深圳大学
Shenzhen University

第8讲

抽象工厂模式&建造者模式&原型模式

软件体系结构与设计模式

Software Architecture & Design Pattern

深圳大学计算机与软件学院

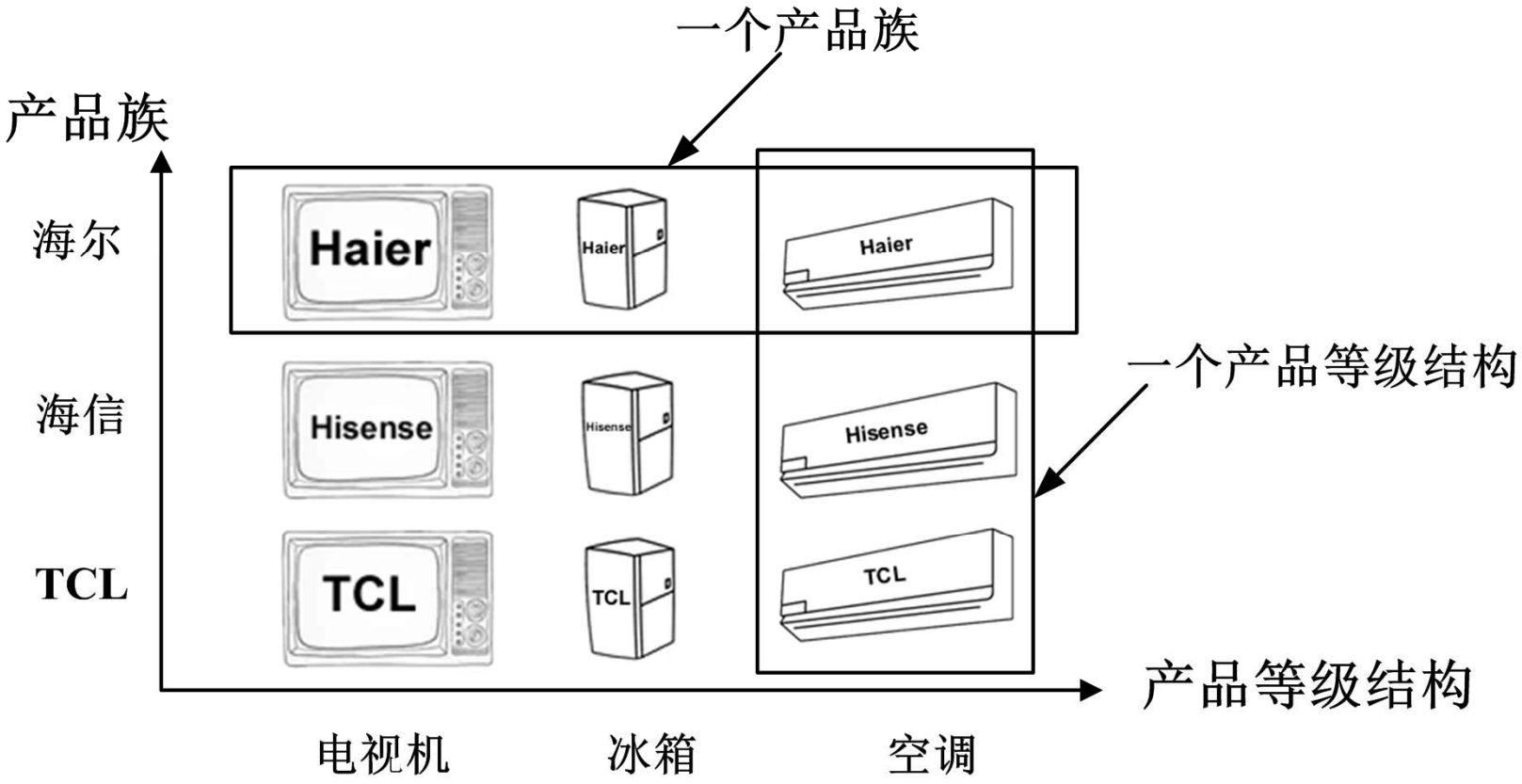
主要内容

- 7.1 抽象工厂模式
- 7.2建造者模式
- 7.3原型模式

7.1 抽象工厂模式

- **抽象工厂模式模式动机**
  - **产品等级结构：**产品等级结构即产品的继承结构，例如一个抽象类是电视机，其子类有海尔电视机、海信电视机、TCL电视机，则抽象电视机与具体品牌的电视机之间构成了一个产品等级结构，抽象电视机是父类，而具体品牌的电视机是其子类。
  - **产品族：**在抽象工厂模式中，产品族是指由同一个工厂生产的，位于不同产品等级结构中的一组产品，例如海尔电器工厂生产的海尔电视机、海尔电冰箱，海尔电视机位于电视机产品等级结构中，海尔电冰箱位于电冰箱产品等级结构中。

抽象工厂模式动机

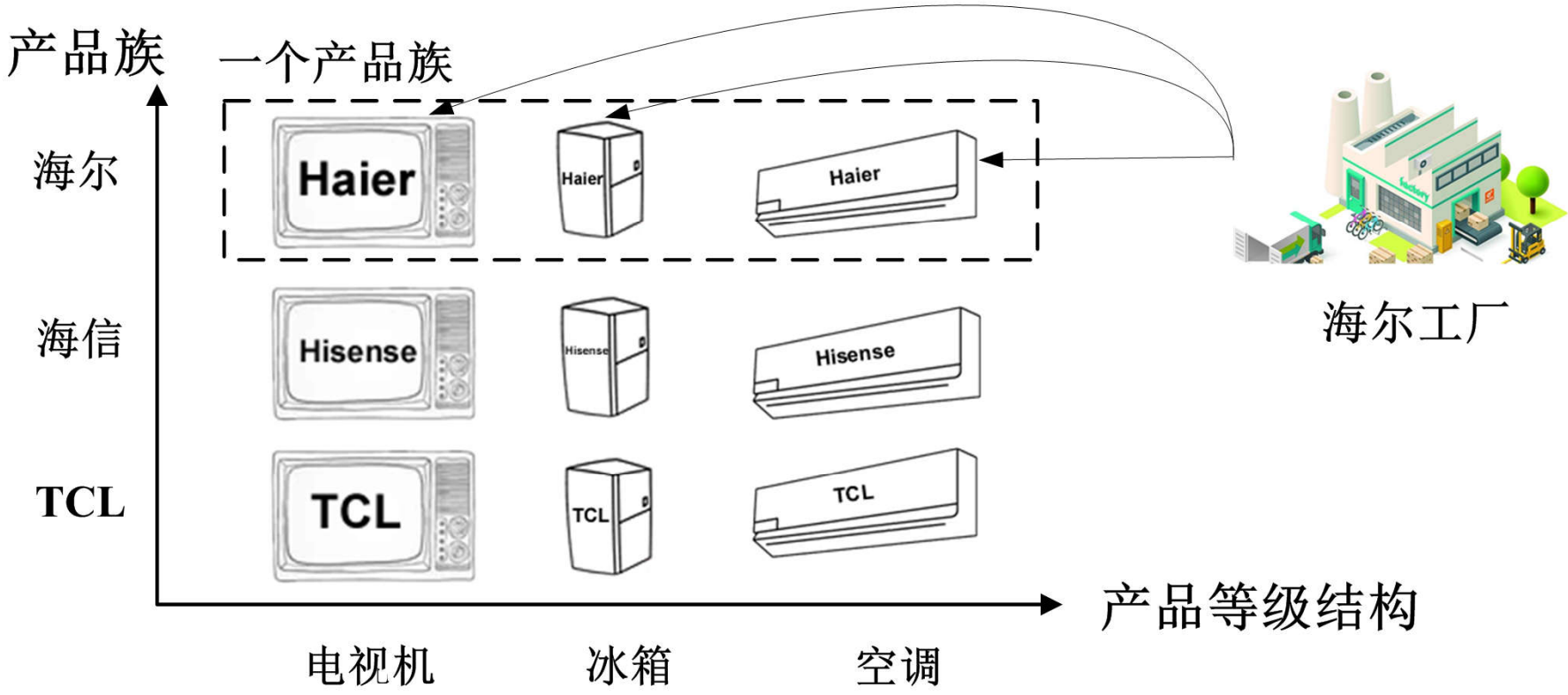


产品族与产品等级结构示意图

抽象工厂模式定义

- 抽象工厂模式(Abstract Factory Pattern)：提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。
- 抽象工厂模式又称为Kit模式，属于对象创建型模式。

