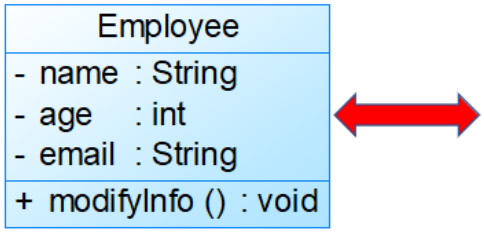
第1章 统一建模语言基础知识：

UML：一种通用的可视化建模语言

类的UML图示

✓ 在UML类图中，类一般由三部分组成：

- 第一部分是类名：每个类都必须有一个名字，类名是一个字符串
- 按照Java语言的命名规范，类名中每一个单词的首字母均大写

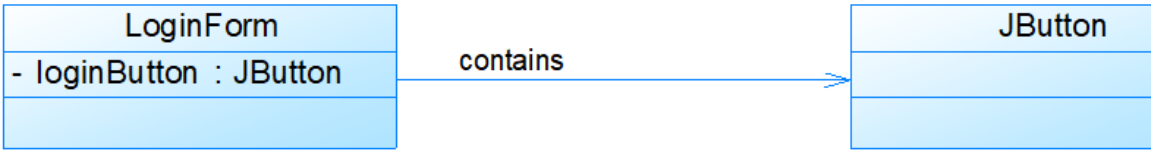



```
public class Employee {
    private String name;
    private int age;
    private String email;

    public void modifyInfo() {
        ......
    }
}
```

关联关系

关联(Association)关系是类与类之间最常用的一种关系，它是一种结构化关系，用于表示一类对象与另一类对象之间有联系

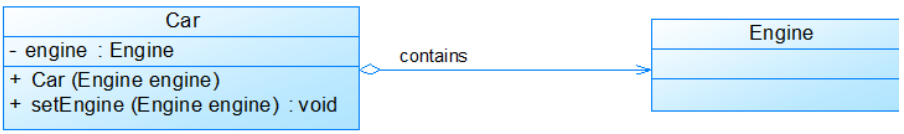
✓ 关联关系




```
public class LoginForm {
    private JButton loginButton; //定义为成员变量
    ......
}
public class JButton {
    ......
}
```

✓ 关联关系

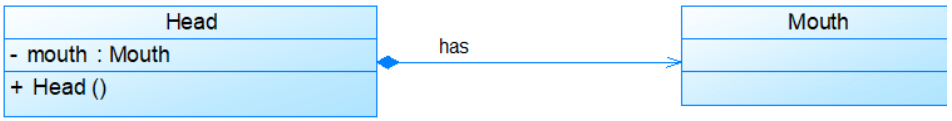
- 聚合关联




```
public class Car {
    private Engine engine;
    public Car(Engine engine) {    //构造注入
        this.engine = engine;
    }

    public void setEngine(Engine engine) {    //设值注入
        this.engine = engine;
    }
    ……
}

public class Engine {
    ……
}
```

关联关系

- 组合关联




```
public class Head {
    private Mouth mouth;
    public Head() {
        mouth = new Mouth();  //实例化成员类
    }
    ……
}

public class Mouth {
    ……
}
```

