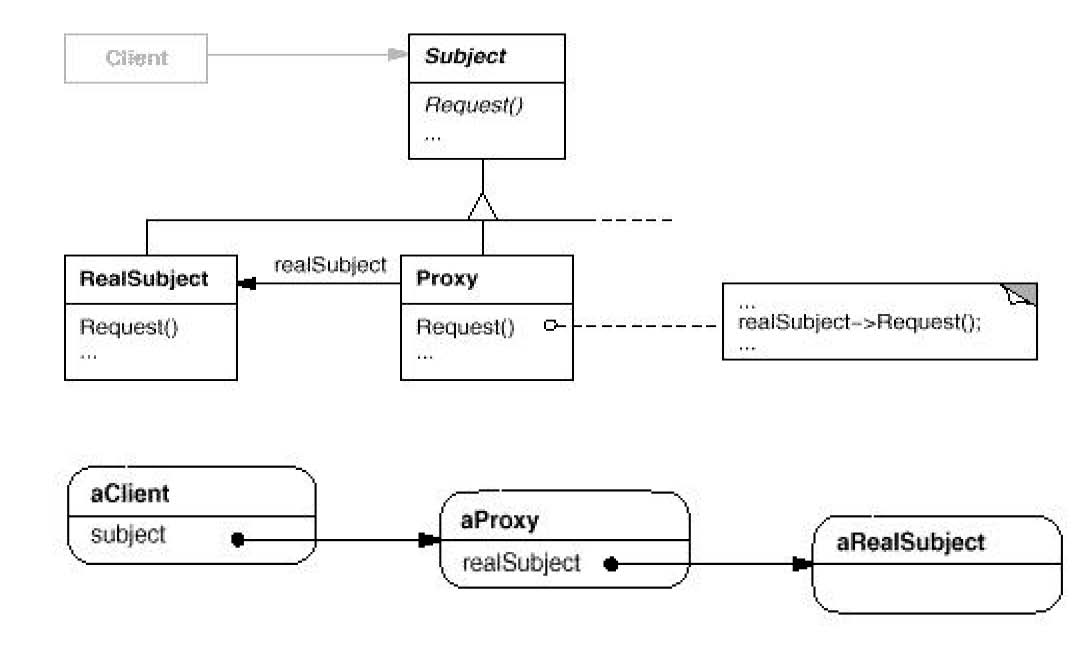
　　（2）虚拟代理：是根据需要创建开销很大的对象，通过它来存放实例化需要很长时间的真实对象。

　　（3）安全代理：用来控制真实对象访问时的权限。

　　（4）安全代理：是指当调用真实的对象时，代理处理另外一些事。

2.2.3 解决方案

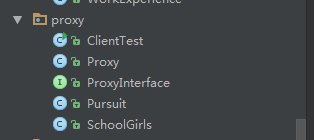
代理模式的结构图：



2.2.4 效果

（1）示例描述

举个例子：比如A代替B去追求C，给C送花之类的，那么A相当于就是中间的代理类了，而B就变成了RealSubject类。抽象出一个接口，用于对具体的代理类和被代理类的抽象。

1. 示例目录结构
2. 代码

ProxyInterface:

package designpattern.proxy;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/24.  
 \*/*public interface ProxyInterface {  
 //送花  
 void sendFlowers();  
 //送巧克力  
 void sendChocolate();  
  
}

Proxy:

package designpattern.proxy;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/24.  
 \*/*public class Proxy implements ProxyInterface {  
 Pursuit gg;  
 Proxy(SchoolGirls mm){  
 gg = new Pursuit(mm);  
 }  
 @Override  
 public void sendFlowers() {  
 gg.sendFlowers();  
 }  
  
 @Override  
 public void sendChocolate() {  
 gg.sendChocolate();  
 }  
}

Pursuit:

package designpattern.proxy;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/24.  
 \*/*//追求者类  
public class Pursuit implements ProxyInterface{  
  
 SchoolGirls mm;  
  
 Pursuit(){}  
 Pursuit(SchoolGirls mm){  
 this.mm = mm;  
 }  
  
 @Override  
 public void sendFlowers() {  
 System.*out*.println("send flowers to "+this.mm.getName());  
 }  
  
 @Override  
 public void sendChocolate() {  
 System.*out*.println("send chocolate to "+this.mm.getName());  
 }  
}

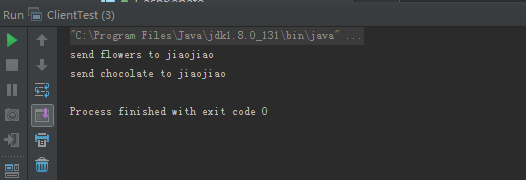
SchoolGirls:

package designpattern.proxy;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/24.  
 \*/*public class SchoolGirls {  
 private String name;  
  
 SchoolGirls(){}  
 SchoolGirls(String name){  
 this.name = name;  
 }  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
}

ClientTest:

package designpattern.proxy;  
*/\*\*  
 \* Created by Administrator on 2018/5/24.  
 \*/*public class ClientTest {  
  
 public static void main(String[] args) {  
 SchoolGirls jiaojiao = new SchoolGirls("jiaojiao");  
 Proxy daiwei = new Proxy(jiaojiao);  
 daiwei.sendFlowers();  
 daiwei.sendChocolate();  
 }  
}

（4）运行结果



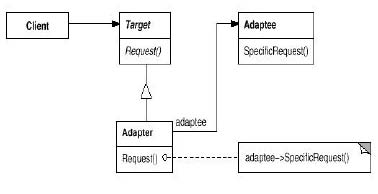
1.3适配器模式

1.3.1 模式名称

模式名称：适配器模式

将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

1.3.2 应用问题

 使用适配器模式时：

（1）希望使用一个现有的类，它的接口与您需要的不匹配

（2）希望创建一个可重用类，该类与具有不兼容接口的不相关类协作

适配器模式的使用场景：

想使用一个已经存在的类，但如果它的接口，也就是它的方法和你的要求不相同时，就用适配器模式，也就是在双方都不太容易修改的时候再使用适配器模式适配，而不是一有不同时就使用它。

1.3.3 解决方案

适配器模式的结构图：

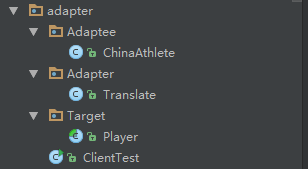
如上图所示：其实就是在Adapter类中包含一个需要适配的类Adaptee的对象，在实现了目标接口的Adapter类中的抽象方法的request()中用需要适配的Adaptee的对象调用Adaptee类对象本身的特殊需要被适配的方法。这样就相当于表面上看适配已经完成了。