抽象工厂模式定义

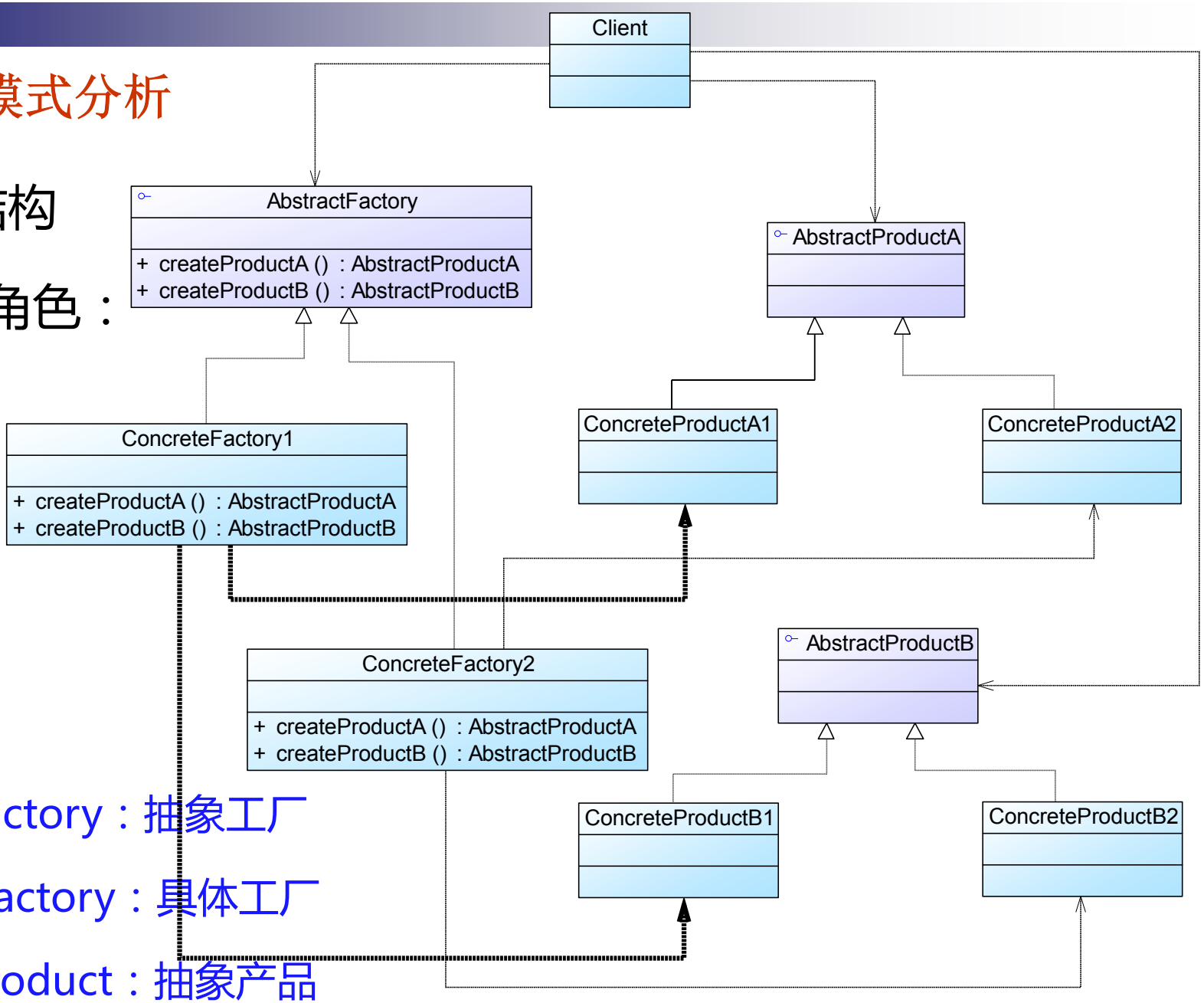


抽象工厂模式示意图

抽象工厂模式分析

- 模式结构

包含如下角色：



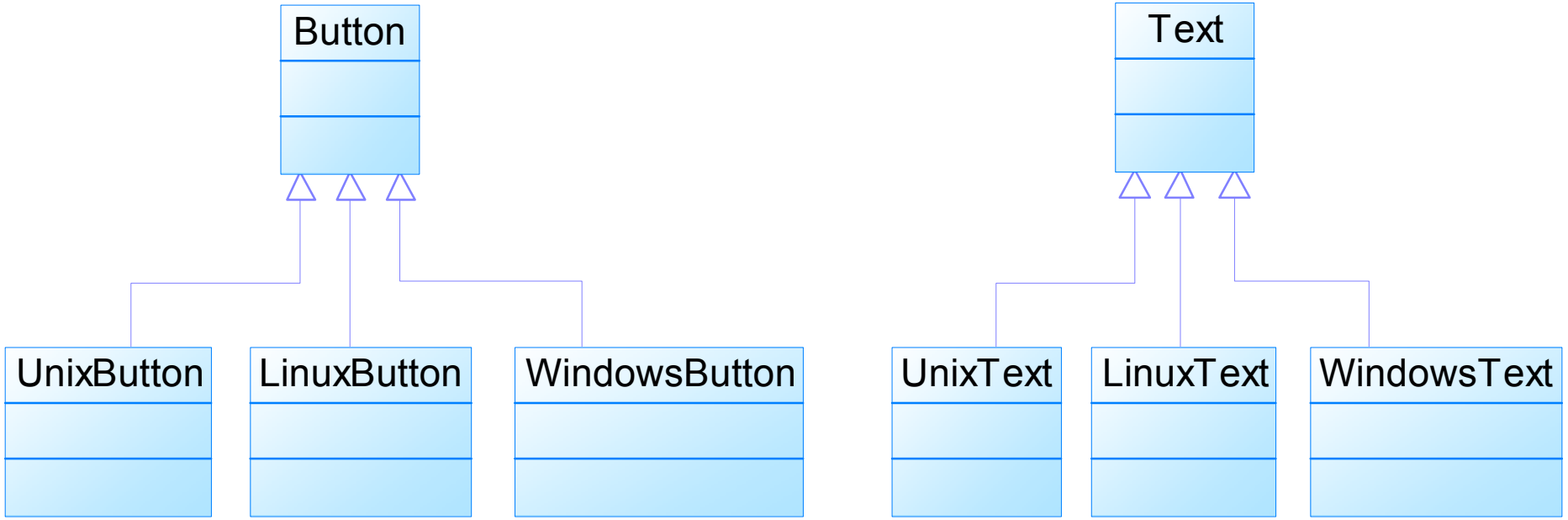
AbstractFactory：抽象工厂

ConcreteFactory：具体工厂

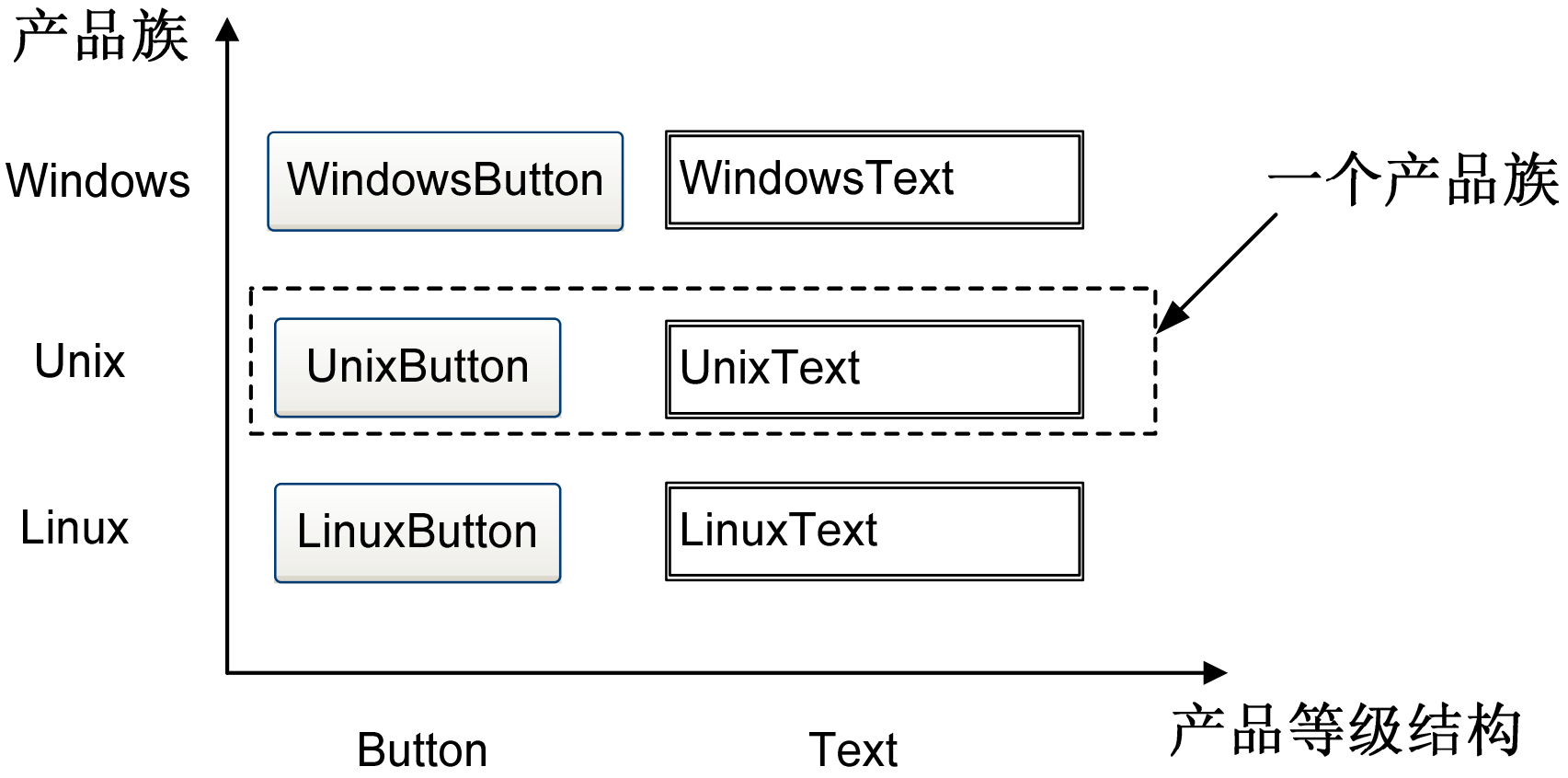
AbstractProduct：抽象产品

ConcreteProduct：具体产品

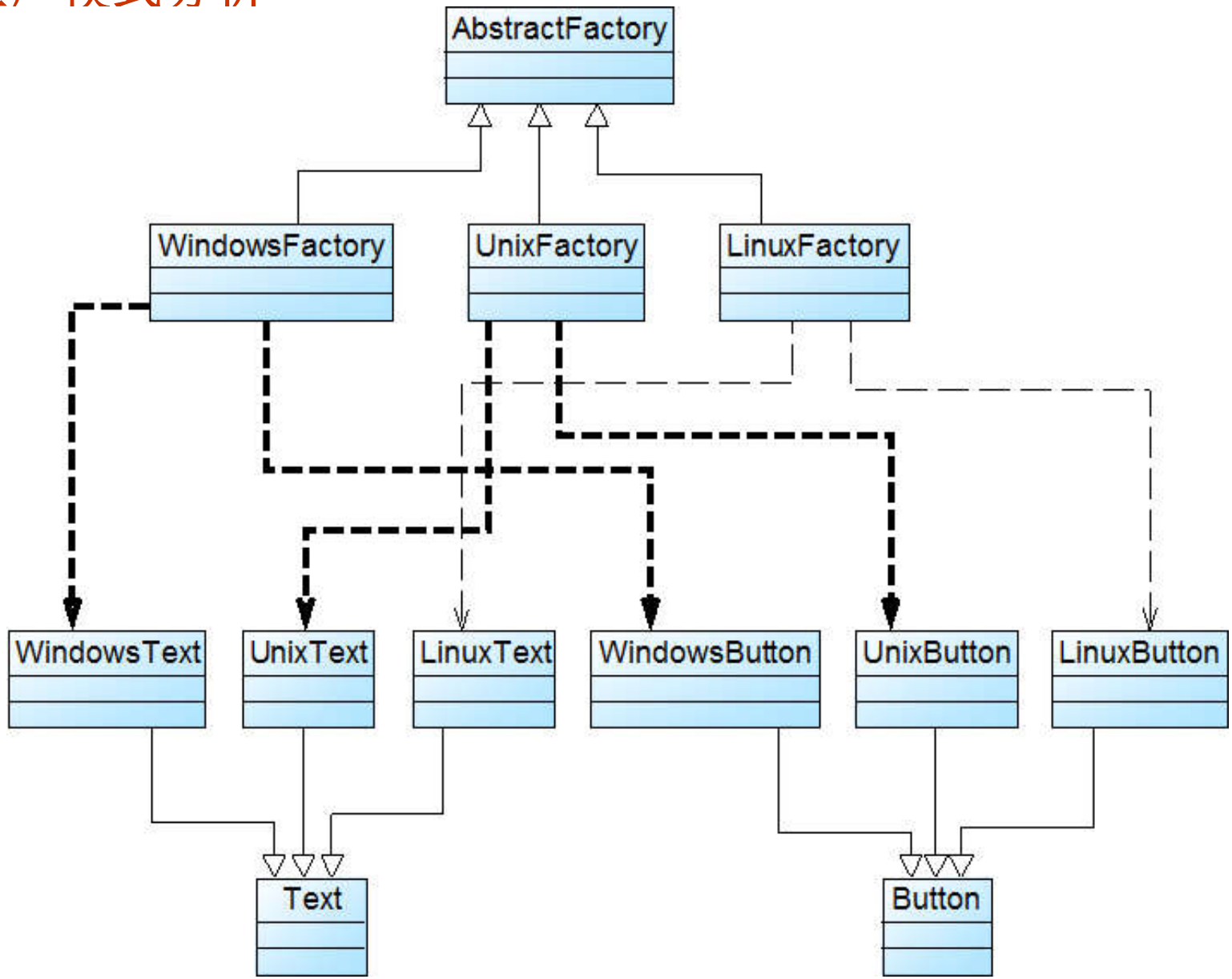
抽象工厂模式分析


Button
Text
UnixButton
LinuxButton
WindowsButton
UnixText
LinuxText
WindowsText

抽象工厂模式分析


产品族
Windows
WindowsButton
WindowsText
一个产品族
Unix
UnixButton
UnixText
Linux
LinuxButton
LinuxText
Button
Text
产品等级结构

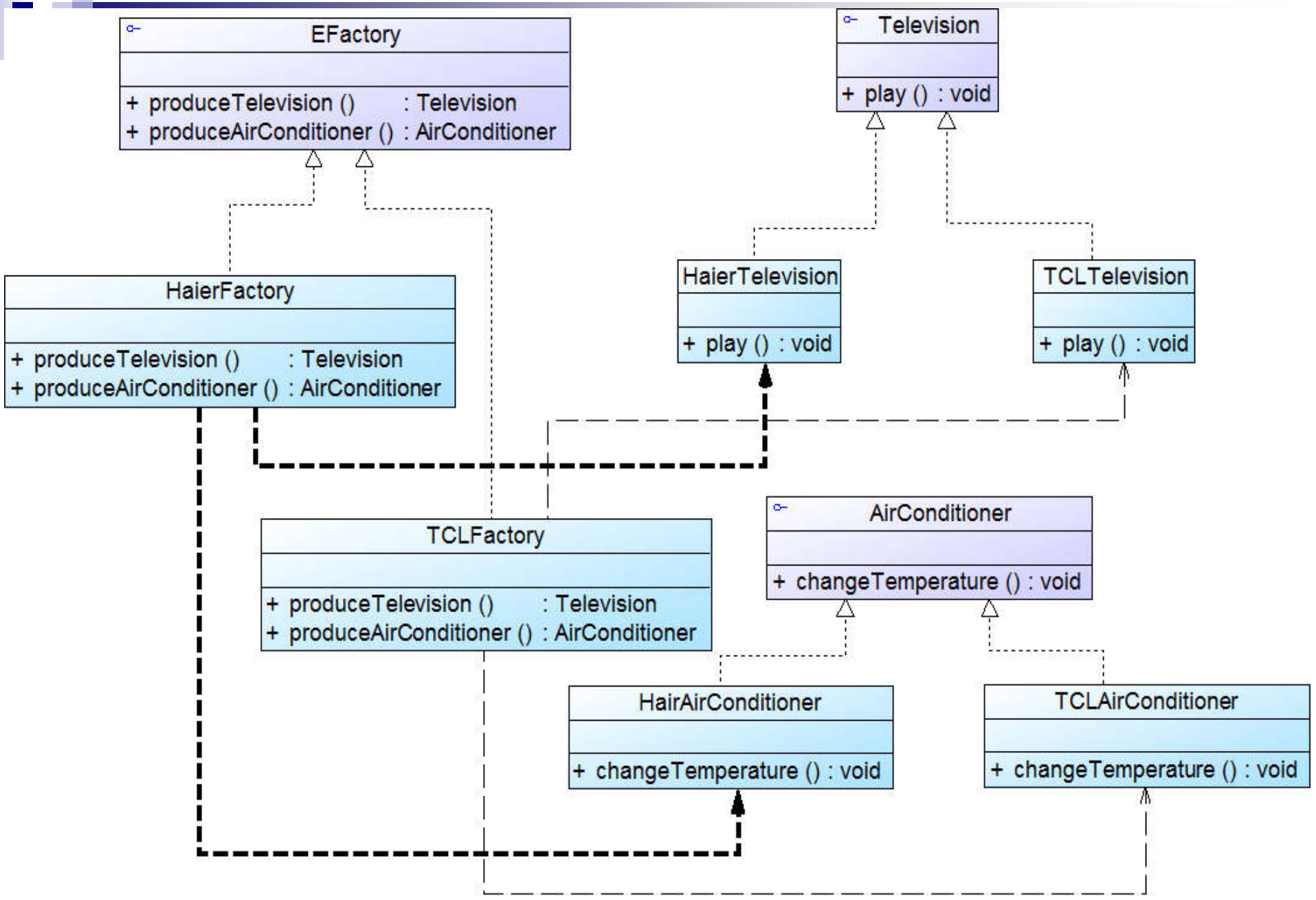
抽象工厂模式分析


AbstractFactory
WindowsFactory
UnixFactory
LinuxFactory
WindowsText
UnixText
LinuxText
WindowsButton
UnixButton
LinuxButton
Text
Button

抽象工厂模式实例

电器工厂：实例说明

- 一个电器工厂可以产生多种类型的电器，如海尔工厂可以生产海尔电视机、海尔空调等，TCL工厂可以生产TCL电视机、TCL空调等，相同品牌的电器构成一个产品族，而相同类型的电器构成了一个产品等级结构，现使用抽象工厂模式模拟该场景。


EFactory
+ produceTelevision () : Television
+ produceAirConditioner () : AirConditioner
Television
+ play () : void
HaierFactory
+ produceTelevision () : Television
+ produceAirConditioner () : AirConditioner
HaierTelevision
+ play () : void
TCLTelevision
+ play () : void
TCLFactory
+ produceTelevision () : Television
+ produceAirConditioner () : AirConditioner
AirConditioner
+ changeTemperature () : void
HairAirConditioner
+ changeTemperature () : void
TCLAirConditioner
+ changeTemperature () : void

抽象工厂模式实例与解析

- **抽象工厂模式实例**
  - **电器工厂：参考代码**
- DesignPatterns之abstractfactory包

Television.java

```
1  package abstractfactory;
2
3  public interface Television
4  {
5      public void play();
6  }
```

AirConditioner.java

```
1  package abstractfactory;
2
3  public interface AirConditioner
4  {
5      public void changeTemperature();
6  }
```

EFactory.java

```
1  package abstractfactory;
2
3  public interface EFactory
4  {
5      public Television produceTelevision();
6      public AirConditioner produceAirConditioner();
7  }
```

TCLTelevision.java

```java
package abstractfactory;

public class TCLTelevision implements Television
{
    public void play()
    {
        System.out.println("TCL电视机播放中......");