依赖关系

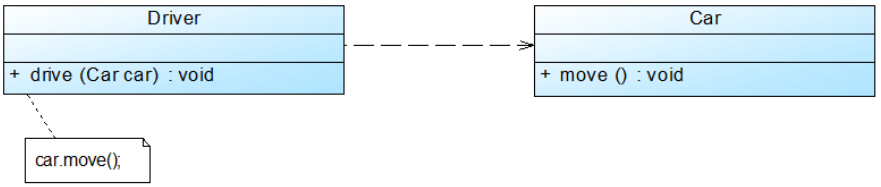
在系统实现阶段，依赖关系通常通过三种方式来实现：

将一个类的对象作为另一个类中方法的参数

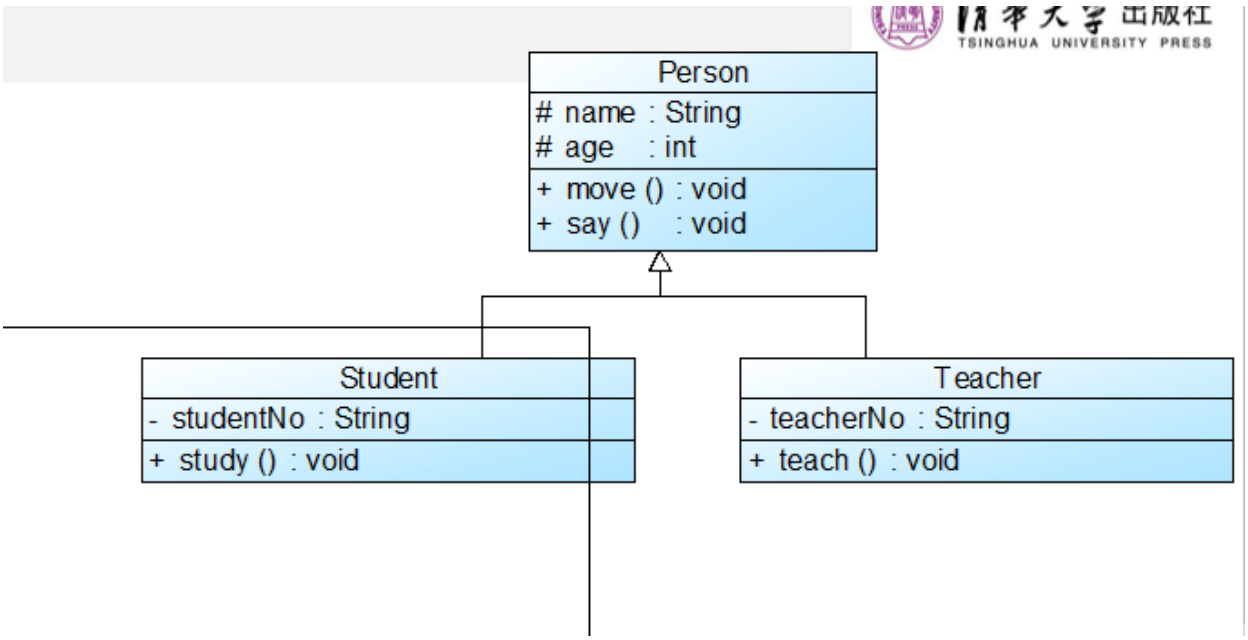
在一个类的方法中将另一个类的对象作为其局部变量

在一个类的方法中调用另一个类的静态方法

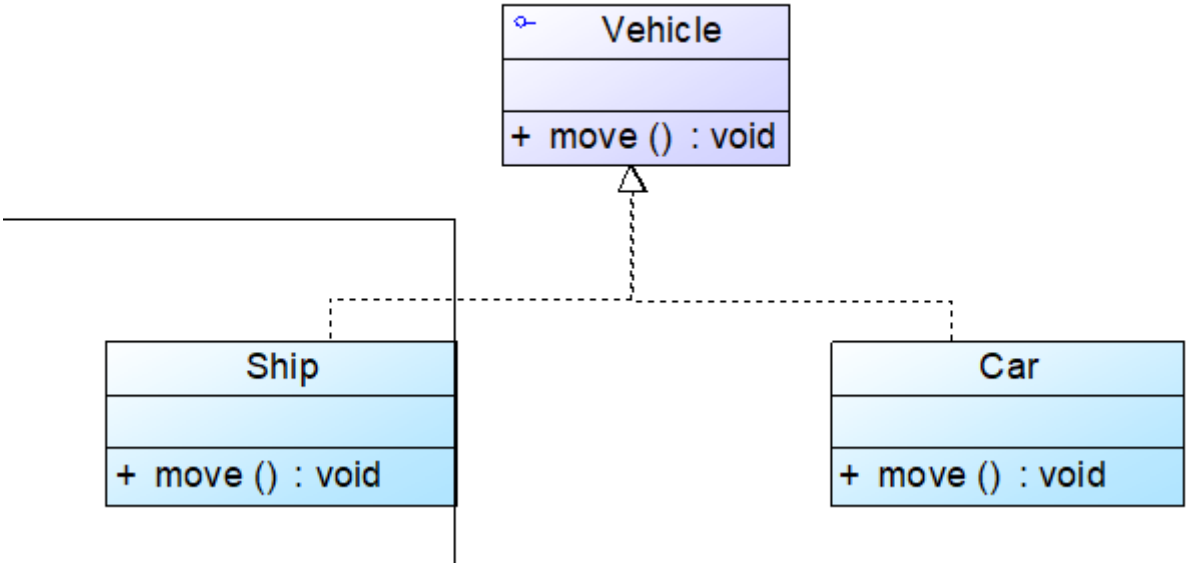
- **类之间的关系**
  - ✓ 依赖关系




```
public class Driver {
   public void drive(Car car)
   {
      car.move();
   }
   ……
}
public class Car {
   public void move() {
      ……
   }
   ……
}
```



继承



实现

第2章 面向对象设计原则

可维护性(Maintainability)：指软件能够被理解、改正、适应及扩展的难易程度

可复用性(Reusability)：指软件能够被重复使用的难易程度

第3章 设计模式概述

范围\目的	创建型模式	结构型模式	行为型模式
类模式	工厂方法模式	（类）适配器模式	解释器模式 模板方法模式
对象模式	抽象工厂模式 建造者模式 原型模式 单例模式	（对象）适配器模式 桥接模式 组合模式 装饰模式 外观模式 享元模式 代理模式	职责链模式 命令模式 迭代器模式 中介者模式 备忘录模式 观察者模式 状态模式 策略模式 访问者模式