1.3.4 效果

（1）示例描述

举个例子：就像姚明作为中国的运动员，他如果要去美国打球，那么他不会英语，就无法进行比赛或者其他工作，这个时候，就可以为他提供一个翻译人员，这里的翻译人员就相当于适配器Adapter。而运动员就是Target抽象类，里面有进攻和防守等方法，而一个中国队员就相当于是Adaptee类。

（2）示例目录结构



1. 代码

ChinaAthlete:

package designpattern.adapter.Adaptee;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public class ChinaAthlete {  
  
 private String name;  
 public ChinaAthlete(){}  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public void specialAttack(){  
 System.*out*.println(this.getName()+" special attack");  
 }  
  
 public void specialDefense(){  
 System.*out*.println(this.getName()+" special defense");  
 }  
}

Translate:

package designpattern.adapter.Adapter;  
  
import designpattern.adapter.Adaptee.ChinaAthlete;  
import designpattern.adapter.Target.Player;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public class Translate extends Player{  
  
 ChinaAthlete player = new ChinaAthlete();  
  
 //父类必须包含默认的无惨构造函数  
 public Translate(String name){  
 player.setName(name);  
 }  
  
 @Override  
 public void attak() {  
 //达到适配的效果  
 player.specialAttack();  
 }  
  
 @Override  
 public void defense() {  
 //达到适配的效果  
 player.specialDefense();  
 }  
}

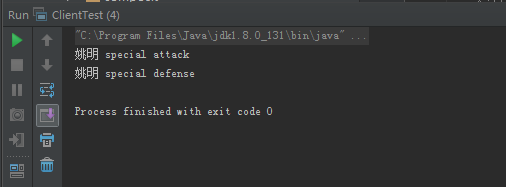
Player:

package designpattern.adapter.Target;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public abstract class Player {  
  
 private String name;  
  
 public Player(){}  
  
 public Player(String name){  
 this.name = name;  
 }  
  
 public abstract void attak();  
 public abstract void defense();  
}

ClientTest:

package designpattern.adapter;  
  
import designpattern.adapter.Adapter.Translate;  
import designpattern.adapter.Target.Player;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public class ClientTest {  
 public static void main(String[] args) {  
 Player player = new Translate("姚明");  
 player.attak();  
 player.defense();  
  
 }  
}

（4）运行结果



1.4 桥接模式

1.4.1 模式名称

模式名称：桥接模式

将抽象部分与它的实现部分分离，使它们可以独立地变化。

1.4.2 应用问题

桥接模式使用在：

（1）希望避免抽象及其实现之间的永久绑定。

（2）抽象及其实现都应该通过子类化来扩展。

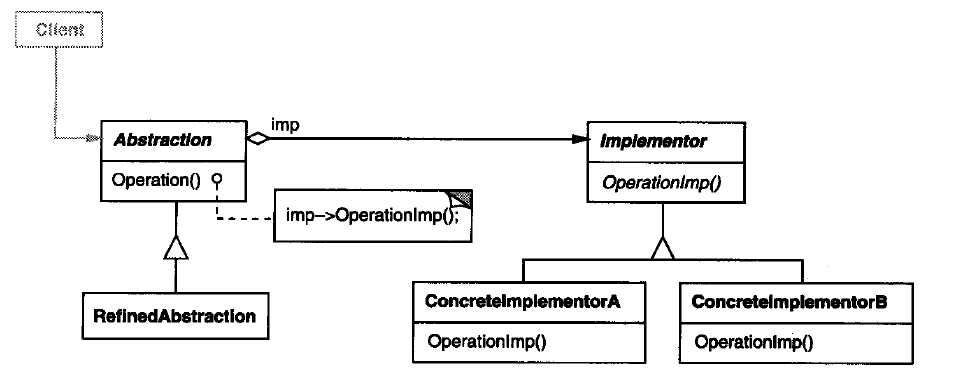
（3）抽象实现中的更改应该不会对客户端产生影响;也就是说，它们的代码不应该被重新编译。

（4）希望完全对客户端隐藏抽象的实现。

（5）希望在多个对象之间共享一个实现(可能使用引用计数)，而这个事实应该隐藏在客户端。

1.4.3 解决方案

桥接模式结构图：



如图所示：Implementor是一个接口，代表按照一种分类类型，而与之对应的Abstarction也是一个接口。代表另一种分类类型，这两种类型之间是聚合关系，ConcreteImplementorA类和ConcreteImplementorB类是Implementor类型下的具体分类，实现了Implementor类，同理，RefinedAbstarction也是Abstarction类的具体实现类。

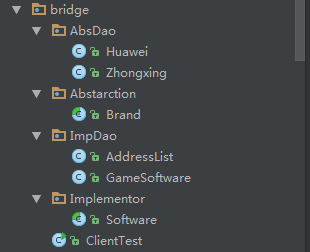
抽象与实现分离的意思并不是说让抽象类和派生类分离，因为这没有意义，实现指的是抽象类和它的派生类用来实现自己的对象，就即将要举的例子来说，就是让“手机”既可以按照品牌来分类，也可以按照功能来分类。也就是说\*\*实现系统可能有多角度分类，每一种分类都有可能发生变化，那么就把这种多角度分离出来让它们独立变化，减少它们之间的耦合。而不是只用继承来造成大量类的增加。

1.4.4 效果

（1）示例描述

举个例子来说：比如手机，手机有各种品牌(相当于Abstarction类)，手机上也有各种手机软件(相当于Implementor)，品牌可能包含华为，中兴(相当于RefinedAbstarction类)，手机软件可能有通讯录(相当于ConcreteImplementorA)和游戏(相当于ConcreteImplementorB)等。

（2）示例目录结构



（3）代码

Software:

package designpattern.bridge.Implementor;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public abstract class Software {  
 public abstract void run();  
}

Brand:

package designpattern.bridge.Abstarction;  
import designpattern.bridge.Implementor.Software;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public abstract class Brand {  
  
 //设置为子类访问权限protected，子类可以直接访问，调用对应的run()  
 protected Software software;  
  
 public Software getSoftware() {  
 return software;  
 }  
 //设置手机软件  
 public void setSoftware(Software software) {  
 this.software = software;  
 }  
 //运行  
 public abstract void run();  
}