    }
}
```

HaierTelevision.java

```java
package abstractfactory;

public class HaierTelevision implements Television
{
    public void play()
    {
        System.out.println("海尔电视机播放中......");
    }
}
```

HairAirConditioner.java

```java
package abstractfactory;

public class HairAirConditioner implements AirConditioner
{
    public void changeTemperature()
    {
        System.out.println("海尔空调温度改变中.....");
    }
}
```

TCLAirConditioner.java

```java
package abstractfactory;

public class TCLAirConditioner implements AirConditioner
{
    public void changeTemperature()
    {
        System.out.println("TCL空调温度改变中.....");
    }
}
```

HaierFactory.java

```java
package abstractfactory;

public class HaierFactory implements EFactory
{
    public Television produceTelevision()
    {
        return new HaierTelevision();
    }

    public AirConditioner produceAirConditioner()
    {
        return new HairAirConditioner();
    }
}
```

TCLFactory.java

```java
package abstractfactory;

public class TCLFactory implements EFactory
{
    public Television produceTelevision()
    {
        return new TCLTelevision();
    }

    public AirConditioner produceAirConditioner()
    {
        return new TCLAirConditioner();
    }
}
```

XMLUtil.java

```java
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.SAXException;
import java.io.*;
public class XMLUtil
{
//该方法用于从XML配置文件中提取具体类类名，并返回一个实例对象
    public static Object getBean()
    {
        try
        {
            //创建文档对象
            DocumentBuilderFactory dFactory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = dFactory.newDocumentBuilder();
            Document doc;
            doc = builder.parse(new File("AbstractFactoryconfig.xml"));

            //获取包含类名的文本节点
            NodeList nl = doc.getElementsByTagName("className");
            Node classNode=nl.item(0).getFirstChild();
            String cName=classNode.getNodeValue();

            //通过类名生成实例对象并将其返回
            Class c=Class.forName(cName);
            Object obj=c.newInstance();
            return obj;
          }
```

