设计模式的分类和功能

根据目的来分

> * 创建型模式：用于描述“怎样创建对象”，它的主要特点是“将对象的创建与使用分离”。

GoF 中提供了单例、原型、工厂方法、抽象工厂、建造者等 5 种创建型模式。
* 结构型模式：用于描述如何将类或对象按某种布局组成更大的结构，
GoF 中提供了代理、适配器、桥接、装饰、外观、享元、组合等 7 种结构型模式。
* 行为型模式：用于描述类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，以及怎样分配职责。
GoF 中提供了模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器等 11 种行为型模式。

根据作用范围来分

根据模式是主要用于类上还是主要用于对象上来分，这种方式可分为类模式和对象模式两种。
* 类模式：用于处理类与子类之间的关系，静态性
GoF中的工厂方法、（类）适配器、模板方法、解释器属于该模式。
* 对象模式：用于处理对象之间的关系，这些关系可以通过组合或聚合来实现，具有动态性。
GoF 中除了以上 4 种，其他的都是对象模式。

表1GoF 的 23 种设计模式的分类表

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	(类）适配器	模板方法、解释器
对象模式	单例 原型 抽象工厂 建造者	代理 (对象）适配器 桥接 装饰 外观 享元 组合	策略 命令 职责链 状态 观察者 中介者 迭代器 访问者 备忘录

各个模式的功能

单例（Singleton）模式：某个类只能生成一个实例，该类提供了一个全局访问点供外部获取该实例，其拓展是有限多例模式。
原型（Prototype）模式：将一个对象作为原型，通过对其进行复制而克隆出多个和原型类似的新实例。
工厂方法（Factory Method）模式：定义一个用于创建产品的接口，由子类决定生产什么产品。
抽象工厂（AbstractFactory）模式：提供一个创建产品族的接口，其每个子类可以生产一系列相关的产品。
建造者（Builder）模式：将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建它们，最后构建成该复杂对象。
代理（Proxy）模式：为某对象提供一种代理以控制对该对象的访问。即客户端通过代理间接地访问该对象，从而限制、增强或修改该对象的一些特性。
适配器（Adapter）模式：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。
桥接（Bridge）模式：将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度。
装饰（Decorator）模式：动态的给对象增加一些职责，即增加其额外的功能。
外观（Facade）模式：为多个复杂的子系统提供一个一致的接口，使这些子系统更加容易被访问。
享元（Flyweight）模式：运用共享技术来有效地支持大量细粒度对象的复用。
组合（Composite）模式：将对象组合成树状层次结构，使用户对单个对象和组合对

象具有一致的访问性。
模板方法（TemplateMethod）模式：定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。
策略（Strategy）模式：定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的改变不会影响使用算法的客户。
命令（Command）模式：将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。
职责链（Chain of Responsibility）模式：把请求从链中的一个对象传到下一个对象，直到请求被响应为止。通过这种方式去除对象之间的耦合。
状态（State）模式：允许一个对象在其内部状态发生改变时改变其行为能力。
观察者（Observer）模式：多个对象间存在一对多关系，当一个对象发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为。
中介者（Mediator）模式：定义一个中介对象来简化原有对象之间的交互关系，降低系统中对象间的耦合度，使原有对象之间不必相互了解。
迭代器（Iterator）模式：提供一种方法来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。
访问者（Visitor）模式：在不改变集合元素的前提下，为一个集合中的每个元素提供多种访问方式，即每个元素有多个访问者对象访问。
备忘录（Memento）模式：在不破坏封装性的前提下，获取并保存一个对象的内部状态，以便以后恢复它。
解释器（Interpreter）模式：提供如何定义语言的文法，以及对语言句子的解释方法，即解释器。