AddressList:

package designpattern.bridge.ImpDao;  
  
import designpattern.bridge.Implementor.Software;  
  
public class AddressList extends Software{  
  
 @Override  
 public void run() {  
 System.*out*.println("run address list");  
 }  
}

GameSoftware:

package designpattern.bridge.ImpDao;  
  
import designpattern.bridge.Implementor.Software;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public class GameSoftware extends Software {  
  
 @Override  
 public void run() {  
 System.*out*.println("run gameSoftware");  
 }  
}

Huawei:

package designpattern.bridge.AbsDao;  
  
import designpattern.bridge.Abstarction.Brand;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public class Huawei extends Brand{  
  
 @Override  
 public void run() {  
 System.*out*.print("Huawei ");  
 software.run();  
 }  
}

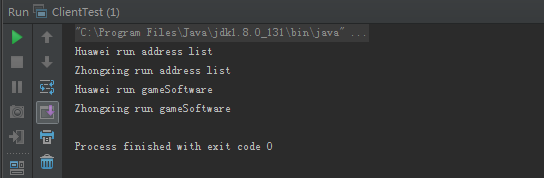
Zhongxing:

package designpattern.bridge.AbsDao;  
  
import designpattern.bridge.Abstarction.Brand;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public class Zhongxing extends Brand{  
 @Override  
 public void run() {  
 System.*out*.print("Zhongxing ");  
 software.run();  
 }  
}

ClientTest:

package designpattern.bridge;  
  
import designpattern.bridge.AbsDao.Huawei;  
import designpattern.bridge.AbsDao.Zhongxing;  
import designpattern.bridge.Abstarction.Brand;  
import designpattern.bridge.ImpDao.AddressList;  
import designpattern.bridge.ImpDao.GameSoftware;  
import designpattern.bridge.Implementor.Software;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/21.  
 \*/*public class ClientTest {  
 public static void main(String[] args) {  
  
 Software software1 = new AddressList();  
  
 Brand phone1 = new Huawei();  
 phone1.setSoftware(software1);  
 phone1.run();  
 Brand phone2 = new Zhongxing();  
 phone2.setSoftware(software1);  
 phone2.run();  
  
 Software software2 = new GameSoftware();  
 phone1.setSoftware(software2);  
 phone1.run();  
 phone2.setSoftware(software2);  
 phone2.run();  
 }  
}

（4）运行结果



2.5 组合模式

2.5.1 模式名称

模式名称：组合模式

将对象组合成树形结构以表示“部分-整体”的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。实际就是部分与整体能够被一致对待的问题。

2.5.2 应用问题

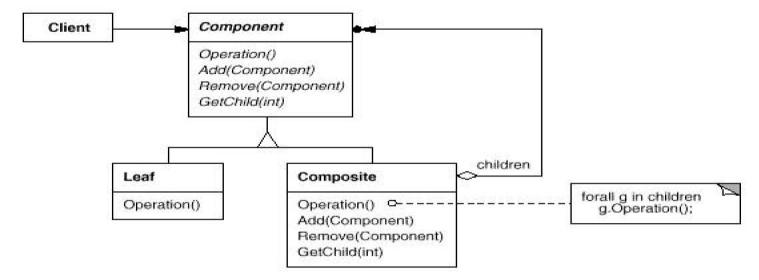
使用组合模式时：

（1）希望表示对象的部分-整个层次结构

（2）希望客户端能够忽略对象的组合和单个对象之间的差异。客户端将统一处理复合结构中的所有对象。

组合模式的好处在于其定义了基础对象和组合对象的类层次结构，基本对象可以被组合成更加复杂的组合对象，而这个组合对象又可以被组合，这样不断地递归下去，客户代码中，任何用到基本对象的地方都可以使用组合对象了。用户不用关心到底是叶节点还是枝节点。组合模式让客户可以一致使用组合结构和单个对象。

2.5.3 解决方案

 组合模式的结构图：

如上图所示：Component是组合对象声明的接口，在适当情况下，实现所有类共有接口的默认行为，声明一个接口用于访问和管理Component的子部件。Composite定义存储叶节点的行为，用来存储子部件，在Component接口中实现与子部件有关的操作。既然是叶节点的组合，那么在该类中，属性就是一个存有多个Component的list，并且该类重写了Component定义的方法。

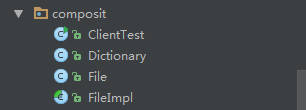
当树叶不可以再长分支的时候，这种方式叫做透明方式，也就是说在Component中声明所有的用来管理子对象的方法，这样树叶子类和枝节点对于外界没有区别，它们具有完全一致的接口，但有一个问题就是其实叶子节点有些增加或者移除的操作是不能进行的，所以实现它没有意义。如果不想让叶节点类去做这些无用功，那么就是安全模式，也就是Component接口中不去声明增加和移除方法，那么叶子类不用实现其方法，而在枝节点类中声明管理子类对象的方法，这样就不会使得树节点有其不具备的操作，但同时树节点和枝节点也没有了相同的接口，这时，客户端就需要在调用时做判断，带来了不便。

2.5.4 效果

（1）示例描述

举个例子：比如文件夹里面可以存放文件，也可以存放文件夹，有增删改查的操作。

（2）示例目录结构



（3）代码

FileImpl:

package designpattern.composit;  
import java.util.Iterator;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*public abstract class FileImpl {  
 String filename;  
  
 public FileImpl(){}  
  
 public FileImpl(String name){  
 this.filename = name;  
 }  
 public abstract void insert(FileImpl file);  
 public abstract void delete(FileImpl file);  
 public abstract void show();  
 public void setFilename(String name){  
 this.filename = name;  
  
 }  
  
 public String getFilename(){  
 return this.filename;  
 }  
  
}

File:

package designpattern.composit;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*//透明方式  
public class File extends FileImpl{  
  
 public File(String name){  
 super(name);  
 }  
  
 @Override  
 public void insert(FileImpl file) {  
 System.*out*.println("file insert");  
 }  
  
 @Override  
 public void delete(FileImpl file) {  
 System.*out*.println("file delete");  
 }  
  
 @Override  
 public void show() {  
 System.*out*.println("this file name is "+ filename);  
 }  
}