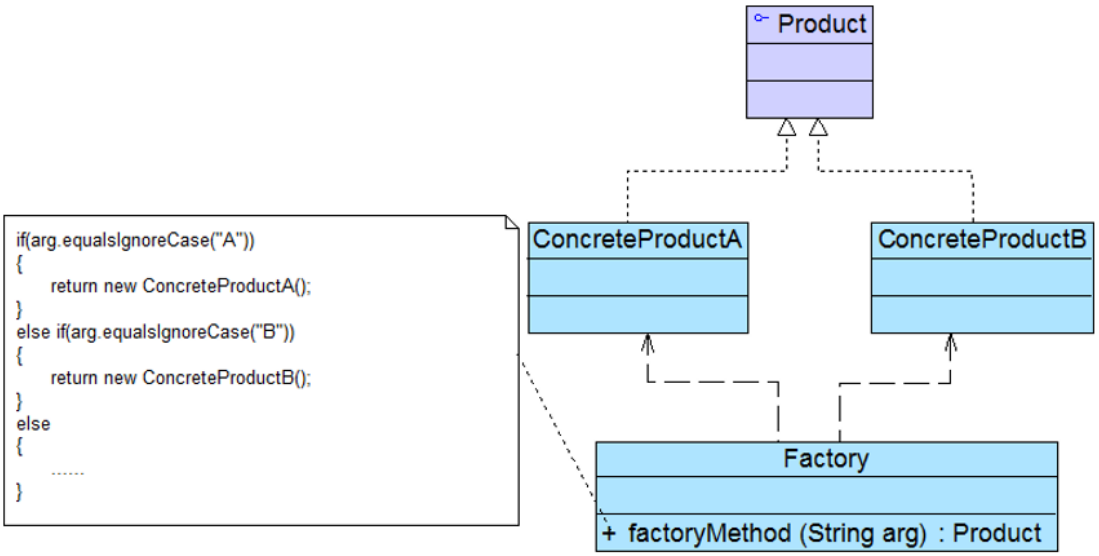
第4章 简单工厂模式

简单工厂模式(Simple Factory Pattern)：又称为静态工厂方法(Static Factory Method)模式，它属于类创建型模式

在简单工厂模式中，可以根据参数的不同返回不同类的实例

简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类

模式结构



简单工厂模式包含如下角色：

Factory：工厂角色

Product：抽象产品角色

ConcreteProduct：具体产品角色

简单工厂模式优点：

1）实现了对象创建和使用的分离

2）客户端无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可

3）通过引入配置文件，可以在不修改任何客户端代码的情况下更换和增加新的具体产品类，在一定程度上提高了系统的灵活性

简单工厂模式缺点（不符合开闭原则）：

1）工厂类集中了所有产品的创建逻辑，职责过重，一旦不能正常工作，整个系统都要受到影响

2）增加系统中类的个数（引入了新的工厂类），增加了系统的复杂度和理解难度

3）系统扩展困难，一旦添加新产品不得不修改工厂逻辑

4）由于使用了静态工厂方法，造成工厂角色无法形成基于继承的等级结构，工厂类不能得到很好地扩展

在以下情况下可以使用简单工厂模式：

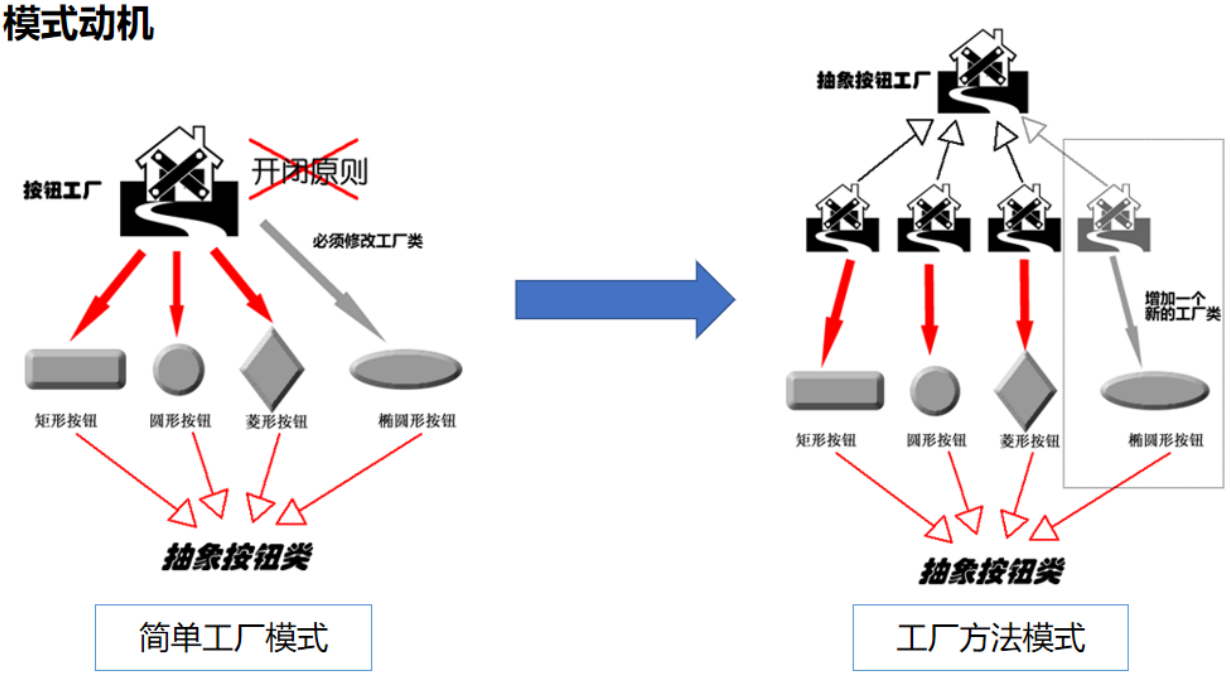
1）工厂类负责创建的对象比较少：由于创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂

2）客户端只知道传入工厂类的参数，对于如何创建对象不关心：客户端既不需要关心创建细节，甚至连类名都不需要记住，只需要知道类型所对应的参数

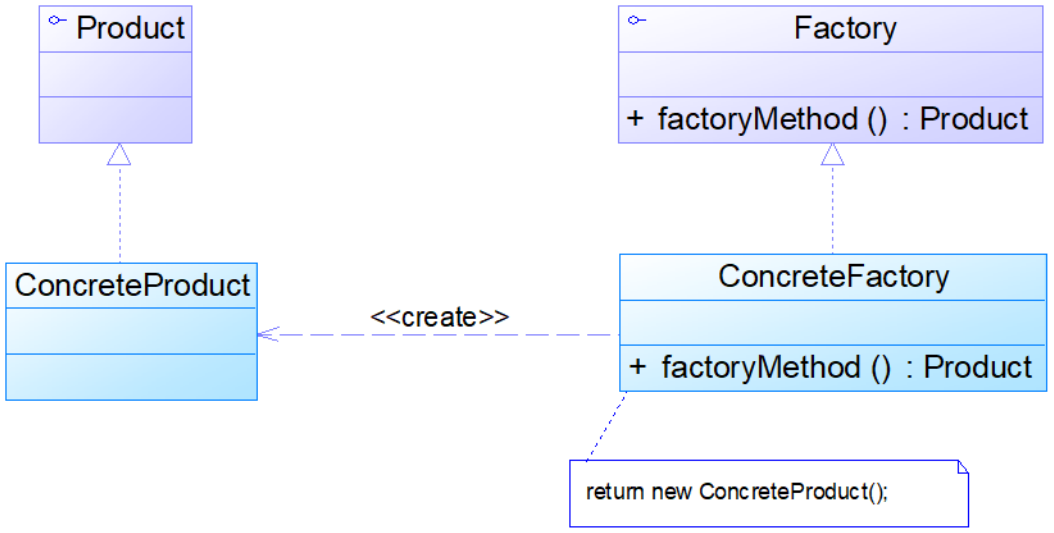
第5章 工厂方法模式 ★

面向单一产品等级结构

模式动机



模式结构



工厂方法模式包含如下角色：

Product：抽象产品

ConcreteProduct：具体产品

Factory：抽象工厂

ConcreteFactory：具体工厂

模式分析

✓ Java反射(Java Reflection)机制的应用

```
//创建一个字符串类型的对象
Class c = Class.forName("String");
Object obj = c.newInstance();
return obj;
```