XMLUtil.java

```java
            catch(Exception e)
            {
                e.printStackTrace();
                return null;
            }
        }
}
```

AbstractFactoryconfig.xml

```xml
<?xml version="1.0"?>
<config>
    <className>abstractfactory.HaierFactory</className>
</config>
```

Client.java

```java
package abstractfactory;

public class Client
{
    public static void main(String args[])
    {
        try
        {
            EFactory factory;
            Television tv;
            AirConditioner ac;
            factory=(EFactory)XMLUtil.getBean();
            tv=factory.produceTelevision();
            tv.play();
            ac=factory.produceAirConditioner();
            ac.changeTemperature();
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

抽象工厂模式效果与应用

- **抽象工厂模式优点：**
  - 隔离了具体类的生成，使得客户端并不需要知道什么被创建
  - 当一个产品族中的多个对象被设计成一起工作时，它能够保证客户端始终只使用同一个产品族中的对象
  - 增加新的产品族很方便，无须修改已有系统，符合开闭原则

抽象工厂模式效果与应用

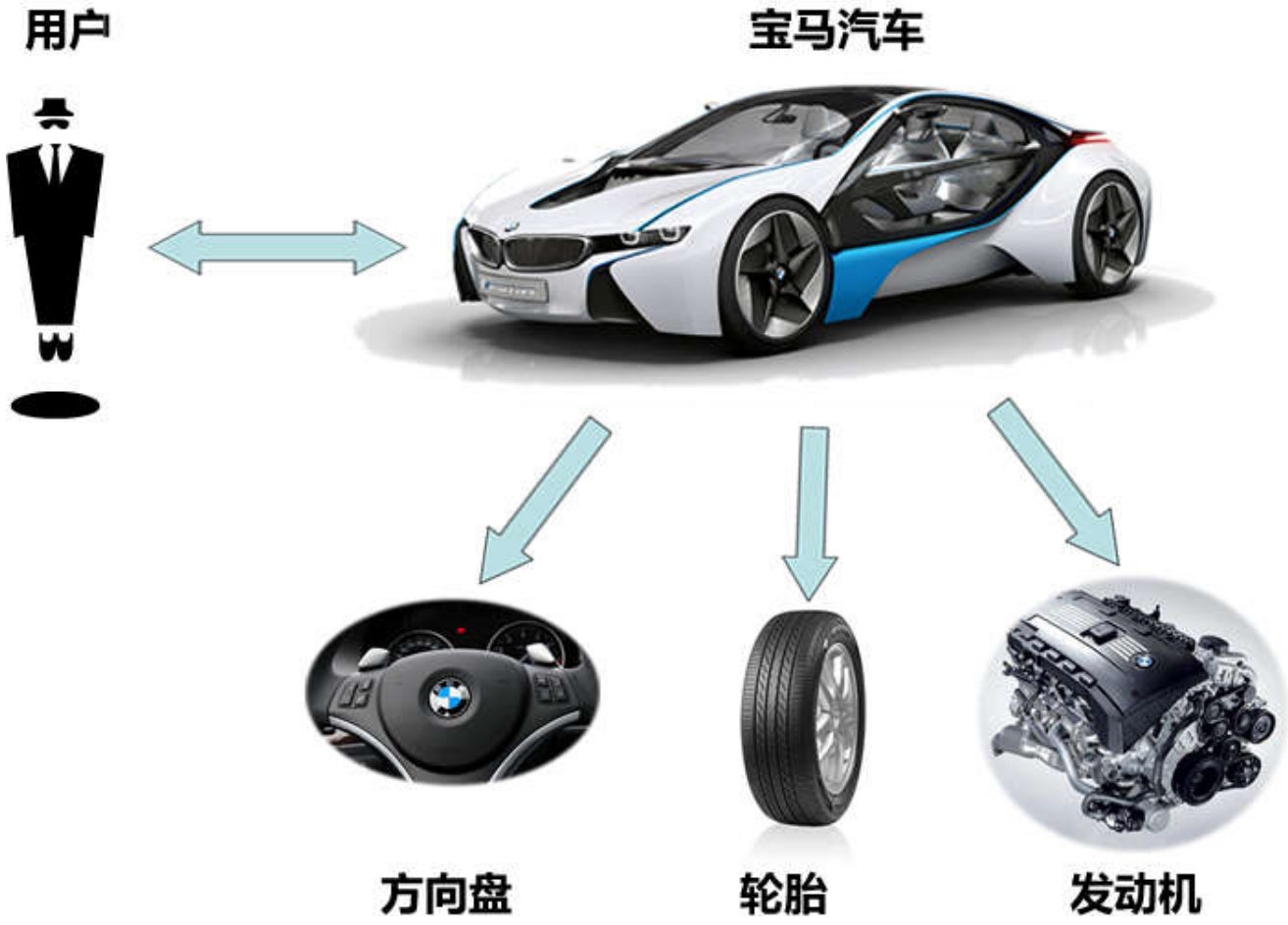
- **抽象工厂模式缺点：**
  - 增加新的产品等级结构麻烦，需要对原有系统进行较大的修改，甚至需要修改抽象层代码，这显然会带来较大的不便，违背了开闭原则

抽象工厂模式效果与应用

- **在以下情况下可以使用抽象工厂模式：**
  - 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节
  - 系统中有多于一个的产品族，但每次只使用其中某一产品族
  - 属于同一个产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来
  - 产品等级结构稳定，在设计完成之后不会向系统中增加新的产品等级结构或者删除已有的产品等级结构

7.2建造者模式

- **建造者模式动机**
  - **创建复杂对象**



建造者模式动机

- **分析**
  - 如何将这些部件组装成一辆完整的汽车并返回给用户？

建造者模式

建造者模式定义

- 是对象创建型模式

建造者模式：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。

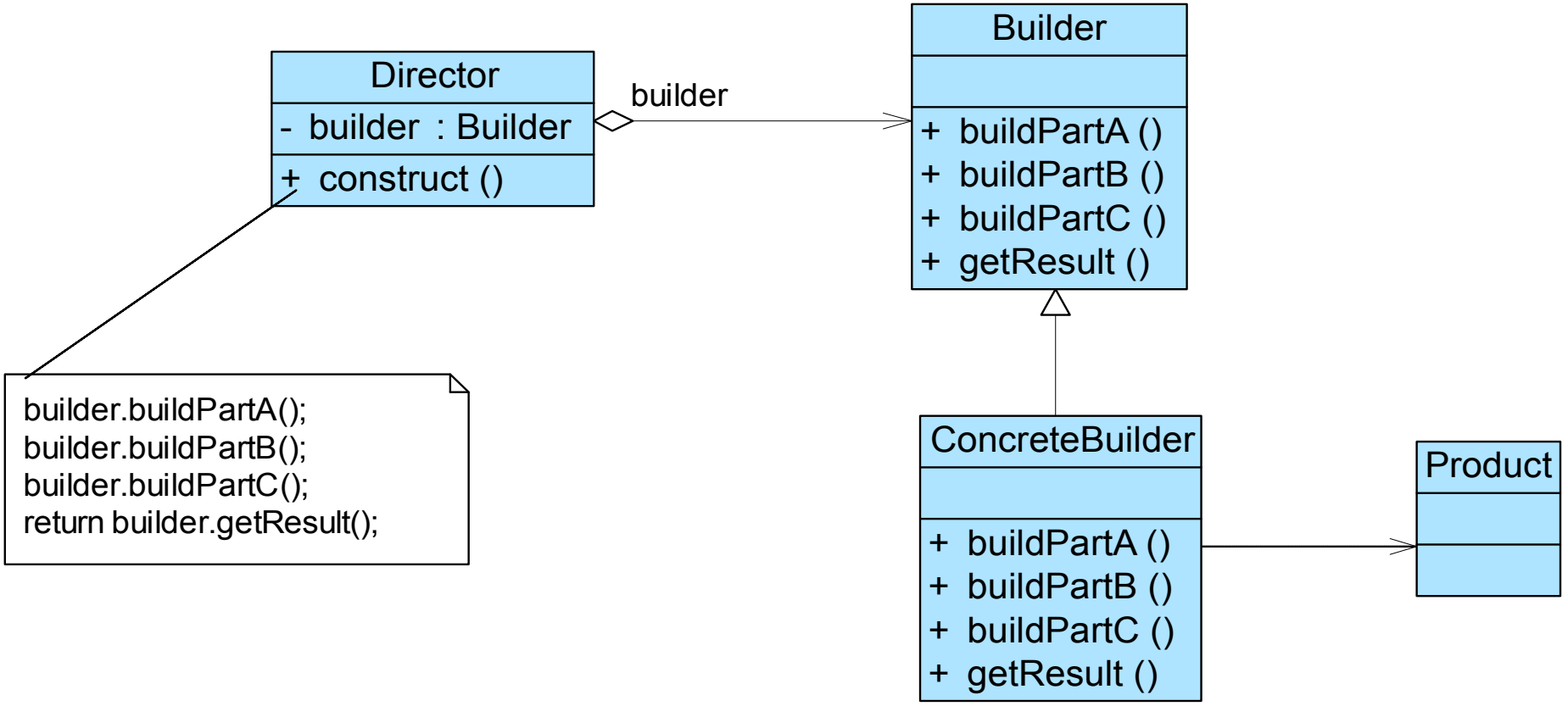
Builder Pattern: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

建造者模式定义

- 将客户端与包含多个部件的复杂对象的创建过程分离，客户端无须知道复杂对象的内部组成部分与装配方式，只需要知道所需建造者的类型即可
- 关注如何逐步创建一个复杂的对象，不同的建造者定义了不同的创建过程

建造者模式结构与分析

- **建造者模式结构包含如下角色：**
  - Builder：抽象建造者
  - Director：指挥者
  - ConcreteBuilder：具体建造者
  - Product：产品角色



建造者模式实现

- 典型的复杂对象类代码

```
public class Product {
        private String partA; //定义部件，部件可以是任意类型，包括值类型和引用类型
        private String partB;
        private String partC;