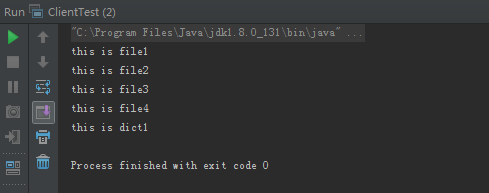
Dictionary:

package designpattern.composit;  
import java.util.ArrayList;  
import java.util.Iterator;  
import java.util.List;  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*public class Dictionary extends FileImpl {  
 List<FileImpl> files = new ArrayList<>(10);  
 public Dictionary(String name){  
 super(name);  
 }  
 @Override  
 public void insert(FileImpl file) {  
 files.add(file);  
 }  
  
 @Override  
 public void delete(FileImpl file) {  
 files.remove(file);  
 }  
  
 @Override  
 public void show() {  
 for(int i = 0;i<files.size();++i){  
 System.*out*.println("this is "+files.get(i).getFilename());  
 }  
 }  
}

ClientTest:

package designpattern.composit;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*public class ClientTest {  
 public static void main(String[] args) {  
 Dictionary root = new Dictionary("root");  
 File myfile1 = new File("file1");  
 root.insert(myfile1);  
 File myfile2 = new File("file2");  
 root.insert(myfile2);  
 File myfile3 = new File("file3");  
 root.insert(myfile3);  
 File myfile4 = new File("file4");  
 root.insert(myfile4);  
 Dictionary dict1 = new Dictionary("dict1");  
 root.insert(dict1);  
 root.show();  
 }  
  
}

（4）运行结果



三．行为型

3.1 职责链模式

3.1.1 模式名称

模式名称：职责链模式

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系，将这个对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。这里发出这个请求的客户端并不知道这当中的哪一个对象最终处理这个请求，这样系统的更改可以在不影响客户端的情况下动态地重新组织和分配责任。

3.1.2 应用问题

使用职责链：:

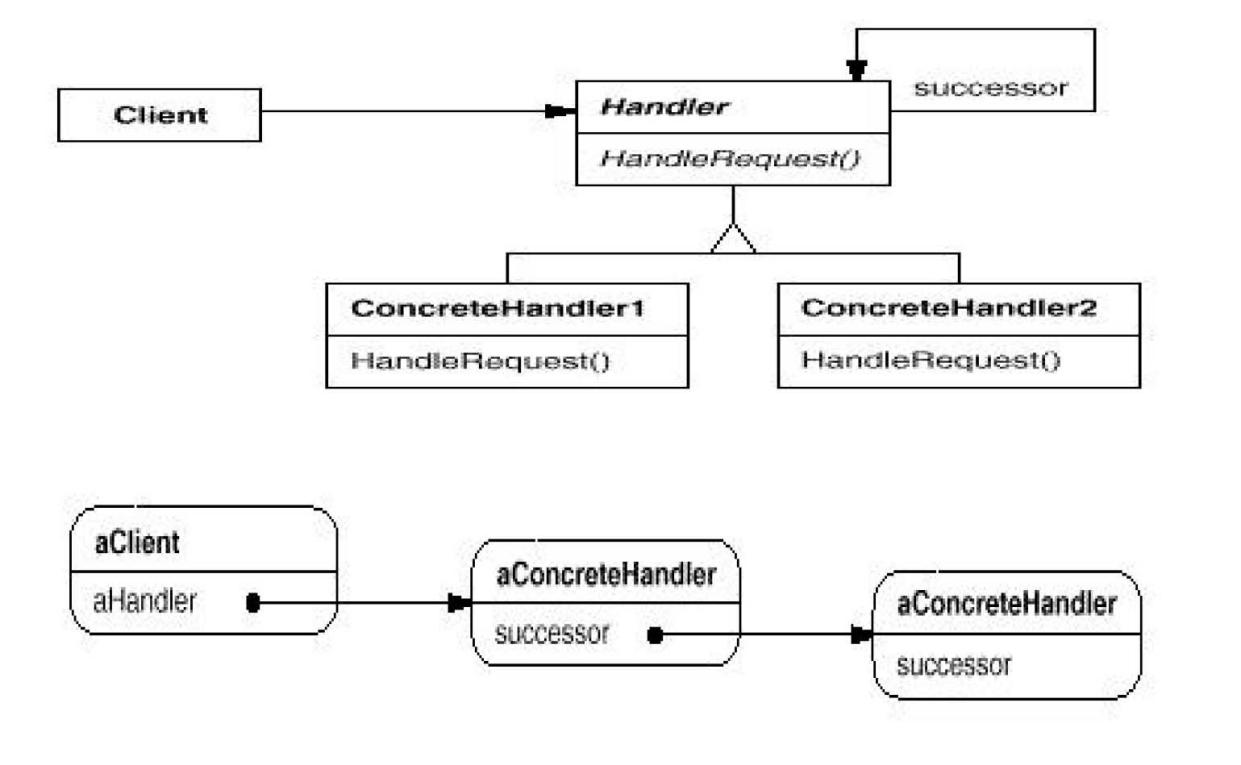
（1）当多个对象可以处理请求和实际处理程序不知道提前(预先,提前)。

（2）当请求遵循“句柄或转发”模型时——也就是说，一些请求可以在生成它们的地方处理，而另一些请求则必须转发到要处理的另一个对象。

职责链的好处:当客户提交一个请求时，请求是沿着链传递直至有一个ConcreteHandler对象负责处理它。这样做的好处就是请求者不用管哪个对象来处理，反正该请求会被处理就对了，这就使得接收者和发送者都没有对象的明确的信息，且链中的对象自己也并不知道链的结构，结果是职责链可简化对象的相互连接，它们仅需要保持一个指向其后继者的引用，而不需要保持它所有的候选接收者的引用。降低了耦合度。并且可以随时增加或者修改处理一个请求的结构，增强了给对象指派职责的灵活性。但是一个请求极有可能到了链的末端都得不到处理，或者因为没有正确配置而得不到处理。所以必须考虑全面。

3.1.3 解决方案

职责链模式的结构图：



如下图所示：Handler类作为一个处理请示接口的抽象类，ConcreteHandler类就是具体处理的类，处理它所处理的请求，可访问它的后继者，如果可以处理该请求，就处理之，否则就将该请求转发给它的后继者。

3.1.4 效果

（1）示例描述

举个例子：我要请假，先去找部门经理，经理说他没有权利，可以问问总监，总监说他没权利，可以问问总经理，总经理处理完请假之后，我就可以走人了。所以在这个例子中，管理者可以作为Handler类（抽象类），而申请请求可以作为抽象方法。