工厂方法模式优点：

1）工厂方法用来创建客户所需要的产品，同时还向客户隐藏了哪种具体产品类将被实例化这一细节

2）能够让工厂自主确定创建何种产品对象，而如何创建这个对象的细节则完全封装在具体工厂内部

3）在系统中加入新产品时，完全符合开闭原则

工厂方法模式缺点：

1）系统中类的个数将成对增加，在一定程度上增加了系统的复杂度，会给系统带来一些额外的开销

2）增加了系统的抽象性和理解难度

在以下情况下可以使用工厂方法模式：

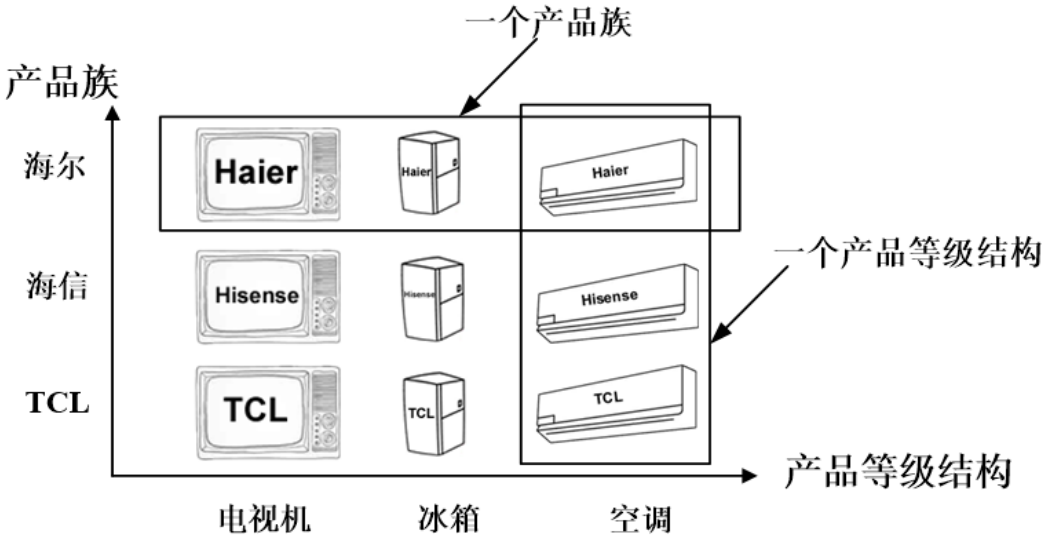
1）客户端不知道它所需要的对象的类（客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体产品对象由具体工厂类创建）