UML类图及之间的关系

类、接口和类图

- 类
  类（Class）是指具有相同属性、方法和关系的对象的抽象，它封装了数据和行为。
  UML 中，类使用包含类名、属性和操作且带有分隔线的矩形来表示。

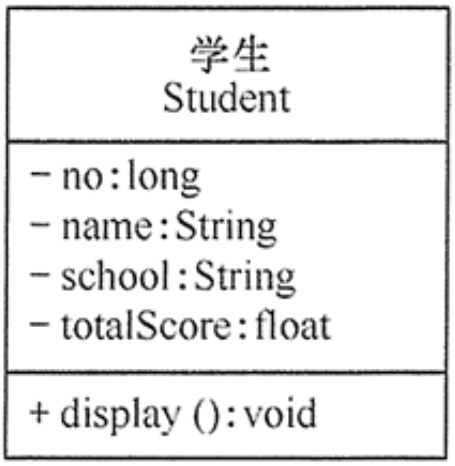


图1 Student 类

- 接口
  接口（Interface）是一种特殊的类，它具有类的结构但不可被实例化，只可以被子类实现。它包含抽象操作，但不包含属性。它描述了类或组件**对外可见的动作**。
  在 UML 中，接口使用一个带有名称的小圆圈来进行表示。

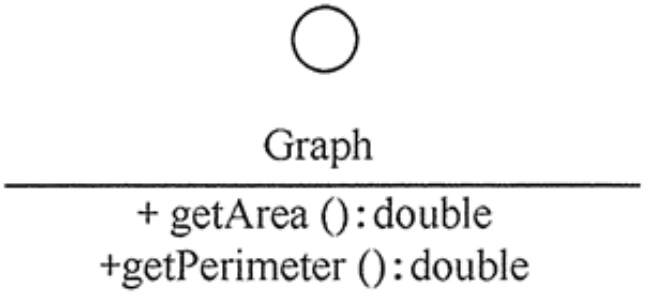


图2 Graph 接口

- 类图
  类图（ClassDiagram）是用来显示系统中的类、接口、协作以及它们之间的静态结构和关系的一种静态模型。

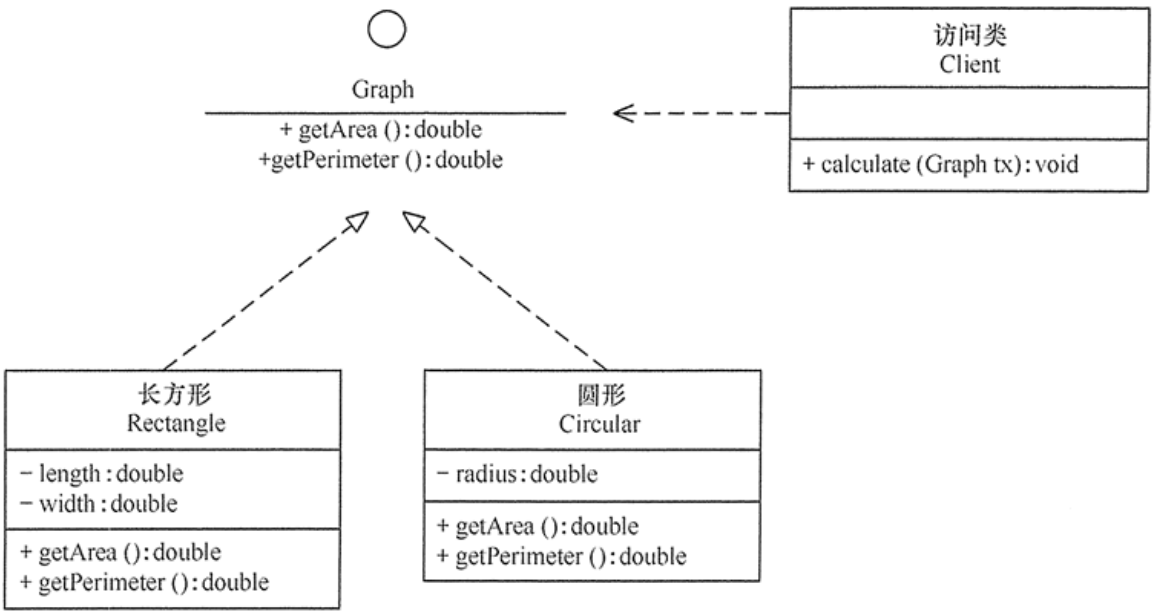


图3 “计算长方形和圆形的周长与面积”的类图

类之间的关系

类不是孤立存在的，类与类之间存在各种关系。
根据类与类之间的耦合度从弱到强排列，UML 中的类图有以下几种关系：**依赖关系、关联关系、聚合关系、组合关系、泛化关系和实现关系。**