（2）示例目录结构



（3）代码

Manager:

package designpattern.chainOfResponsibility;  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*public abstract class Manager {  
 String name;  
 Manager superior;  
 public Manager(){}  
  
 public Manager(String name){  
 this.name = name;  
 }  
 public void setSuperior(Manager superior){  
 this.superior = superior;  
 }  
 public abstract void request(Request request);  
  
}

CommonManager:

package designpattern.chainOfResponsibility;  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*public class CommonManager extends Manager {  
  
 public CommonManager(String name){  
 super(name);  
 }  
 @Override  
 public void request(Request request) {  
 if(request.requestType.equals("rest")){  
 System.*out*.println("the type:"+request.requestType+" the count: "+request.count+  
 " the content : "+request.requestContent+" request is allowed");  
 }else{  
 if(this.superior != null){  
 this.superior.request(request);  
 }  
 }  
 }  
}

GeneralManager:

package designpattern.chainOfResponsibility;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*public class GeneralManager extends Manager {  
  
 public GeneralManager(String name){  
 super(name);  
 }  
 @Override  
 public void request(Request request) {  
 if(request.requestType.equals("resign")){  
 System.*out*.println("the type:"+request.requestType+" count: "+request.count+  
 " the content : "+request.requestContent+" request is allowed");  
 }else{  
 if(this.superior != null){  
 this.superior.request(request);  
 }  
 else{  
 System.*out*.println("no one can handler");  
 return;  
 }  
 }  
 }  
}

Major:

package designpattern.chainOfResponsibility;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*public class Major extends Manager {  
  
 public Major(String name){  
 super(name);  
 }  
  
 @Override  
 public void request(Request request) {  
 if(request.requestType.equals("workovertime")){  
 System.*out*.println("the type:"+request.requestType+" count: "+request.count+  
 " the content : "+request.requestContent+" request is allowed");  
 }else{  
 if(this.superior!=null){  
 this.superior.request(request);  
 }  
 }  
 }  
}

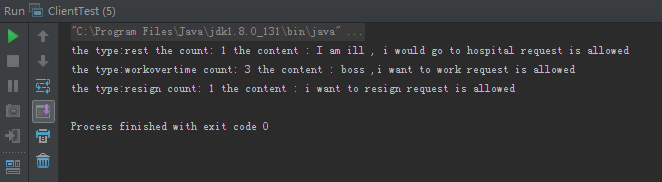
Request:

package designpattern.chainOfResponsibility;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*public class Request {  
 String requestType;  
 String requestContent;  
 int count;  
  
 public Request(){  
 this.requestType = null;  
 this.requestContent = null;  
 this.count = 0;  
 }  
 public Request(String requestType,String requestContent,int count){  
 this.requestType = requestType;  
 this.requestContent = requestContent;  
 this.count = count;  
  
 }  
 public String getRequestType() {  
 return requestType;  
 }  
  
 public void setRequestType(String requestType) {  
 this.requestType = requestType;  
 }  
  
 public String getRequestContent() {  
 return requestContent;  
 }  
  
 public void setRequestContent(String requestContent) {  
 this.requestContent = requestContent;  
 }  
  
 public int getCount() {  
 return count;  
 }  
  
 public void setCount(int count) {  
 this.count = count;  
 }  
}

ClientTest:

package designpattern.chainOfResponsibility;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*public class ClientTest {  
 public static void main(String[] args) {  
 CommonManager commonManager1 = new CommonManager("xiao bai");  
 GeneralManager generalManager1 = new GeneralManager("fang fang");  
 Major major1 = new Major("xiao qiang");  
 Request myrequest = new Request("rest","I am ill , i would go to hospital",1);  
 commonManager1.setSuperior(major1);  
 major1.setSuperior(generalManager1);  
 commonManager1.request(myrequest);  
  
 Request request1 = new Request("workovertime","boss ,i want to work",3);  
 commonManager1.request(request1);  
  
 Request request2 = new Request("resign","i want to resign",1);  
 commonManager1.request(request2);  
 }  
}

（4）效果



3.2 迭代器模式

3.2.1 模式名称

模式名称：迭代器模式

提供一种方法顺序访问一个聚合对象中各个元素，而又不暴露该对象的内部表示。

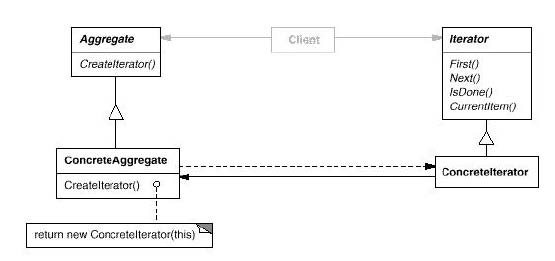
3.2.2 应用问题

使用迭代器模式：

1. 当要访问一个聚集对象，而且不管这些对象是什么都需要遍历的时候，就应该考虑使用迭代器模式。
2. 需要对聚集有多种方式遍历时，可以使用迭代器模式。
3. 支持聚合对象的遍历，而不暴露它们的内部表示
4. 支持聚合对象的多个并发遍历
5. 为遍历不同聚合结构(即支持多态迭代)提供统一的接口

3.2.3 解决方案

迭代器模式的结构图：

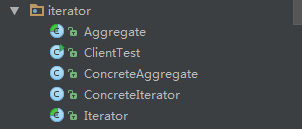
 Iterator类是一个抽象类，用于定义得到开始对象，得到下一个对象，判断是否到尾部，当前对象等抽象方法，统一接口。Aggregate聚集抽象类，用于创建迭代器，ConcreteIterator是具体的迭代器类，继承Iterator,该类属性中有一个具体聚集对象ConcreteAggregate类对象。ConcreteAggregate是具体聚集类，继承Aggregate，该类中有一个属性为list的聚集对象。

3.2.4 效果

（1）示例描述

举个例子：售票员不会管上来的是人还是物，是中国人还是外国人，是员工还是外来人员等，只要是乘客，就必须买票。

（2）示例目录结构



（3）代码

Aggregate:

package designpattern.iterator;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*public abstract class Aggregate {  
 public abstract Iterator createIterator();  
}

ConcreteAggregate:

package designpattern.iterator;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Objects;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*public class ConcreteAggregate extends Aggregate {  
  
 private List<Object> items = new ArrayList<Object>();  
 @Override  
 public Iterator createIterator() {  
 return new ConcreteIterator(this);  
 }  
  
 public List<Object> getItems() {  
 return items;  
 }  
 public int getCount(){  
 return items.size();  
 }  
 public Object getItem(int index){  
 return items.get(index);  
 }  
 public void setItem(int index,Object value){  
 items.add(index,value);  
 }  
}