2）抽象工厂类通过其子类来指定创建哪个对象

第6章 抽象工厂模式 ★

面向多个产品等级结构

模式动机

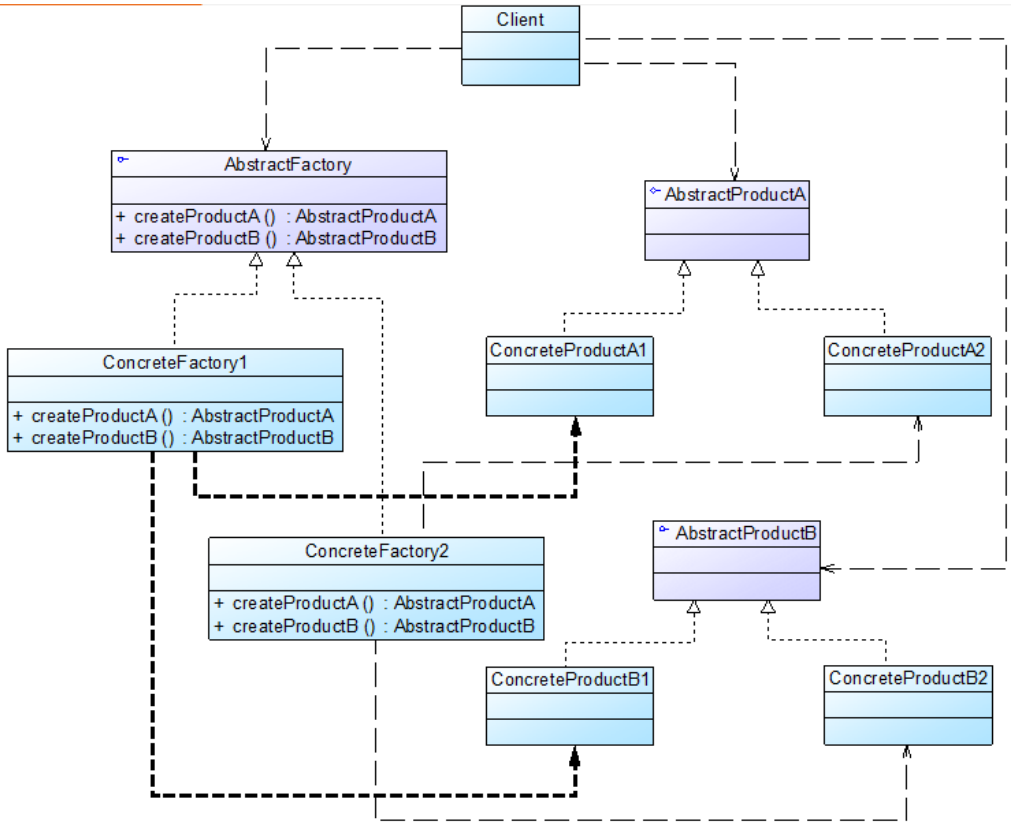


产品族与产品等级结构示意图

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。

模式结构



抽象工厂模式包含如下角色：

AbstractFactory：抽象工厂

ConcreteFactory：具体工厂

AbstractProduct：抽象产品

ConcreteProduct：具体产品

抽象工厂模式优点：

1）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

2）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

3）增加新的产品族很方便，无须修改已有系统，符合开闭原则

抽象工厂模式缺点：

1）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

在以下情况下可以使用抽象工厂模式：

1）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

2）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

3）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

4）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

第9章 单例模式

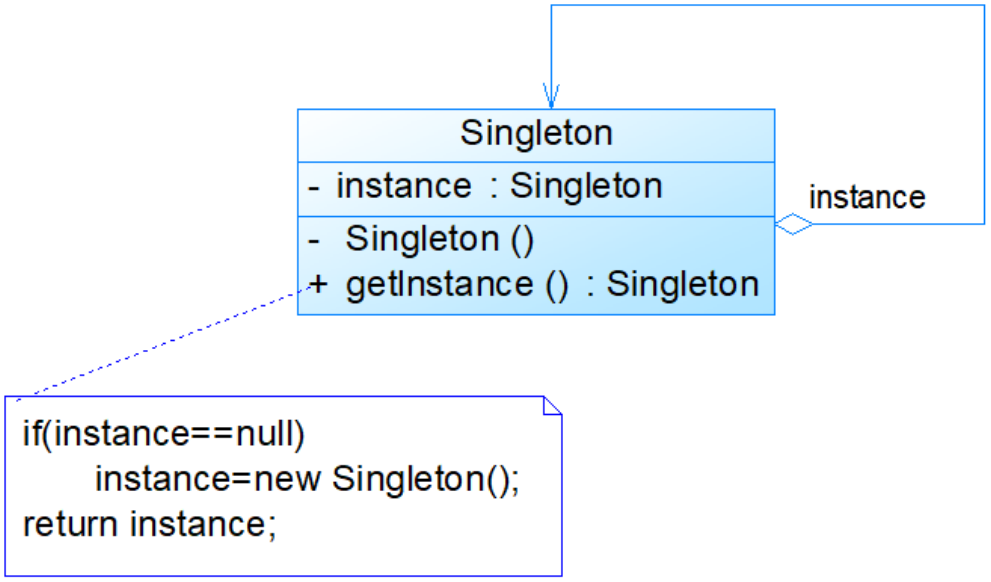
模式动机

- ✓ 如何确保一个类只有一个实例并且这个实例易于被访问?
- ✓ 让类自身负责创建和保存它的唯一实例，并保证不能创建其他实例，并且提供一个访问该实例的方法

模式定义

- ✓ 单例模式(Singleton Pattern)：确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。
- ✓ 单例模式的要点有三个：
  - 某个类只能有一个实例
  - 必须自行创建这个实例
  - 必须自行向整个系统提供这个实例
- ✓ 单例模式是一种对象创建型模式

模式结构



单例模式包含如下角色：

Singleton：单例

- ✓ 饿汉式单例类(Eager Singleton)

```
public class EagerSingleton {
   private static final EagerSingleton instance = new EagerSingleton();
   private EagerSingleton() { }

   public static EagerSingleton getInstance() {
      return instance;
   }
}
```

模式分析

- ✓ 懒汉式单例类与双重检查锁定
  - 延迟加载

```
public class LazySingleton {
    private static LazySingleton instance = null;

    private LazySingleton() { }

    public static LazySingleton getInstance() {
        if (instance == null) {
            instance = new LazySingleton();
        }
        return instance;
    }
}
```

多个线程同时访问将导致创建多个单例对象！怎么办？

需要较长时间

模式分析

- ✓ 懒汉式单例类与双重检查锁定
  - 延迟加载

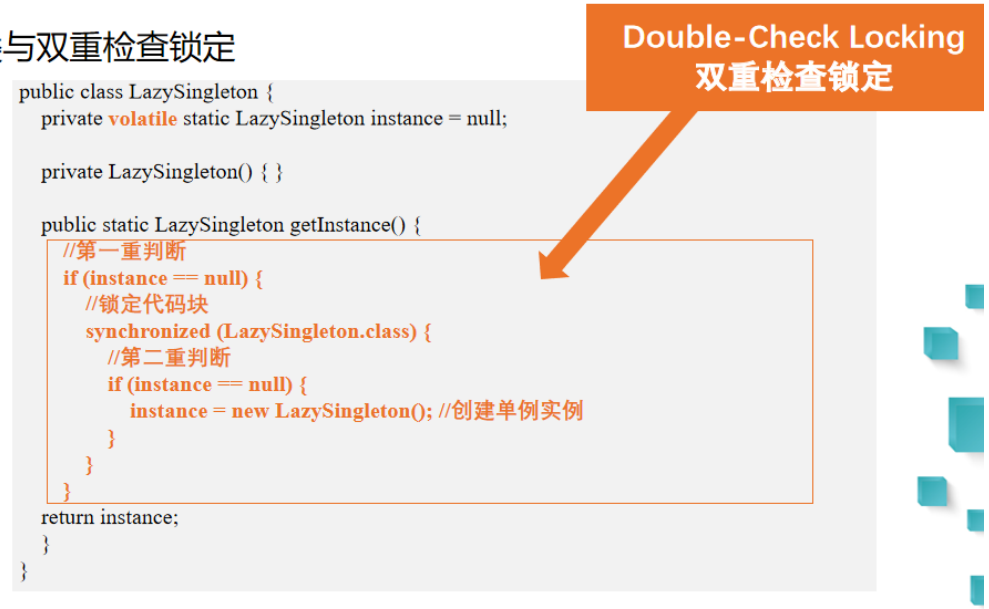
```
public class LazySingleton {
    private static LazySingleton instance = null;

    private LazySingleton() { }

    synchronized public static LazySingleton getInstance() {
        if (instance == null) {
            instance = new LazySingleton();
        }
        return instance;
    }
}
```