> 其中泛化和实现的耦合度相等，它们是最强的。
>
> - 依赖关系 依赖（Dependency）关系是一种使用关系，它是对象之间耦合度**最弱**的一种关联方式，是临时性的关联。 在代码中，某个类的方法通过局部变量、方法的参数或者对静态方法的调用来访问另一个类（被依赖类）中的某些方法来完成一些职责。 在 UML 类图中，依赖关系使用**带箭头的虚线**来表示，箭头从使用类指向被依赖的类。

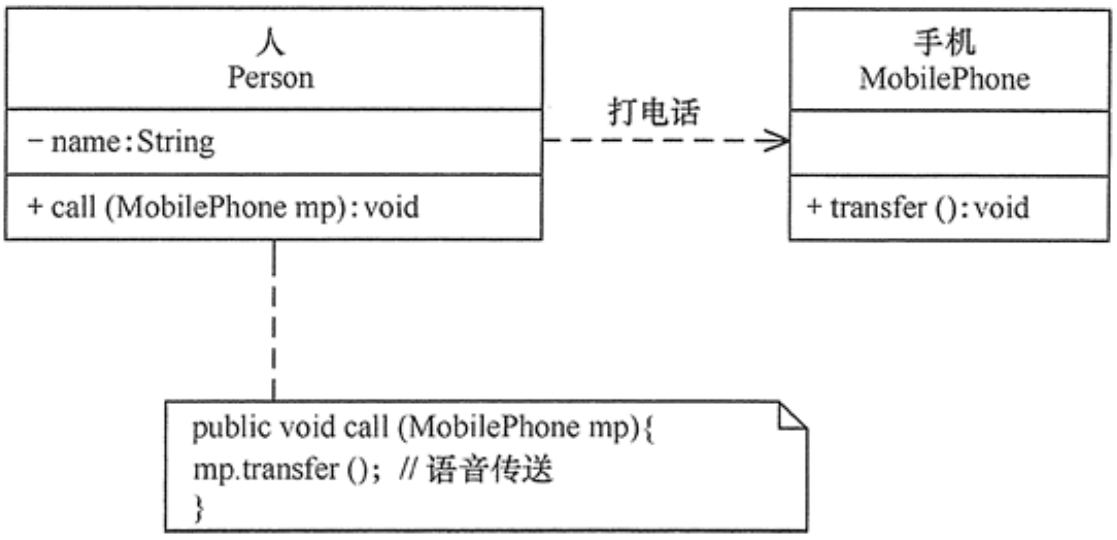


图4 依赖关系的实例

- 关联关系 关联（Association）关系是对象之间的一种引用关系，用于表示一类对象与另一类对象之间的联系，如老师和学生、师傅和徒弟。关联关系是类与类之间最常用的一种关系，分为一般关联关系、聚合关系和组合关系。关联可以是双向的，也可以是单向的。 在 UML 类图中，双向的关联可以用**带两个箭头或者没有箭头的实线**来表示，单向的关联用带一个箭头的实线来表示，箭头从使用类指向被关联的类。也可以在关联线的两端标注角色名，代表两种不同的角色。

