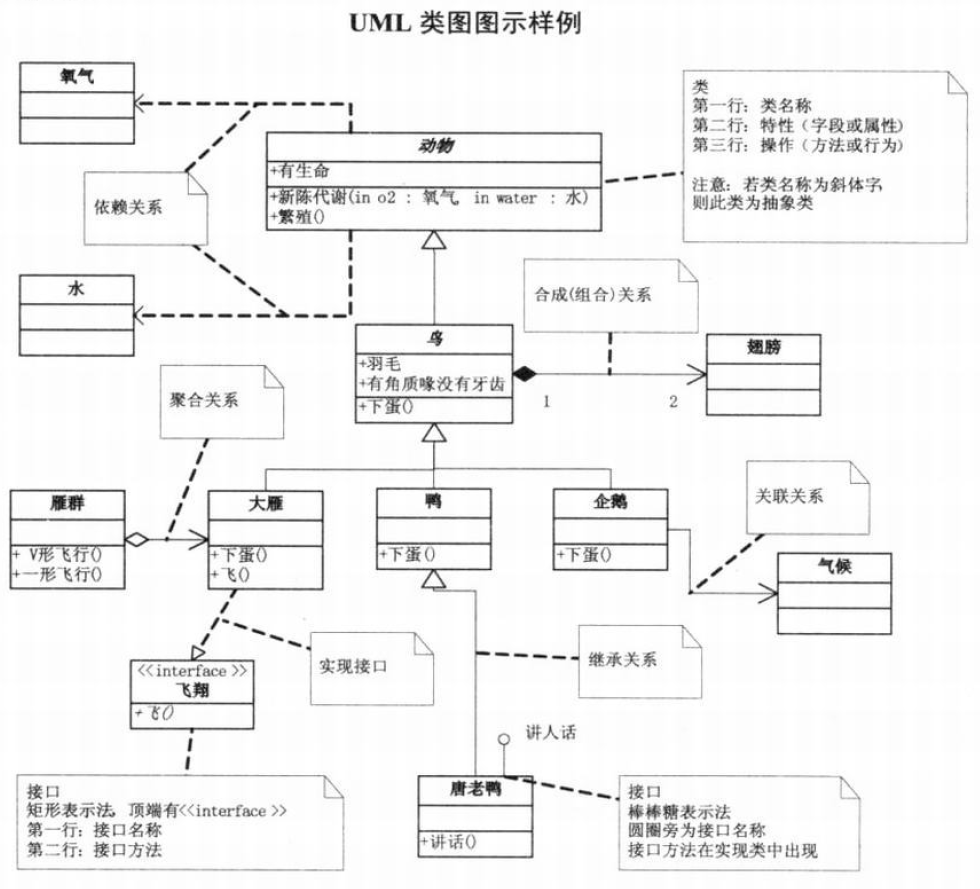
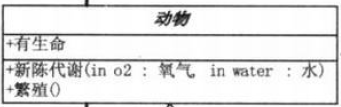
## 1.10 简单工厂模式

定义一个工厂方法，其返回值类型为父类对象，根据传入参数的不同，生成不同子类的引用，以此来生产各种子类对象。

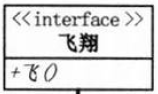
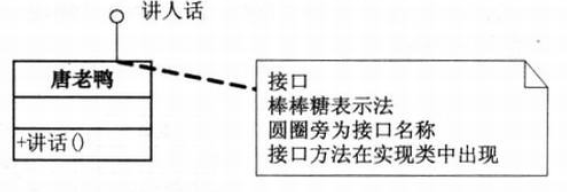
优点：只需要传入参数就可以创建对象，而不必知道对象是如何创建出来的。

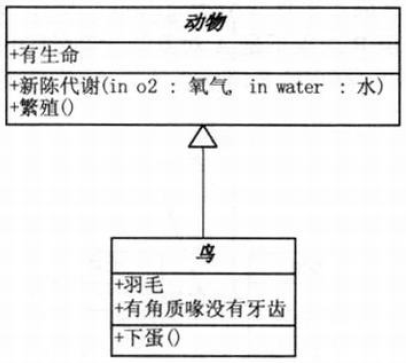
缺点：每添加一个新的子类，都要在工厂类中进行改动，以致代码需要重新编译部署。

## 1.11 UML类图

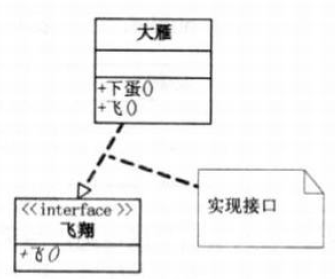
①类图：“动物”矩形框代表一个类（class），类图分三层，第一层显示类的名称，如果是抽象类就用斜体显示，第二层是类的特性，通常就是类的属性和字段。第三层是类的操作，通常是方法和行为，注意前面的符号，“+”表示public，“-”表示private，“#”表示protected，示例图如下：