锁方法

模式分析

✓ 懒汉式单例类与双重检查锁定

- 延迟加载



private volatile static 防止创建对象时重排序

单例模式优点：

1）提供了对唯一实例的受控访问

2）可以节约系统资源，提高系统的性能

3）允许可变数目的实例（多例类）

单例模式缺点：

1）扩展困难（缺少抽象层）

2）单例类的职责过重

3）由于自动垃圾回收机制，可能会导致共享的单例对象的状态丢失

在以下情况下可以使用单例模式：

1）系统只需要一个实例对象，或者因为资源消耗太大而只允许创建一个对象

2）客户调用类的单个实例只允许使用一个公共访问点，除了该公共访问点，不能通过其他途径访问该实例

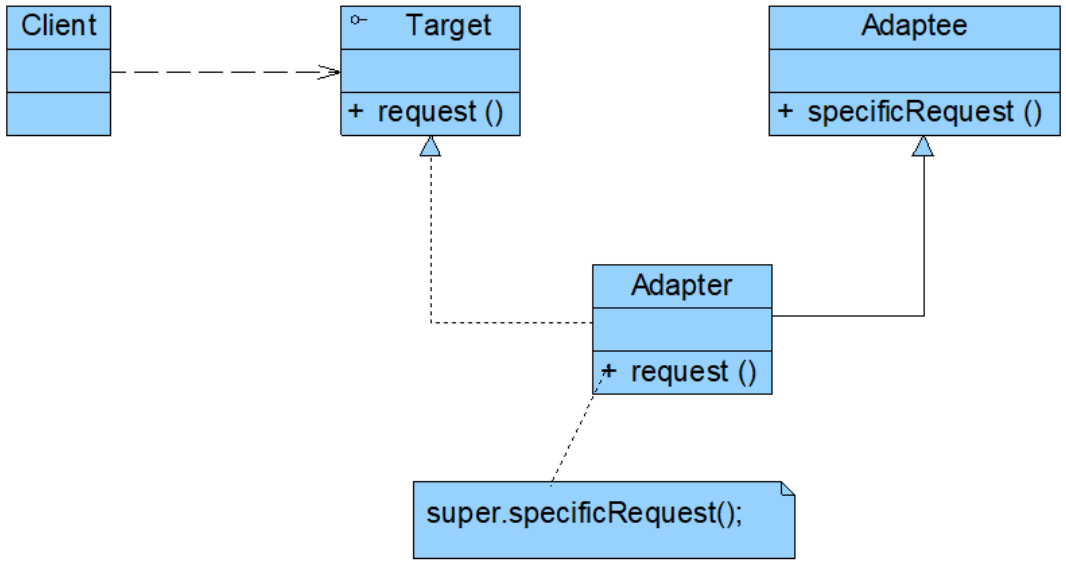
第10章 适配器模式

模式定义

- ✓ 适配器模式(Adapter Pattern)：将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作。
- ✓ 适配器模式既可以作为类结构型模式，也可以作为对象结构型模式
- ✓ 定义中所提及的接口是指广义的接口，它可以表示一个方法或者方法的集合