        //partA的Getter方法和Setter方法省略
        //partB的Getter方法和Setter方法省略
        //partC的Getter方法和Setter方法省略
}
```

建造者模式实现

- 典型的抽象建造者类代码

```
public abstract class Builder {
    //创建产品对象
    protected Product product=new Product();
    public abstract void buildPartA();
    public abstract void buildPartB();
    public abstract void buildPartC();

    //返回产品对象
    public Product getResult() {
        return product;
    }
}
```

建造者模式实现

- 典型的具体建造者类代码

```
public class ConcreteBuilder1 extends Builder{
    public void buildPartA() {
        product.setPartA("A1");
    }

    public void buildPartB() {
        product.setPartB("B1");
    }

    public void buildPartC() {
        product.setPartC("C1");
    }
}
```

建造者模式实现

- 典型的指挥者类代码

```
public class Director {
    private Builder builder;

    public Director(Builder builder) {
        this.builder=builder;
    }

    public void setBuilder(Builder builder) {
        this.builer=builer;
    }

    //产品构建与组装方法
    public Product construct() {
        builder.buildPartA();
        builder.buildPartB();
        builder.buildPartC();
        return builder.getResult();
    }
}
```

建造者模式实现

- 客户类代码片段

```
……
Builder builder = new ConcreteBuilder1(); //可通过配置文件实现
Director director = new Director(builder);
Product product = director.construct();
……
```

建造者模式实例

KFC套餐：实例说明

- 建造者模式可以用于描述KFC如何创建套餐：套餐是一个复杂对象，它一般包含主食（如汉堡、鸡肉卷等）和饮料（如果汁、可乐等）等组成部分，不同的套餐有不同的组成部分，而KFC的服务员可以根据顾客的要求，一步一步装配这些组成部分，构造一份完整的套餐，然后返回给顾客。

建造者模式实例与解析

建造者模式实例

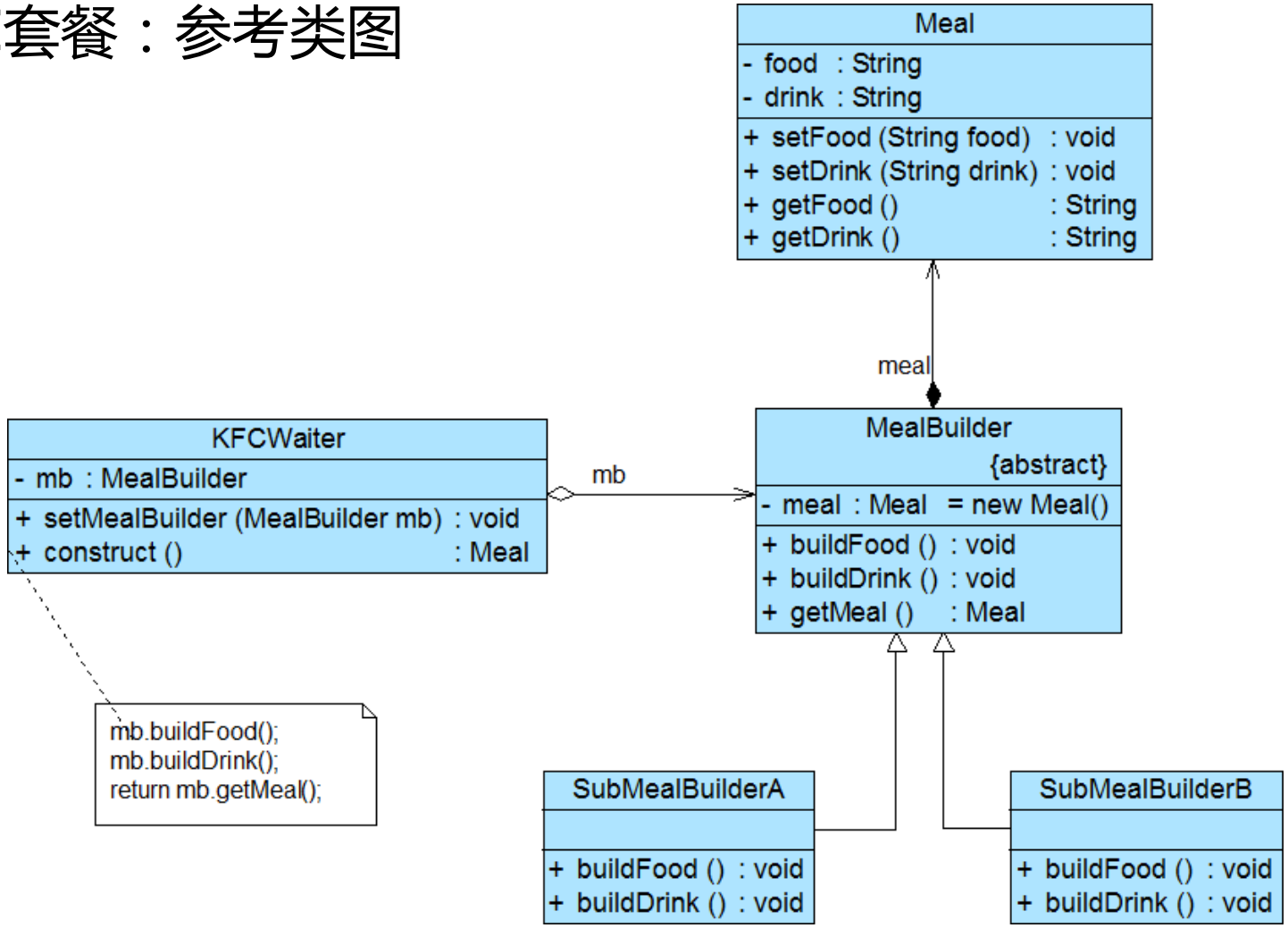
- KFC套餐：实例说明

KFC使用了建造者模式

建造者模式实例与解析

建造者模式实例

- KFC套餐：参考类图



建造者模式实例与解析

- **建造者模式实例**
  - **KFC套餐：参考代码**
- DesignPatterns之builder包

MealBuilder.java

```
1  package builder;
2
3  public abstract class MealBuilder
4  {
5      protected Meal meal=new Meal();
6      public abstract void buildFood();
7      public abstract void buildDrink();
8      public Meal getMeal()
9      {
10         return meal;
11     }
12 }
```

Meal.java

```
2
3  public class Meal
4  {
5      //food和drink是部件
6      private String food;
7      private String drink;
8
9      public void setFood(String food) {
10         this.food = food;
11     }
12
13     public void setDrink(String drink) {
14         this.drink = drink;
15     }
16
17     public String getFood() {
18         return (this.food);
19     }
20
21     public String getDrink() {
22         return (this.drink);
23     }
24 }
```

SubMealBuilderA.java

```java
package builder;

public class SubMealBuilderA extends MealBuilder
{
    public void buildFood()
    {
        meal.setFood("一个鸡腿堡");
    }
    public void buildDrink()
    {
        meal.setDrink("一杯可乐");
    }
}
```

SubMealBuilderB.java

```java
package builder;

public class SubMealBuilderB extends MealBuilder
{
    public void buildFood()
    {
        meal.setFood("一个鸡肉卷");
    }
    public void buildDrink()
    {
        meal.setDrink("一杯果汁");
    }
}
```

KFCWaiter.java

```java
package builder;

public class KFCWaiter
{
    private MealBuilder mb;
    public void setMealBuilder(MealBuilder mb)
    {
        this.mb=mb;
    }
    public Meal construct()
    {
        mb.buildFood();
        mb.buildDrink();
        return mb.getMeal();
    }
}
```

Builderconfig.xml

```xml
<?xml version="1.0"?>
<config>
    <className>builder.SubMealBuilderA</className>
</config>
```

XMLUtil.java

```
3  import javax.xml.parsers.*;
4  import org.w3c.dom.*;
5  import org.xml.sax.SAXException;
6  import java.io.*;
7  public class XMLUtil
8  {
9  //该方法用于从XML配置文件中提取具体类类名，并返回一个实例对象
10     public static Object getBean()
11     {
12         try
13         {
14             //创建文档对象
15             DocumentBuilderFactory dFactory = DocumentBuilderFactory.newInstance();
16             DocumentBuilder builder = dFactory.newDocumentBuilder();
17             Document doc;
18             doc = builder.parse(new File("Builderconfig.xml"));
19
20             //获取包含类名的文本节点
21             NodeList nl = doc.getElementsByTagName("className");
22             Node classNode=nl.item(0).getFirstChild();
23             String cName=classNode.getNodeValue();
24
25             //通过类名生成实例对象并将其返回
26             Class c=Class.forName(cName);
27             Object obj=c.newInstance();
28             return obj;
29           }
```

XMLUtil.java

```
30             catch(Exception e)
31             {
32                 e.printStackTrace();
33                 return null;
34             }
35         }
36 }
```

Client.java

```java
package builder;

public class Client
{
    public static void main(String args[])
    {
        //动态确定套餐种类
        MealBuilder mb=(MealBuilder)XMLUtil.getBean();
        //服务员是指挥者
        KFCWaiter waiter=new KFCWaiter();
        //服务员准备套餐
        waiter.setMealBuilder(mb);
        //客户获得套餐
        Meal meal=waiter.construct();