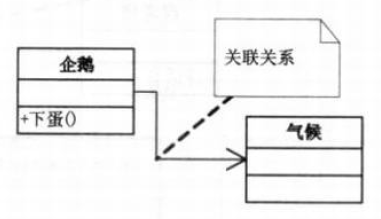
②接口图：左下角的“飞翔”表示一个接口图，与类图的区别是顶端有<<interface>>第一行是接口名称，第二行是接口方法。如图：

接口的另外一种表示方法，俗称棒棒糖表示法，如图：

③继承关系：继承关系用空心三角形+实线来表示，如图：

④实现关系：实现接口用空心三角形+虚线来表示，如图：



⑤关联关系：当一个类知道另一个类时，用关联。关联关系用实线箭头来表示。

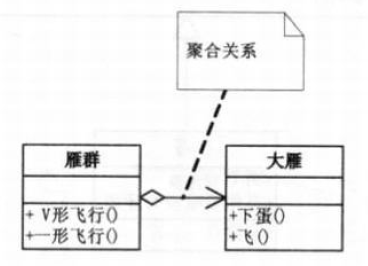
在企鹅类Penguin中，引用到了气候Climate对象:

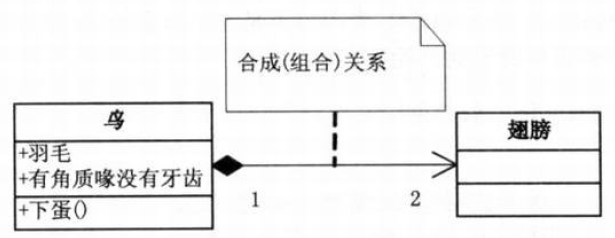
public class Penguin{

private Climate climate;

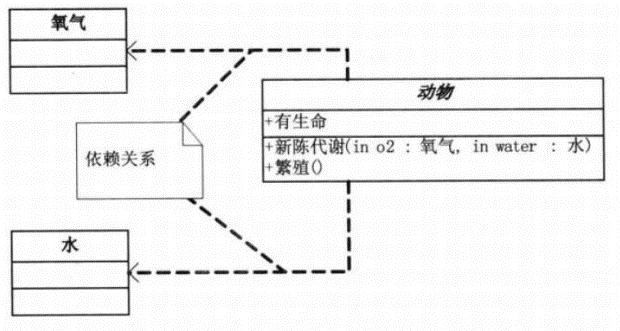
}

⑥聚合关系：每只大雁都属于一个雁群，一个雁群可以有多只大雁。所以它们之间就满足聚合关系。聚合表示一种弱的“拥有”关系，体现的是A对象包含B对象，但B对象不是A对象的一部分。聚合关系用空心的菱形+实现箭头表示。如图：



⑦组合关系：是一种强的“拥有”关系，体现了严格的部分和整体的关系，部分和整体的生命周期一样。在这里鸟和其翅膀就是组合关系。组合关系用实心的菱形+实线箭头来表示。组合关系的连线两端各有一个数字，这被称为基数，表名这一端的类可以有几个实例，很显然鸟应该有两个翅膀。如果一个类可能有无数个实例，就用“n”来表示。关联关系、聚合关系也可以有基数。如图：

⑧依赖关系：用虚线箭头来表示。如图：

abstract class Animal{

public void Matebolism(Oxygen oxygen,Water water){

}

}

# 第2章 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。

实现方式：

1. 首先有一个父类的算法类

abstract class Strategy{

//公共的算法方法

public abstract void Algorighminterface(){}

}

1. 定义具体算法，继承父类算法类

覆盖父类方法，实现自己的算法

③定义一个Context类，用具体算法来配置，维护一个对父类算法对象的引用。

class Context{

Strategy strategy;

public Context(Strategy strategy){//这里传入具体的算法类

this.strategy=strategy;

}

//上下文接口

public void ContextInterface(){

strategy.Aligorithminterface();