Iterator:

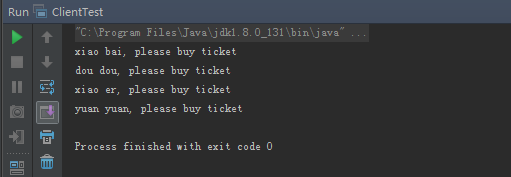
package designpattern.iterator;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*public abstract class Iterator {  
 public Iterator(){}  
 public abstract Object first();  
 public abstract Object next();  
 public abstract boolean isDone();  
 public abstract Object currentItem();  
}

ConcreteIterator:

package designpattern.iterator;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*public class ConcreteIterator extends Iterator {  
 private ConcreteAggregate aggregate;  
 private int current = 0;  
  
 public ConcreteIterator(ConcreteAggregate aggregate){  
 this.aggregate = aggregate;  
 }  
 @Override  
 public Object first() {  
 return aggregate.getItem(0);  
 }  
  
 @Override  
 public Object next() {  
 Object ret = null;  
 current++;  
 if(current<aggregate.getCount()){  
 ret = aggregate.getItem(current);  
 }  
 return ret;  
 }  
  
 @Override  
 public boolean isDone() {  
 return current>= aggregate.getCount()?true:false;  
 }  
  
 @Override  
 public Object currentItem() {  
 return aggregate.getItem(current);  
 }  
}

ClientTest:

package designpattern.iterator;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/28.  
 \*/*public class ClientTest {  
 public static void main(String[] args) {  
 ConcreteAggregate aggregate1 = new ConcreteAggregate();  
 aggregate1.setItem(0,"xiao bai");  
 aggregate1.setItem(1,"dou dou");  
 aggregate1.setItem(2,"xiao er");  
 aggregate1.setItem(3,"yuan yuan");  
  
 Iterator it = new ConcreteIterator(aggregate1);  
 Object item = it.first();  
 while( it.isDone()== false){  
 System.*out*.println(it.currentItem()+", please buy ticket");  
 it.next();  
 }  
 }  
 }

（4）运行结果

3.3 观察者模式

3.3.1 模式名称

模式名称：观察者模式

定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象，这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

3.3.2 应用问题

在下列任何一种情况下使用观察者模式:

（1）当一个抽象有两个方面，一个依赖另一个。

（2）将这些方面封装在单独的对象中，可以使您独立地更改和重用它们。

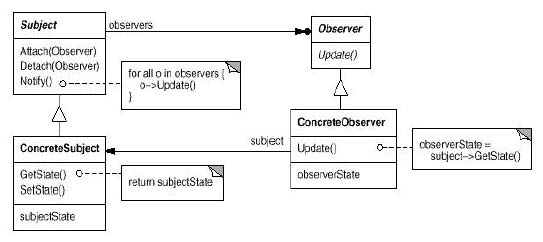
（3）当对一个对象的更改需要更改其他对象时

（4）当一个对象应该能够通知其他对象而不需要对这些对象做出假设时

观察者模式的使用场景：当一个对象的改变需要同时改变其他对象的时候。而且它不知道具体有多少对象待改变时，就可以考虑使用观察者模式。当一个抽象模型有两个方面，其中一方面依赖另一个方面，这时用观察者模式可以将两者封装在独立的对象中使它们各自独立地改变和复用。观察者模式所做的工作就是解除耦合，让耦合的双方都依赖于抽象，而不是依赖于具体，从而使得各自的变化都不会影响另一边的变化。

3.3.3 解决方案

观察者模式的结构图：



如上图所示：在观察者模式中，Observer类即是抽象观察者，为所有的具体观察者定义一个接口，在得到主题的通知时更新自己。这个接口叫做更新接口，抽象观察者一般用抽象类或者接口实现，更新接口一般包含一个update()方法，这个方法叫做更新方法。ConcreteObserve类是具体观察者，实现抽象观察者角色所要求的更新接口，以便使得本身的状态与主题状态相协调。而Subject类，它把所有对观察者对象的引用保存在一个聚集里，每个主题都可以有任何数量的观察者，抽象主题提供一个接口，可以增加和删除观察者对象。ConcreteSubject类是具体的主题，将有关的状态存入具体观察者对象，在具体主题的内部状态改变时，给所有登记过的观察者发出通知。

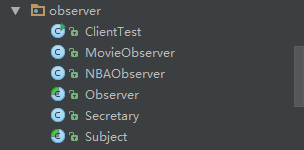
观察者模式的特点：如果一个系统分割成一系列相互协作的类一个不好的地方在于需要维护相关对象间的一致性。我们不希望为了维护一致性而使得各类紧密耦合，这样会给维护更新带来不便。但是观察者模式中的主题Subject类和观察者Observer类，一个主题类可以有任意数目的依赖它的Observer，一旦Subject的状态发生改变，那么所有的Observer都可以得到通知。主题对象发出通知时并不知道谁是它的观察者，并且任何一个具体观察者也不需要知道其他观察者的存在。

3.3.4 效果

（1）示例描述

举个例子：在公司，作为一个员工，想在上班时间做一些无关工作的事情，但又不想让老板发现，那么可以让前台秘书在老板回来的时候通知我，这样在老板回来后，我就从娱乐模式切换回了工作模式。而所做的具体娱乐活动的人员我就是具体的观察者。前台秘书就相当于具体主题(通知者)类。

（2）示例目录结构

（3）代码

Subject:

package designpattern.observer;  
import java.util.ArrayList;  
import java.util.List;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/29.  
 \*/*public abstract class Subject {  
 public abstract void attach(Observer observer);  
 public abstract void detach(Observer observer);  
 public abstract void notifys();  
 public abstract String getSubState();  
 public abstract void setSubState(String state);  
}

Secretary:

package designpattern.observer;  
import java.util.ArrayList;  
import java.util.List;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/29.  
 \*/*public class Secretary extends Subject {  
 List<Observer> objList = new ArrayList<Observer>();  
 String action;  
 @Override  
 public void attach(Observer observer) {  
 objList.add(observer);  
 }  
  
 @Override  
 public void detach(Observer observer) {  
 objList.remove(observer);  
 }  
  