        System.out.println("套餐组成: ");
        System.out.println(meal.getFood());
        System.out.println(meal.getDrink());
    }
}
```

建造者模式效果与应用

- **建造者模式优点：**
  - 客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象
  - 每一个具体建造者都相对独立，与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者，扩展方便，符合开闭原则
  - 可以更加精细地控制产品的创建过程

建造者模式效果与应用

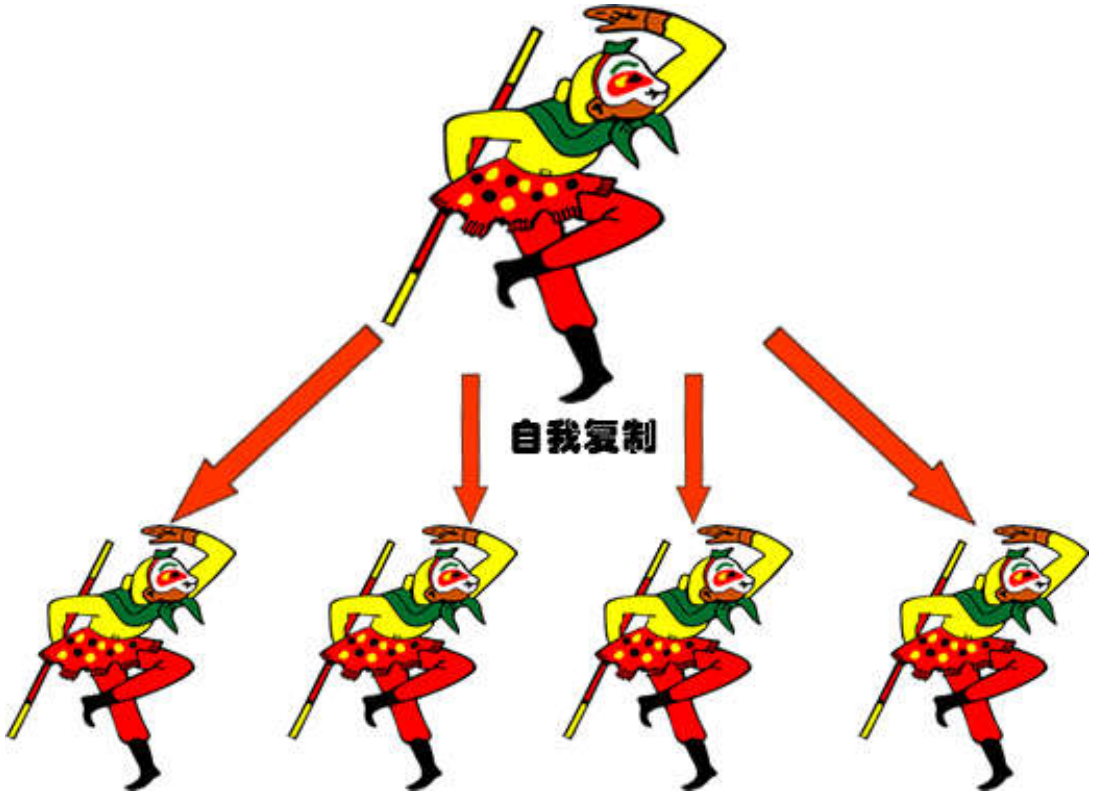
- **建造者模式缺点：**
  - 建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，不适合使用建造者模式，因此其使用范围受到一定的限制
  - 如果产品的内部变化复杂，可能会需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大，增加了系统的理解难度和运行成本

建造者模式效果与应用

- **在以下情况下可以使用建造者模式：**
  - 需要生成的产品对象有复杂的内部结构，这些产品对象通常包含多个成员变量
  - 需要生成的产品对象的属性相互依赖，需要指定其生成顺序
  - 对象的创建过程独立于创建该对象的类。在建造者模式中通过引入了指挥者类，将创建过程封装在指挥者类中，而不在建造者类和客户类中
  - 隔离复杂对象的创建和使用，并使得相同的创建过程可以创建不同的产品

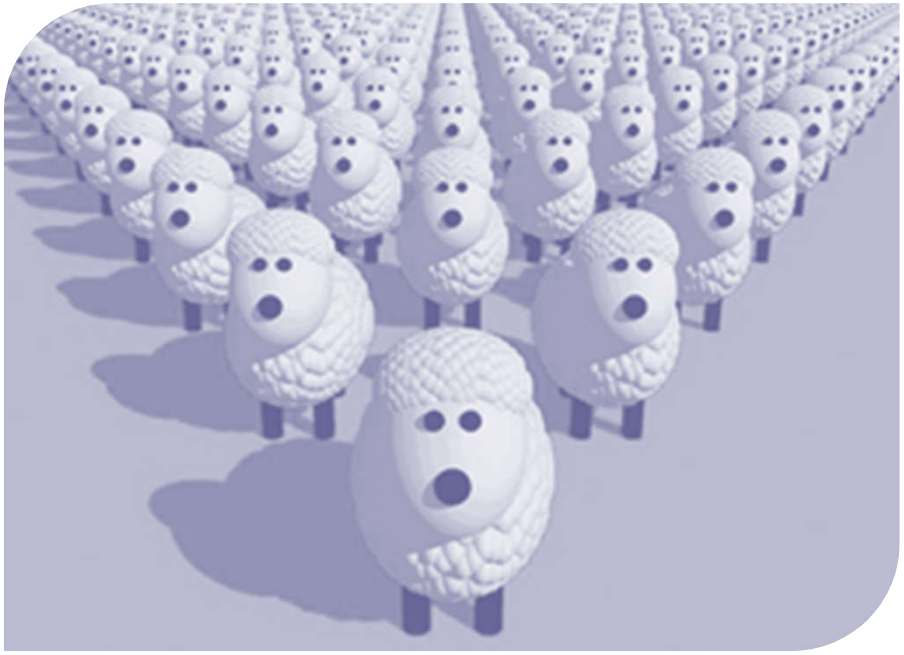
7.3 原型模式

- **原型模式动机**
  - 孙悟空“拔毛变小猴”



原型模式动机

- **分析**
  - 孙悟空：根据自己的形状复制（克隆）出多个身外身
  - 软件开发：通过复制一个原型对象得到多个与原型对象一模一样的新对象



原型模式

原型模式动机

- 复制一个对象，从而克隆出多个与原型对象一模一样的对象——**原型模式**
- 有些对象的创建过程较为复杂，而且需要频繁创建
- 通过给出一个原型对象来指明所要创建的对象的类型，然后用复制这个原型对象的办法创建出更多同类型的对象

原型模式定义

- 是对象创建型模式

原型模式：使用原型实例指定待创建对象的类型，并且通过复制这个原型来创建新的对象。

Prototype Pattern: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

- 原型模式允许通过一个原型对象创建一个或多个同类型的其他对象，而无须知道任何创建的细节

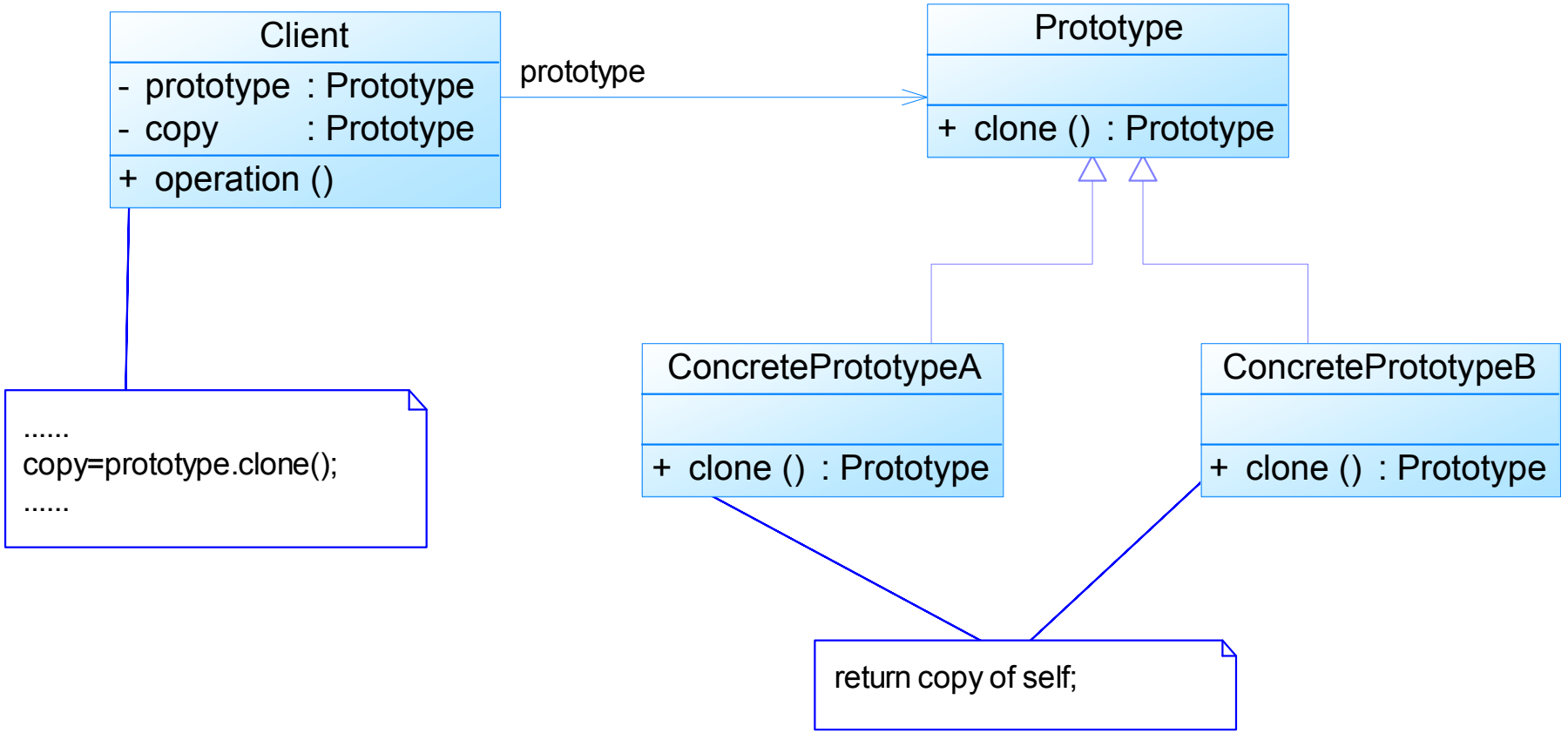
原型模式工作原理

- 工作原理：将一个原型对象传给要发动创建的对象（即客户端对象），这个要发动创建的对象通过请求原型对象复制自己来实现创建过程
- 创建新对象（也称为克隆对象）的工厂就是原型类自身，工厂方法由负责复制原型对象的克隆方法来实现
- 通过克隆方法所创建的对象是全新的对象，它们在内存中拥有新的地址，每一个克隆对象都是独立的

原型模式结构与分析

原型模式结构

- 原型模式包含三个角色：
- Prototype：抽象原型类，ConcretePrototype：具体原型类，Client：客户类



原型模式分析

- 所有的Java类都继承自java.lang.Object，而Object类提供一个clone()方法，可以将一个Java对象复制一份
- 在Java中可以直接使用Object提供的clone()方法来实现对象的克隆（浅克隆）
- 能够实现克隆的Java类必须实现一个标识接口Cloneable，表示这个Java类支持复制
- 如果一个类没有实现这个接口但是调用了clone()方法，Java编译器将抛出一个CloneNotSupportedException异常

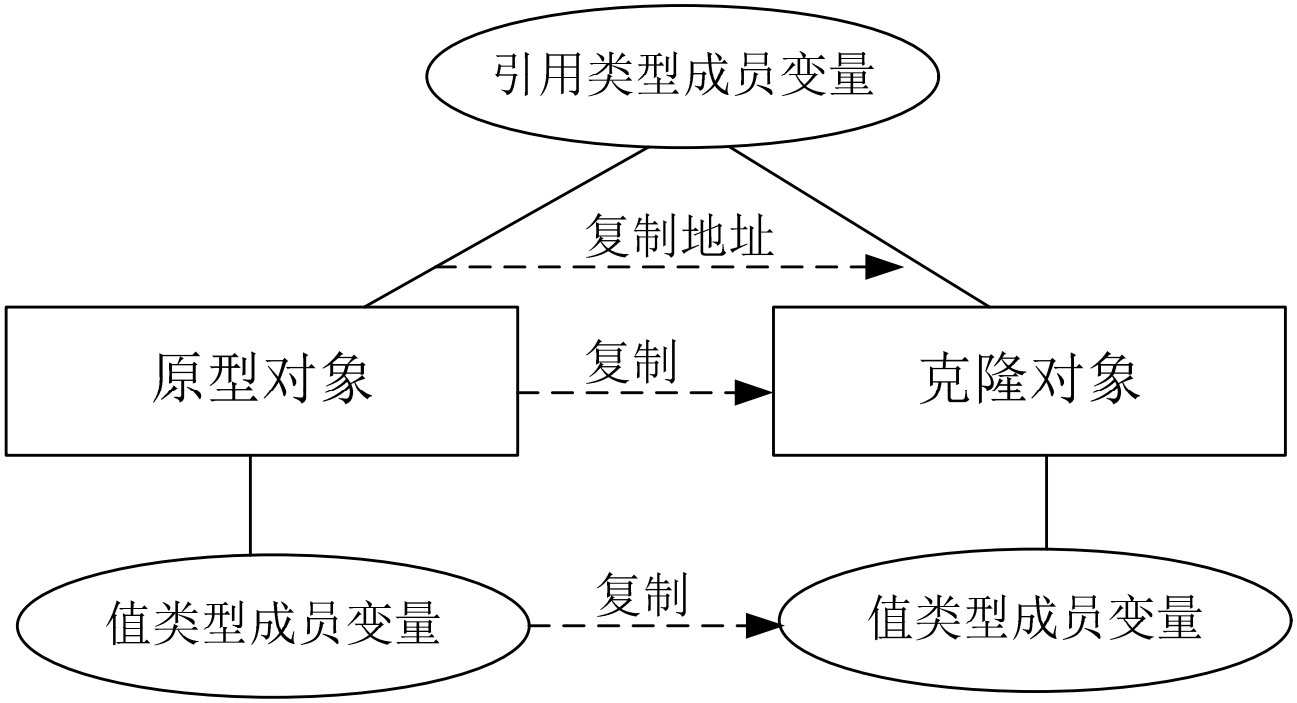
原型模式分析

- 示例代码：