}

}

# 第3章 单一职责原则

就一个类而言，应该仅有一个引起它变化的原因。

# 第4章 开放-封闭原则

对于扩展是开放的，对于修改是封闭的

# 第5章 依赖倒转原则

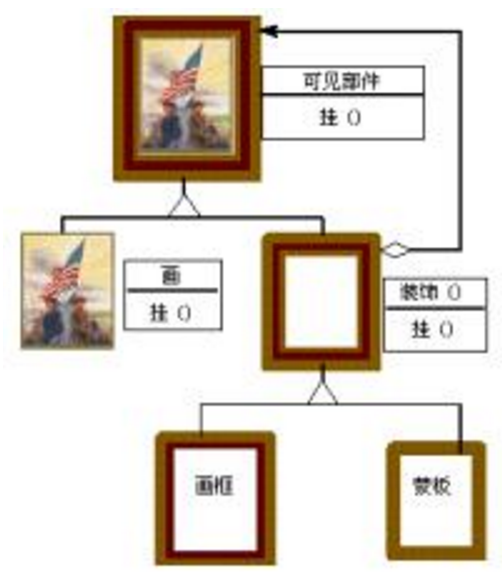
高层模块不应该依赖于底层模块，两个都应该依赖抽象。

## 5.4 里氏替换原则

子类型必须能够替换掉它们的父类型。

正是由于子类型的可替换性才使得使用父类类型的模块在无需修改的情况下就可以扩展。

# 第6章 装饰模式

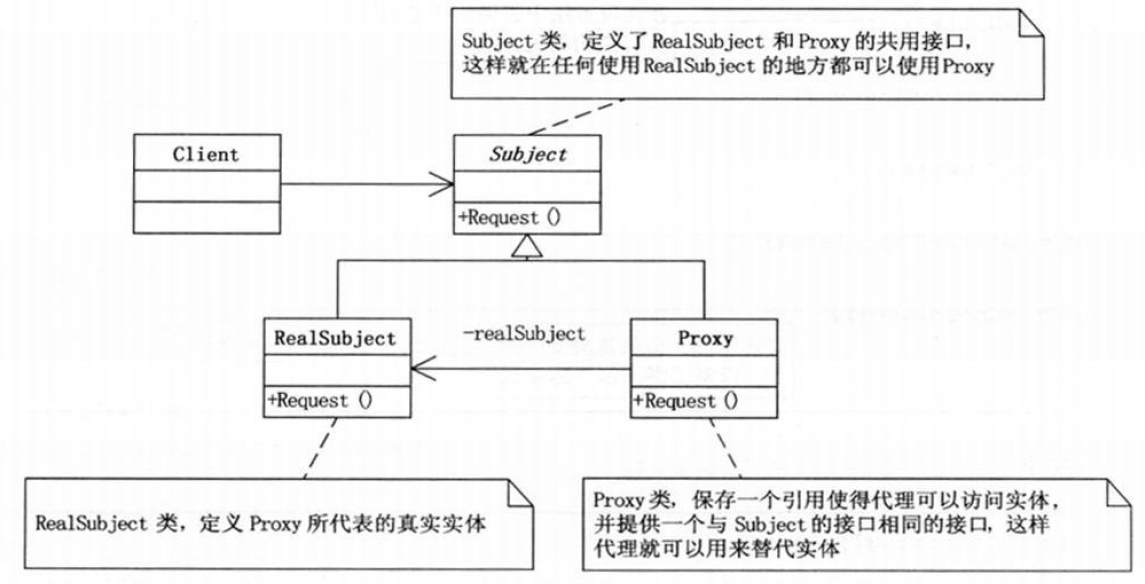


通过上图就可以理解装饰模式的含义。

最原始的类就是左侧的“画”类，要拓展画类的功能不必在其中添加新的代码，而只需要继承“装饰”类，在新的子类中在“画”类的核心功能上添加新的代码就可以了，这样即保留了“画”类最原始的功能，又能动态的用装饰类的子类为其添加新功能。

好处：有效的把类的核心职责和装饰功能区分开，而且可以去除相关类中重复的装饰逻辑。

# 第7章 代理模式



为其他对象提供一种代理以控制对这个对象的访问。

控制这个对象的访问体现在下列代码中：

class Proxy extends Subject{

RealSubject realSubject;

public void Request(){

if(realSubject==null){

realSubject=new RealSubject();