 @Override  
 public void notifys() {  
 for (Observer obj: objList) {  
 obj.update();  
 }  
 }  
  
 @Override  
 public String getSubState() {  
 return action;  
 }  
  
 @Override  
 public void setSubState(String state) {  
 this.action = state;  
 }  
}

Observer:

package designpattern.observer;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/29.  
 \*/*public abstract class Observer {  
 String name;  
 Subject subject;  
 public Observer(){}  
 public Observer(String name,Subject subject){  
 this.name = name;  
 this.subject = subject;  
 }  
 public abstract void update();  
}

MovieObserver:

package designpattern.observer;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/29.  
 \*/*public class MovieObserver extends Observer {  
 public MovieObserver(String name,Subject subject){  
 super(name,subject);  
 }  
  
 @Override  
 public void update() {  
 System.*out*.println(subject.getSubState()+", "+name+" : close the movie,begin work ");  
 }  
}

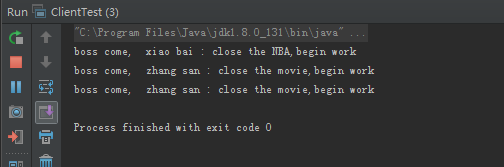
NBAObserver:

package designpattern.observer;  
*/\*\*  
 \* Created by Administrator on 2018/5/29.  
 \*/*public class NBAObserver extends Observer {  
 public NBAObserver(String name,Subject subject){  
 super(name,subject);  
 }  
 @Override  
 public void update() {  
 System.*out*.println(subject.getSubState()+", "+name+" : close the NBA,begin work ");  
 }  
}

ClientTest:

package designpattern.observer;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/29.  
 \*/*public class ClientTest {  
 public static void main(String[] args) {  
 Subject subject = new Secretary();  
 Observer obj1 = new NBAObserver("xiao bai",subject);  
 Observer obj2 = new MovieObserver("zhang san",subject);  
 subject.attach(obj1);  
 subject.attach(obj2);  
 subject.setSubState("boss come");  
 subject.notifys();  
   
 subject.detach(obj1);  
 subject.notifys();  
 }  
}

（4）运行结果



3.4 状态模式

3.4.1 模式名称

模式名称：状态模式

当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类。

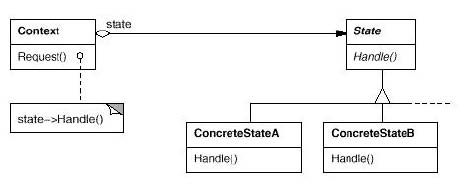
3.4.2 应用问题

状态模式的应用场景：当一个对象的行为取决于状态，并且它必须在运行时刻根据状态改变它的行为时，就可以考虑使用状态模式了。

状态模式的好处在于将与特定状态相关的行为局部化，并且将不同状态的行为分割开来。这样做消除了庞大的条件分支语句，状态模式通过把各种状态转移逻辑分布到State的子类之间，来减少相互间的依赖。

3.4.3 解决方案

状态模式的结构图：



如图所示：State是一个抽象类，在该类中封装引起状态变化的操作的方法，其子类为具体的状态类，实现特定行为下的特定状态转换。在Context类中将State类对象作为该类的属性，这样，当客户端定义一个具体的状态类，在该类的方法中就可以关联到具体的状态类的对应特定操作的特定状态转换实现。

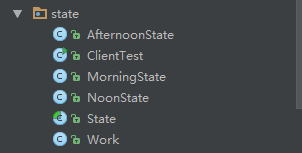
状态模式主要解决的是当控制一个对象状态转换的条件表达式过于复杂时的情况，把状态判断逻辑转移到表示不同状态的一系列当中，可以把复杂的判断逻辑简化，当然，如果这个状态判断很简单，那就没有必要用状态模式了。

3.4.4 效果

（1）示例描述

举个例子：比如工作状态：上午状态好，中午想睡觉，下午渐渐恢复，加班苦煎熬，其实是一种状态的转换，不同的时间，会有不同的状态。

（2）示例目录结构



（3）代码

State:

package designpattern.state;  
*/\*\*  
 \* Created by Administrator on 2018/5/31.  
 \*/*public abstract class State {  
 public State(){}  
 public abstract void writeProgram(Work work);  
}

Work:

package designpattern.state;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/31.  
 \*/*public class Work {  
 private State state;  
 private int hour;  
  
 public Work(){  
 this.state = new MorningState();  
 this.hour = 8;  
 }  
 public Work(State state,int hour){  
 this.state = state;  
 this.hour = hour;  
 }  
  
 public State getState() {  
 return state;  
 }  
  
 public void setState(State state) {  
 this.state = state;  
 }  
  
 public int getHour() {  
 return hour;  
 }  
  
 public void setHour(int hour) {  
 this.hour = hour;  
 }  
  
 public void writeProgram(){  
 this.state.writeProgram(this);  
 }  
}

Morning:

package designpattern.state;  
*/\*\*  
 \* Created by Administrator on 2018/5/31.  
 \*/*public class MorningState extends State {  
 @Override  
 public void writeProgram(Work work) {  
 if(work.getHour() >= 12 ) {  
 work.setState(new NoonState());  
 // System.out.println("curreent state is: " + work.getState() + ",and will to state: NoonState");  
 work.writeProgram();  
 }  
 else{  
 System.*out*.println("MorningState, i am energetic");  
 }  
 }  
}

NoonState:

package designpattern.state;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/31.  
 \*/*public class NoonState extends State {  
 @Override  
 public void writeProgram(Work work) {  
 if(work.getHour() >= 14) {  
 work.setState(new AfternoonState());  
 //System.out.println("curreent state is: " + work.getState() + ",and will to state: AfernoonState");  
 work.writeProgram();  
 }  
 else{  
 System.*out*.println("noonState, i am tired");  
 }  
 }  
}

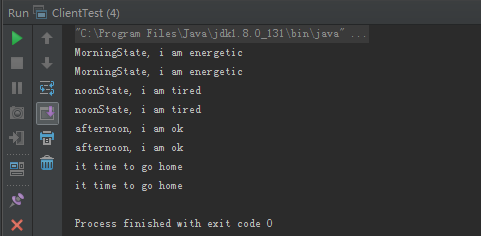
AfternoonState:

package designpattern.state;  
*/\*\*  
 \* Created by Administrator on 2018/5/31.  
 \*/*public class AfternoonState extends State {  
 @Override  
 public void writeProgram(Work work) {  
 if(work.getHour() >= 18) {  
 System.*out*.println("it time to go home");  
 }  
 else{  
 System.*out*.println("afternoon, i am ok");  
 }  
 }  
}