```
public class PrototypeDemo implements Cloneable
{
    ……
        public Object clone()
        {
                Object object = null;
                try {
                        object = super.clone();
                } catch (CloneNotSupportedException exception) {
                        System.err.println("Not support cloneable");
                }
                return object;
    }
    ……
}
```

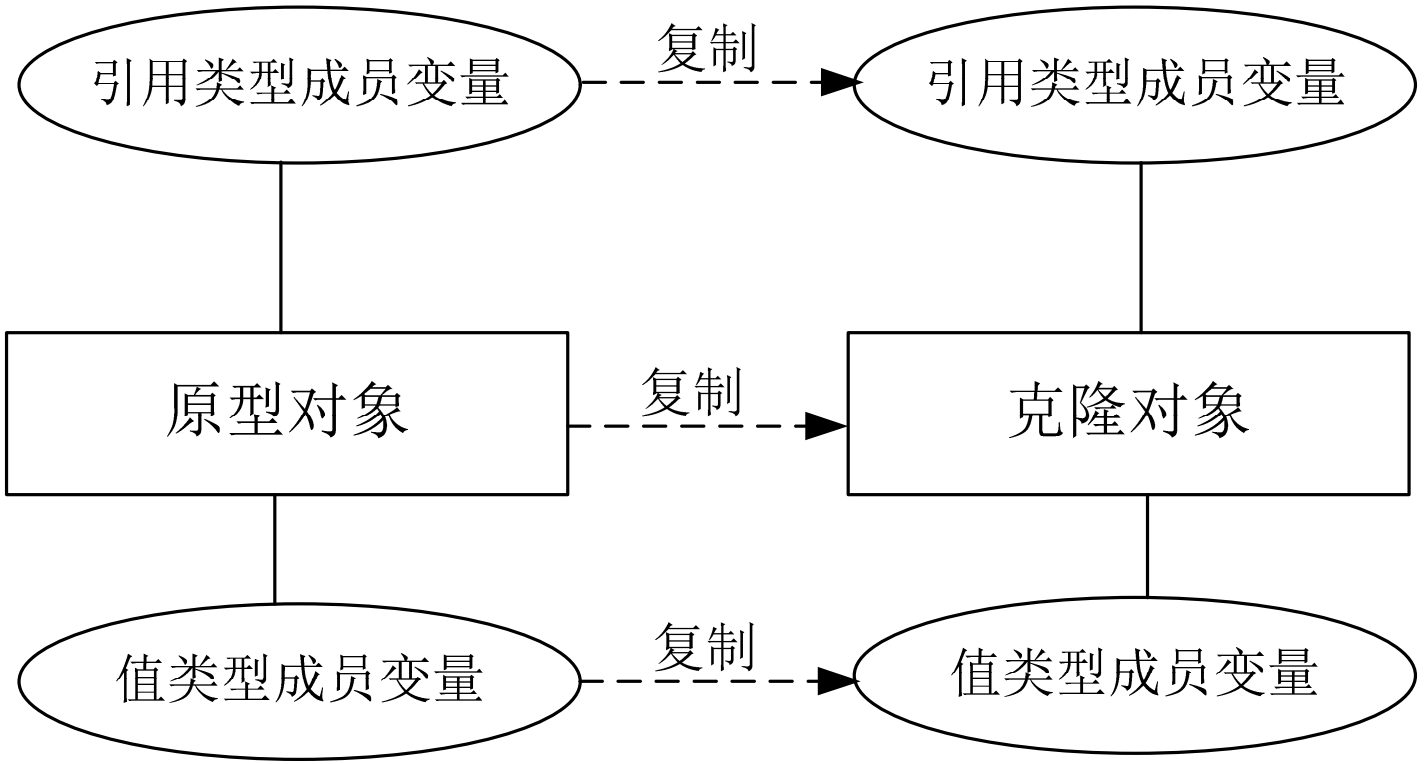
原型模式分析

- 浅克隆(Shallow Clone)：当原型对象被复制时，只复制它本身和其中包含的值类型的成员变量，而引用类型的成员变量并没有复制



原型模式分析

- 深克隆(Deep Clone)：除了对象本身被复制外，对象所包含的所有成员变量也将被复制



原型模式实例

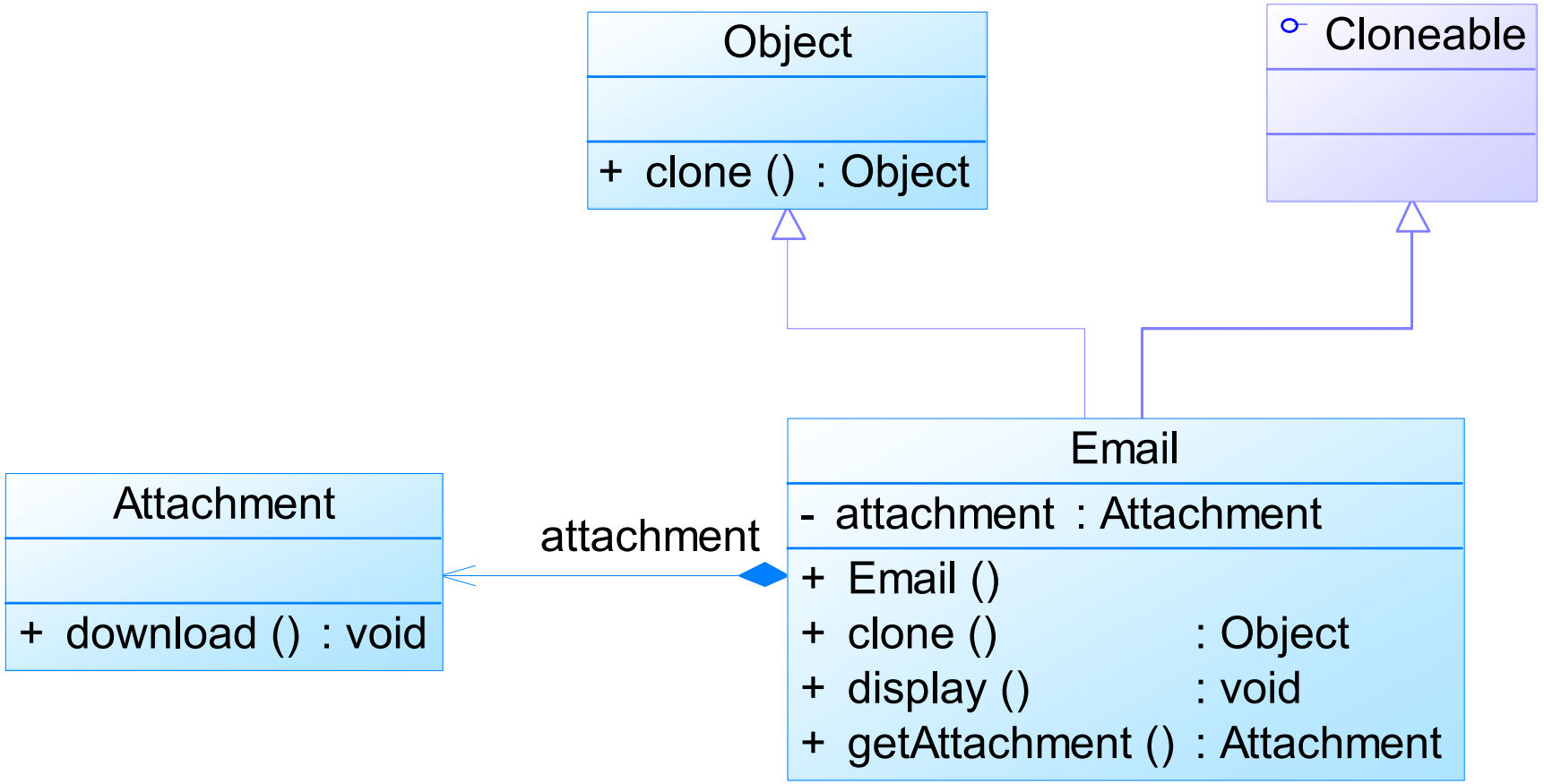
邮件复制（浅克隆）：实例说明

- 由于邮件对象包含的内容较多（如发送者、接收者、标题、内容、日期、附件等），某系统中现需要提供一个邮件复制功能，对于已经创建好的邮件对象，可以通过复制的方式创建一个新的邮件对象，如果需要改变某部分内容，无须修改原始的邮件对象，只需要修改复制后得到的邮件对象即可。使用原型模式设计该系统。
- 在本实例中使用浅克隆实现邮件复制，即复制邮件(Email)的同时不复制附件(Attachment)。

原型模式实例与解析

原型模式实例

邮件复制（浅克隆）：参考类图



原型模式实例与解析

- **原型模式实例**
  - **邮件复制（浅克隆）：参考代码**
- DesignPatterns之prototype.shallow包

Email.java