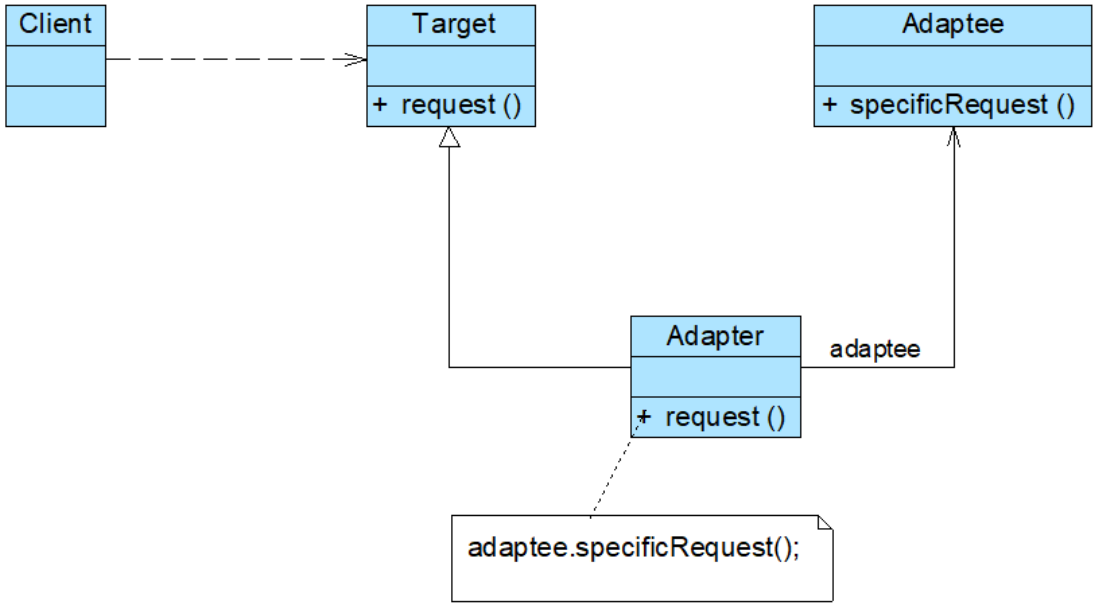
模式结构

- ✓ 类适配器



模式结构

- ✓ 对象适配器



适配器模式包含如下角色：

Target：目标抽象类

Adapter：适配器类

Adaptee：适配者类

适配器模式优点：

1）将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，无须修改原有结构

2）增加了类的透明性和复用性，提高了适配者的复用性，同一个适配者类可以在多个不同的系统中复用

3）灵活性和扩展性非常好

适配器模式缺点：

类适配器模式：

(1) 一次最多只能适配一个适配者类，不能同时适配多个适配者

(2) 适配者类不能为最终类

(3) 目标抽象类只能为接口，不能为类

对象适配器模式：在适配器中置换适配者类的某些方法比较麻烦

在以下情况下可以使用适配器模式：

1）系统需要使用一些现有的类，而这些类的接口不符合系统的需要，甚至没有这些类的源代码

2）创建一个可以重复使用的类，用于和一些彼此之间没有太大关联的类，包括一些可能在将来引进的类一起工作

第12章 组合模式

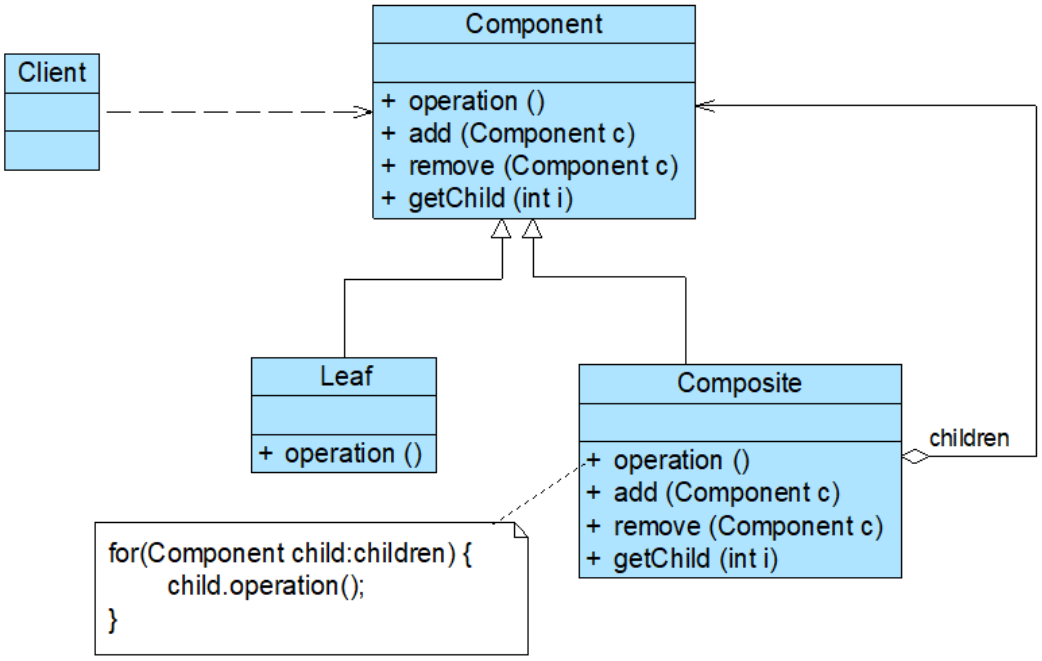
模式动机

- ✓ 在树形目录结构中，包含文件和文件夹两类不同的元素
  - 在文件夹中可以包含文件，还可以继续包含子文件夹
  - 在文件中不能再包含子文件或者子文件夹
- ✓ 文件夹 ←→ 容器(Container)
- ✓ 文件 ← → 叶子(Leaf)
- ✓ 如何将容器对象和叶子对象进行递归组合，使得用户在使用时无须对它们进行区分，可以一致地对待容器对象和叶子对象？ → **组合模式**

模式定义

- ✓ 组合模式(Composite Pattern)：组合多个对象形成树形结构以表示“部分-整体”的结构层次。组合模式对单个对象（即叶子对象）和组合对象（即容器对象）的使用具有一致性。
- ✓ 对象结构型模式
- ✓ 将对象组织到树形结构中，可以用来描述整体与部分的关系

模式结构



组合模式包含如下角色：

Component：抽象构件

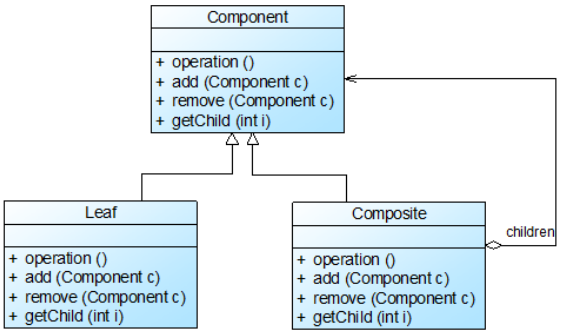
Leaf：叶子构件

Composite：容器构件

模式分析

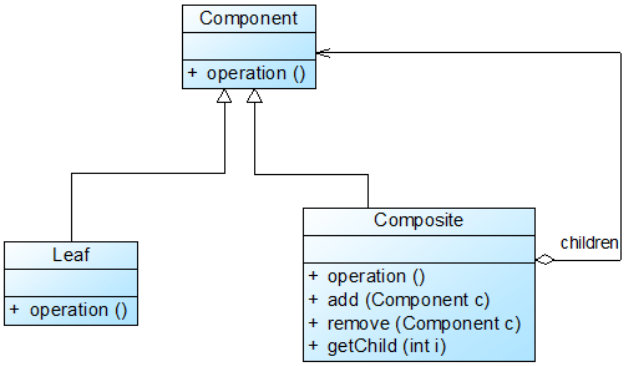
✓ 透明组合模式

- 抽象构件Component中声明了所有用于管理成员对象的方法，包括add()、remove()，以及getChild()等方法
- 在客户端看来，叶子对象与容器对象所提供的方法是一致的，客户端可以一致地对待所有的对象
- 缺点是不够安全，因为叶子对象和容器对象在本质上是有区别的



安全组合模式

- 抽象构件Component中没有声明任何用于管理成员对象的方法，而是在Composite类中声明并实现这些方法
- 对于叶子对象，客户端不可能调用到这些方法
- 缺点是不够透明，客户端不能完全针对抽象编程，必须有区别地对待叶子构件和容器构件



组合模式优点：

1）可以清楚地定义分层次的复杂对象，表示对象的全部或部分层次，让客户端忽略了层次的差异，方便对整个层次结构进行控制

2）客户端可以一致地使用一个组合结构或其中单个对象，不必关心处理的是单个对象还是整个组合结构，简化了客户端代码

3）增加新的容器构件和叶子构件都很方便，符合开闭原则

4）为树形结构的面向对象实现提供了一种灵活的解决方案

组合模式缺点：

1）在增加新构件时很难对容器中的构件类型进行限制

在以下情况下可以使用组合模式：

1）在具有整体和部分的层次结构中，希望通过一种方式忽略整体与部分的差异，客户端可以一致地对待它们

2）在一个使用面向对象语言开发的系统中需要处理一个树形结构

3）在一个系统中能够分离出叶子对象和容器对象，而且它们的类型不固定，需要增加一些新的类型

第13章 装饰者模式

模式动机

- ✓ 可以在不改变一个对象本身功能的基础上给对象增加额外的新行为
- ✓ 是一种用于替代继承的技术，它通过一种无须定义子类的方式给对象动态增加职责，使用对象之间的关联关系取代类之间的继承关系
- ✓ 引入了装饰类，在装饰类中既可以调用待装饰的原有类的方法，还可以增加新的方法，以扩展原有类的功能