ClientTest:

package designpattern.state;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/31.  
 \*/*public class ClientTest {  
 public static void main(String[] args) {  
 Work work = new Work();  
  
 work.setHour(9);  
 work.writeProgram();  
  
 work.setHour(11);  
 work.writeProgram();  
  
 work.setHour(12);  
 work.writeProgram();  
  
 work.setHour(13);  
 work.writeProgram();  
  
 work.setHour(14);  
 work.writeProgram();  
  
 work.setHour(16);  
 work.writeProgram();  
  
 work.setHour(18);  
 work.writeProgram();  
  
 work.setHour(19);  
 work.writeProgram();  
 }  
}

（4）运行结果



3.5 策略模式

3.5.1 模式名称

模式名称：策略模式

它定义了算法家族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化，不会影响到使用算法的客户。

3.5.2 应用名称

使用策略模式:

（1）许多相关的类只在它们的行为上有所不同。

（2）我们需要不同的算法变体。

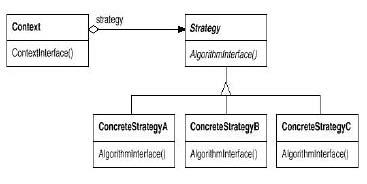
（3）算法使用客户不应该知道的数据。使用策略模式避免公开复杂的、特定于算法的数据结构。

（4）类定义了许多行为，这些行为在其操作中以多个条件语句的形式出现。与许多条件相反，将相关的有条件分支转移到它们自己的策略类中。

策略模式的应用场景：在分析过程中需要在不同时间应用不同的业务规则，就可以考虑使用策略模式来处理这种变化的可能性。常常与简单工厂模式结合使用。

3.5.3 解决方案

策略模式的结构图：



如上图所示：在Stategy类中，定义了所有支持的算法的公共接口，其子类是具体的策略类，封装了具体的算法或者行为，而Context类需要用一个策略类来配置，维护一个对策略对象的引用。

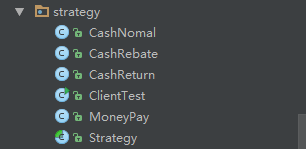
策略模式的优点在于定义了一系列的算法的方法，从概念上来将，所有这些算法完成的都是相同的工作，只是实现不同，它可以以相同的方式来调用所有算法，减少各种算法类与使用算法类之间的耦合。其次，简化了单元测试，因为每一个算法都有自己的类，可以通过自己的接口单独测试。还有，当不同行为堆砌在一个类中，就很难避免使用条件语句来选择合适的行为，将这些行为封装在一个个独立的策略类中，可以使用这些行为的类中消除条件语句。

3.5.4 效果

（1）示例描述

举个例子：在商场，关于现金收费，有时价格正常，有时打折收费，有时返利收费，对于这种情况，在收费时价格优惠不同的情况就可以使用策略模式。

（2）示例目录结构



（3）代码

Strategy:

package designpattern.strategy;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/31.  
 \*/*public abstract class Strategy {  
 public abstract double acceptCash(double money);  
}

MoneyPay:

package designpattern.strategy;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/31.  
 \*/*public class MoneyPay {  
 private Strategy strategy;  
 public MoneyPay(){}  
 public MoneyPay(int type){  
 switch(type){  
 case 1:  
 CashNomal cashNomal = new CashNomal();  
 this.strategy = cashNomal;  
 break;  
 case 2:  
 CashRebate cashRebate = new CashRebate(0.8);  
 this.strategy = cashRebate;  
 break;  
 case 3:  
 CashReturn cashReturn = new CashReturn(300,100);  
 this.strategy = cashReturn;  
 break;  
 }  
  
 }  
  
 public Strategy getStrategy() {  
 return strategy;  
 }  
  
 public void setStrategy(Strategy strategy) {  
 this.strategy = strategy;  
 }  
  
 public double acceptCash(double money){  
 return this.strategy.acceptCash(money);  
 }  
}

CashNomal:

package designpattern.strategy;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/31.  
 \*/*public class CashNomal extends Strategy {  
 @Override  
 public double acceptCash(double money) {  
 return money;  
 }  
}

CashRebate:

package designpattern.strategy;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/31.  
 \*/*public class CashRebate extends Strategy {  
 private double moneyRate = 1;  
 public CashRebate(){}  
 public CashRebate(double rate){  
 this.moneyRate = rate;  
 }  
 @Override  
 public double acceptCash(double money) {  
 return money\*this.moneyRate;  
 }  
}

CashReturn:

package designpattern.strategy;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/31.  
 \*/*public class CashReturn extends Strategy {  
 private double moneyCondition ;  
 private double moneyReturn ;  
 public CashReturn(){  
 this.moneyCondition = 0;  
 this.moneyReturn = 0;  
 }  
  
 public CashReturn(double moneyCondition, double moneyReturn){  
 this.moneyCondition = moneyCondition;  
 this.moneyReturn = moneyReturn;  
 }  
  
 public double getMoneyCondition() {  
 return moneyCondition;  
 }  
  
 public void setMoneyCondition(double moneyCondition) {  
 this.moneyCondition = moneyCondition;  
 }  
  
 public double getMoneyReturn() {  
 return moneyReturn;  
 }  
  
 public void setMoneyReturn(double moneyReturn) {  
 this.moneyReturn = moneyReturn;  
 }  
  
 @Override  
 public double acceptCash(double money) {  
 double result = money;  
 if(money >= moneyCondition){  
 result = money - (money/moneyCondition) \* moneyReturn;  
 }  
 return result;  
 }  
}

ClientTest:

package designpattern.strategy;  
  
*/\*\*  
 \* Created by Administrator on 2018/5/31.  
 \*/*public class ClientTest {  
 public static void main(String[] args) {  
 MoneyPay moneyPay = new MoneyPay(2);  
 double money = moneyPay.acceptCash(200);  
 System.*out*.println("should pay money:"+money);  
  
 MoneyPay moneyPay1 = new MoneyPay(3);  
 double money1 = moneyPay1.acceptCash(400);  
 System.*out*.println("should pay money:"+money1);  
  
 MoneyPay moneyPay2 = new MoneyPay(1);  
 double money2 = moneyPay2.acceptCash(200);  
 System.*out*.println("should pay money:"+money2);  
 }  
}

（4）运行结果