```
3  public class Email implements Cloneable
4  {
5      private Attachment attachment=null;
6
7      public Email()
8      {
9          this.attachment=new Attachment();
10     }
11
12     public Object clone()
13     {
14         Email clone=null;
15         try
16         {
17             clone=(Email)super.clone();
18         }
19         catch(CloneNotSupportedException e)
20         {
21             System.out.println("Clone failure!");
22         }
23         return clone;
24     }
```

Email.java

```
25
26     public Attachment getAttachment()
27     {
28         return this.attachment;
29     }
30
31     public void display()
32     {
33         System.out.println("查看邮件");
34     }
35
36 }
```

Attachment.java

```java
package prototype.shallow;

public class Attachment
{
    public void download()
    {
        System.out.println("下载附件");
    }
}
```

Client.java

```java
public class Client
{
    public static void main(String a[])
    {
        Email email,copyEmail;

        email=new Email();

        copyEmail=(Email)email.clone();

        System.out.println("email==copyEmail?");
        System.out.println(email==copyEmail);

        System.out.println("email.getAttachment==copyEmail.getAttachment?");
        System.out.println(email.getAttachment()==copyEmail.getAttachment());
    }
}
```

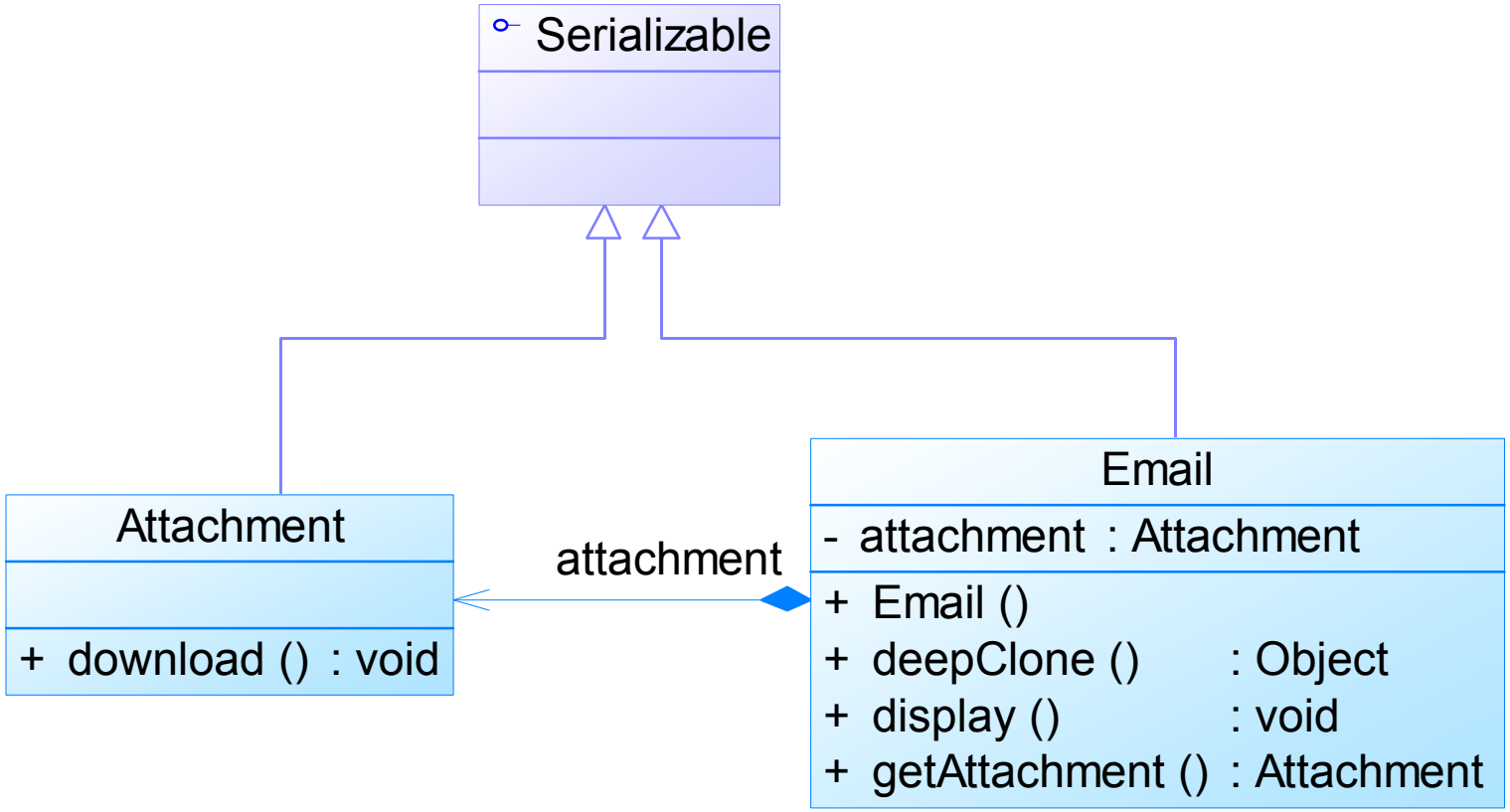
原型模式实例与解析

- **原型模式实例**
  - **邮件复制（深克隆）：实例说明**
    - 使用深克隆实现邮件复制，即复制邮件的同时复制附件。
  - **邮件复制（深克隆）：参考代码**
- DesignPatterns之prototype. deep包

原型模式实例与解析

原型模式实例

邮件复制（深克隆）：参考类图



Email.java

```java
import java.io.*;
public class Email implements Serializable
{
    private Attachment attachment=null;

    public Email()
    {
        this.attachment=new Attachment();
    }

    public Object deepClone() throws IOException, ClassNotFoundException, OptionalDataException
    {
        //将对象写入流中
        ByteArrayOutputStream bao=new ByteArrayOutputStream();
        ObjectOutputStream oos=new ObjectOutputStream(bao);
        oos.writeObject(this);

        //将对象从流中取出
        ByteArrayInputStream bis=new ByteArrayInputStream(bao.toByteArray());
        ObjectInputStream ois=new ObjectInputStream(bis);
        return(ois.readObject());
    }
```

Email.java

```java
    public Attachment getAttachment()
    {
        return this.attachment;
    }

    public void display()
    {
        System.out.println("查看邮件");
    }

}
```

Attachment.java

```
1  package prototype.deep;
2
3  import java.io.*;
4
5  public class Attachment implements Serializable
6  {
7      public void download()
8      {
9          System.out.println("下载附件");
10     }
11 }
```

Client.java

```
3  public class Client
4  {
5      public static void main(String a[])
6      {
7          Email email,copyEmail=null;
8
9          email=new Email();
10
11         try{
12             copyEmail=(Email)email.deepClone();
13         }
14         catch(Exception e)
15         {
16             e.printStackTrace();
17         }
18
19
20         System.out.println("email==copyEmail?");
21         System.out.println(email==copyEmail);
22
23         System.out.println("email.getAttachment==copyEmail.getAttachment?");
24         System.out.println(email.getAttachment()==copyEmail.getAttachment());
25     }
26 }
```

原型模式效果与应用

- **原型模式优点：**
  - 简化对象的创建过程，通过复制一个已有实例可以提高新实例的创建效率
  - 扩展性较好
  - 简化创建结构，原型模式中产品的复制是通过封装在原型类中的克隆方法实现的，无须专门的工厂类来创建产品
  - 可以使用深克隆的方式保存对象的状态，以便在需要的时候使用，可辅助实现撤销操作

原型模式效果与应用

- **原型模式缺点：**
  - 需要为每一个类配备一个克隆方法，而且该克隆方法位于一个类的内部，当对已有的类进行改造时，需要修改源代码，违背了开闭原则
  - 在实现深克隆时需要编写较为复杂的代码，而且当对象之间存在多重的嵌套引用时，为了实现深克隆，每一层对象对应的类都必须支持深克隆，实现起来可能会比较麻烦

原型模式效果与应用

- **在以下情况下可以使用原型模式：**
  - 创建新对象成本较大，新对象可以通过复制已有对象来获得，如果是相似对象，则可以对其成员变量稍作修改
  - 系统要保存对象的状态，而对象的状态变化很小
  - 需要避免使用分层次的工厂类来创建分层次的对象

谢谢