}

}

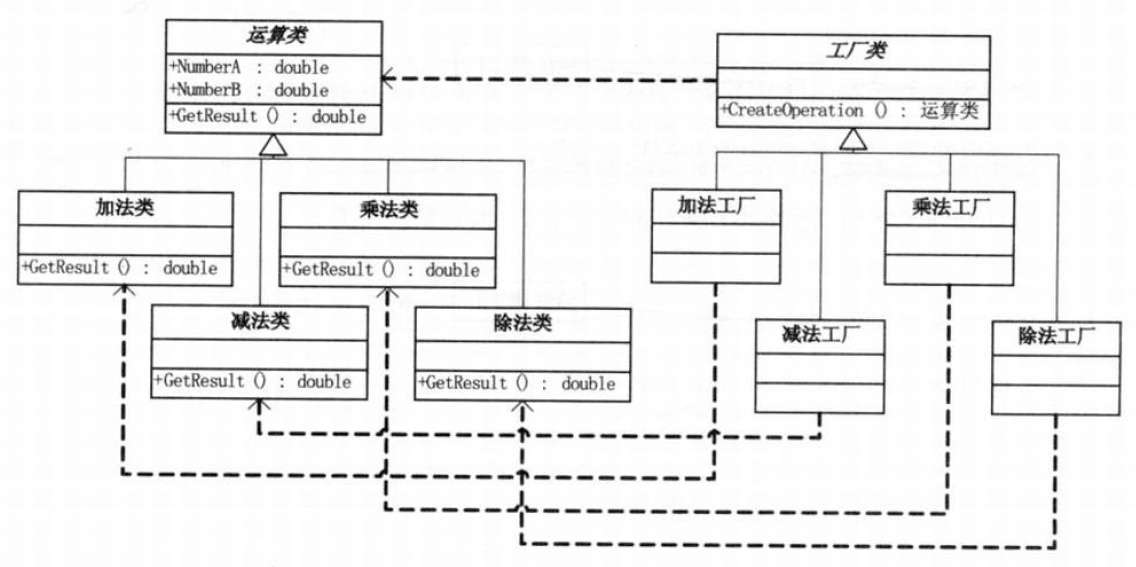
realSubject.Request();

}

代理模式与装饰模式的区别：代理模式可以有对被代理对象的访问控制，而装饰模式只是为了扩展核心类的功能。

# 第8章 工厂方法模式

UML图：



# 第9章 原型模式

这里先要说一下浅拷贝和深拷贝：

①浅拷贝：对值类型的成员变量进行值的复制，对引用类型成员变量只复制引用，不复制引用的对象。

（1）实例一：

**public** **class** Prototype **implements** Cloneable {

**private** String name;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Object clone(){

**try** {

**return** **super**.clone();

} **catch** (CloneNotSupportedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

**return** **null**;

}

}

}

**public** **class** TestMain {

//这里是浅拷贝

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

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

*testPrototype*();

}

**private** **static** **void** testPrototype(){

Prototype pro=**new** Prototype();

pro.setName("original object");

Prototype pro1=(Prototype) pro.clone();

pro1.setName("change object1");

System.*out*.println("original object:"+pro.getName());

System.*out*.println("change object:"+pro1.getName());

}

}

输出结果：

original object:original object

change object:change object1

可以看到pro1.setName("change object1");这一行代码并没有覆盖前面的name的值。

（2）实例二：

public class Prototype{  
 private String name;

 public String getName() {  
  return name;  
 }

 public void setName(String name) {  
  this.name = name;  
 }   
   
}  
public class NewPrototype implements Cloneable {  
   
 private String id;  
   
 public String getId() {  
  return id;  
 }

 public void setId(String id) {  
  this.id = id;  
 }

 private Prototype prototype;  
   
 public Prototype getPrototype() {  
  return prototype;  
 }

 public void setPrototype(Prototype prototype) {  
  this.prototype = prototype;  
 }

 public Object clone(){   
  try {  
   return super.clone();  
  } catch (CloneNotSupportedException e) {  
   e.printStackTrace();  
   return null;  
  }    
 }

}  
public class TestMain {

 public static void main(String[] args) {  
  // TODO Auto-generated method stub  
  testPrototype();  
 }  
   
 private static void testPrototype(){  
  Prototype pro = new Prototype();  
  pro.setName("original object");  
  NewPrototype newObj = new NewPrototype();  
  newObj.setId("test1");  
  newObj.setPrototype(pro);  
    
  NewPrototype copyObj = (NewPrototype)newObj.clone();  
  copyObj.setId("testCopy");  
  copyObj.getPrototype().setName("changed object");  
    