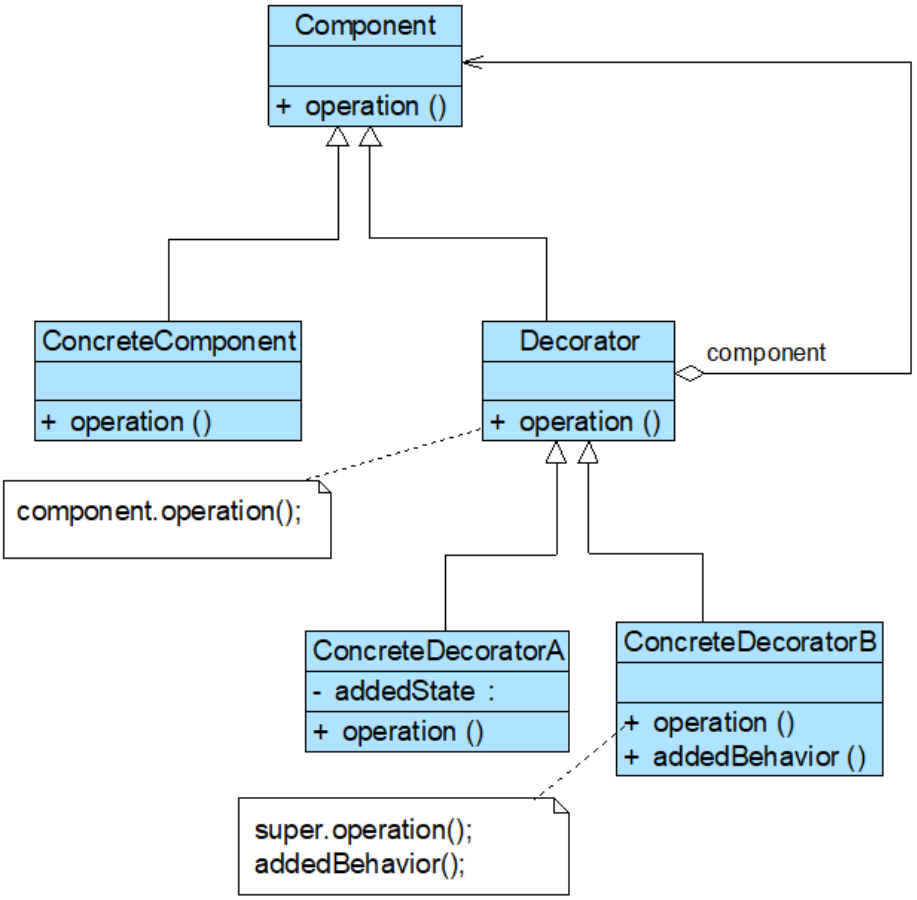
模式定义

- ✓ 装饰模式(Decorator Pattern)：动态地给一个对象增加一些额外的职责，就增加对象功能来说，装饰模式比生成子类实现更为灵活
- ✓ 对象结构型模式
- ✓ 以对客户透明的方式动态地给一个对象附加上更多的责任
- ✓ 可以在不需要创建更多子类的情况下，让对象的功能得以扩展

模式定义

- ✓ 装饰模式(Decorator Pattern)：动态地给一个对象增加一些额外的职责，就增加对象功能来说，装饰模式比生成子类实现更为灵活
- ✓ 对象结构型模式
- ✓ 以对客户透明的方式动态地给一个对象附加上更多的责任
- ✓ 可以在不需要创建更多子类的情况下，让对象的功能得以扩展

模式结构



模式结构

✓ 装饰模式包含如下角色：

- Component: 抽象构件类
- ConcreteComponent: 具体构件类
- Decorator: 抽象装饰类
- ConcreteDecorator: 具体装饰类

模式分析

✓ 透明装饰模式

- 透明(Transparent)装饰模式：要求客户端完全针对抽象编程，装饰模式的透明性要求客户端程序不应该将对象声明为具体构件类型或具体装饰类型，而应该全部声明为抽象构件类型
- 对于客户端而言，具体构件对象和具体装饰对象没有任何区别
- 可以让客户端透明地使用装饰之前的对象和装饰之后的对象，无须关心它们的区别
- 可以对一个已装饰过的对象进行多次装饰，得到更为复杂、功能更为强大的对象
- 无法在客户端单独调用新增方法addedBehavior()

模式分析

✓ 不透明装饰模式

- 半透明(Semi-transparent)装饰模式：用具体装饰类型来定义装饰之后的对象，而具体构件使用抽象构件类型来定义
- 对于客户端而言，具体构件类型无须关心，是透明的；但是具体装饰类型必须指定，这是不透明的
- 可以给系统带来更多的灵活性，设计相对简单，使用起来也非常方便
- 客户端使用具体装饰类型来定义装饰后的对象，因此可以单独调用addedBehavior()方法
- 最大的缺点在于不能实现对同一个对象的多次装饰，而且客户端需要有区别地对待装饰之前的对象和装饰之后的对象

装饰模式优点：

✓ 对于扩展一个对象的功能，装饰模式比继承更加灵活，不会导致类的个数急剧增加

✓ 可以通过一种动态的方式来扩展一个对象的功能，通过配置文件可以在运行时选择不同的具体装饰类，从而实现不同的行为

✓ 可以对一个对象进行多次装饰

✓ 具体构件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类，且原有类库代码无须改变，符合开闭原则

装饰模式缺点：

- ✓ 使用装饰模式进行系统设计时将产生很多小对象，大量小对象的产生势必会占用更多的系统资源，在一定程度上影响程序的性能
- ✓ 比继承更加易于出错，排错也更困难，对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为烦琐

在以下情况下可以使用装饰模式：

- ✓ 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责
- ✓ 当不能采用继承的方式对系统进行扩展或者采用继承不利于系统扩展和维护时可以使用装饰模式