  System.out.println("original object id:" + newObj.getId());  
  System.out.println("original object name:" + newObj.getPrototype().getName());  
    
  System.out.println("cloned object id:" + copyObj.getId());  
  System.out.println("cloned object name:" + copyObj.getPrototype().getName());  
   
 }

}

结果:  
original object id:test1  
original object name:changed object  
cloned object id:testCopy  
cloned object name:changed object

正是因为浅拷贝对引用类型只复制引用，所以后面为name赋值，也会改变前面的值。

②深拷贝：对值类型的成员变量进行值的复制，对引用类型成员变量也进行引用对象的复制。

public class Prototype implements Cloneable {  
 private String name;

 public String getName() {  
  return name;  
 }

 public void setName(String name) {  
  this.name = name;  
 }

 public Object clone() {  
  try {   
   return super.clone();  
  } catch (CloneNotSupportedException e) {     
   e.printStackTrace();  
   return null;  
  }  
 }   
   
}

public class NewPrototype implements Cloneable {  
   
 private String id;  
   
 public String getId() {  
  return id;  
 }

 public void setId(String id) {  
  this.id = id;  
 }

 private Prototype prototype;  
   
 public Prototype getPrototype() {  
  return prototype;  
 }

 public void setPrototype(Prototype prototype) {  
  this.prototype = prototype;  
 }

 public Object clone(){  
  NewPrototype ret = null;  
  try {  
   ret = (NewPrototype)super.clone();  
   ret.prototype = (Prototype)this.prototype.clone();  
   return ret;  
  } catch (CloneNotSupportedException e) {  
   e.printStackTrace();  
   return null;  
  }    
 }

}

public class TestMain {

 /\*\*  
  \* @param args  
  \*/  
 public static void main(String[] args) {  
  testDeepCopy();  
 }  
   
 private static void testDeepCopy(){  
  Prototype pro = new Prototype();  
  pro.setName("original object");  
  NewPrototype newObj = new NewPrototype();  
  newObj.setId("test1");  
  newObj.setPrototype(pro);  
    
  NewPrototype copyObj = (NewPrototype)newObj.clone();  
  copyObj.setId("testCopy");  
  copyObj.getPrototype().setName("changed object");  
    
  System.out.println("original object id:" + newObj.getId());  
  System.out.println("original object name:" + newObj.getPrototype().getName());  
    
  System.out.println("cloned object id:" + copyObj.getId());  
  System.out.println("cloned object name:" + copyObj.getPrototype().getName());  
    
 }

}

结果:  
original object id:test1  
original object name:original object  
cloned object id:testCopy  
cloned object name:changed object

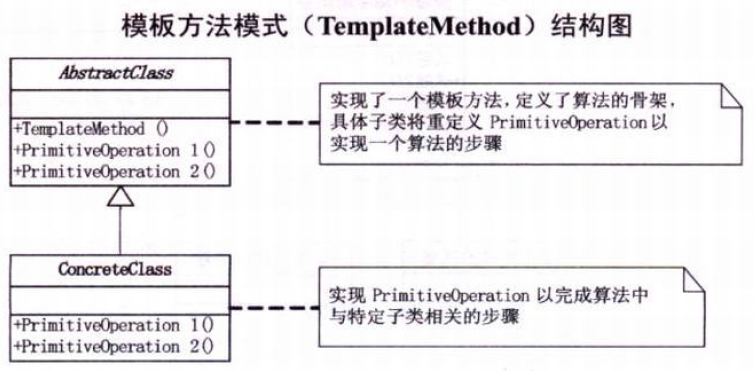
这里重点分析这段代码：

public Object clone(){  
  NewPrototype ret = null;  
  try {  
   ret = (NewPrototype)super.clone();①  
   ret.prototype = (Prototype)this.prototype.clone();②  
   return ret;  
  } catch (CloneNotSupportedException e) {  
   e.printStackTrace();  
   return null;  
  }    
 }

}

只进行到位置①表示浅拷贝，ret中的值类型进行复制，Prototype引用类型只是进行了引用的复制；进行到②才算进行了深拷贝，将Prototype引用类型中的值类型进行复制，这样对于ret来说，里面的所有值类型都进行了复制，所以最后的结果是不相同的，对拷贝对象的操作并不会影响前面对象。

# 第10章 模板方法模式

定义一个操作中算法的骨架，而将一些操作延伸到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的默写特定步骤。

# 第11章 迪米特原则

也叫最少知识原则。

# 第12章 外观模式

又叫门面模式

# 第13章 建造者模式