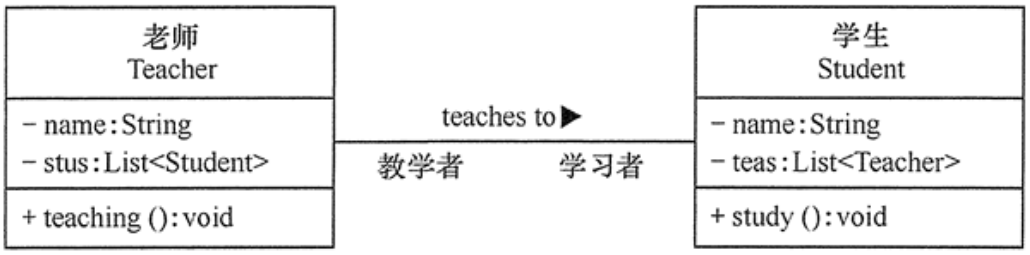


图5 关联关系的实例

- 聚合关系 聚合（Aggregation）关系是关联关系的一种，是**强关联关系**，是整体和部分之间的关系，是 has-a 的关系。 聚合关系也是通过成员对象来实现的，其中成员对象是整体对象的一部分，但是成员对象可以脱离整体对象而独立存在。例如，学校与老师的关系，学校包含老师，但如果学校停办了，老师依然存在。 在 UML 类图中，聚合关系可以用**带空心菱形的实线**来表示，**菱形指向整体**。

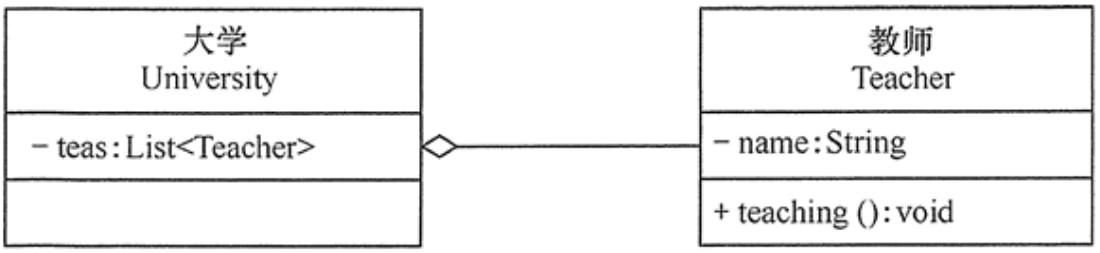


图6 聚合关系的实例

- 组合关系 组合（Composition）关系也是关联关系的一种，也表示类之间的整体与部分的关系，但它是一种更强烈的聚合关系，是 cxmtains-a 关系。 在组合关系中，整体对象可以控制部分对象的生命周期，一旦整体对象不存在，部分对象也将不存在，部分对象不能脱离整体对象而存在。例如，头和嘴的关系，没有了头，嘴也就不存在了。 在 UML 类图中，组合关系用**带实心菱形的实线**来表示，**菱形指向整体**。

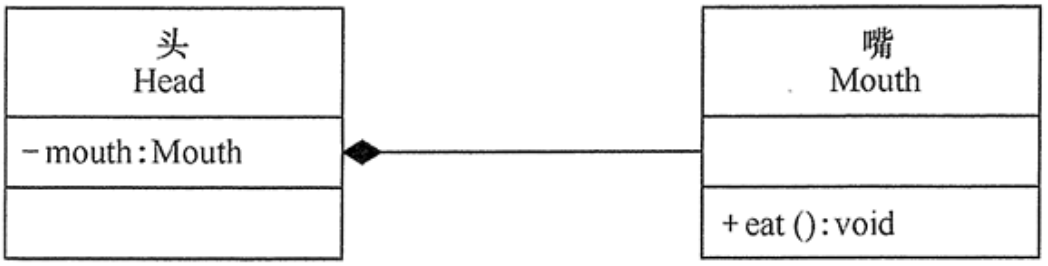


图7 组合关系的实例

- 泛化关系 泛化（Generalization）关系是对象之间耦合度最大的一种关系，表示一般与特殊的关系，是父类与子类之间的关系，是一种继承关系，是 is-a 的关系。 在 UML 类图中，泛化关系用**带空心三角箭头的实线**来表示，**箭头从子类指向父类**。在代码实现时，使用面向对象的继承机制来实现泛化关系。例如，Student 类和 Teacher 类都是 Person 类的子类。

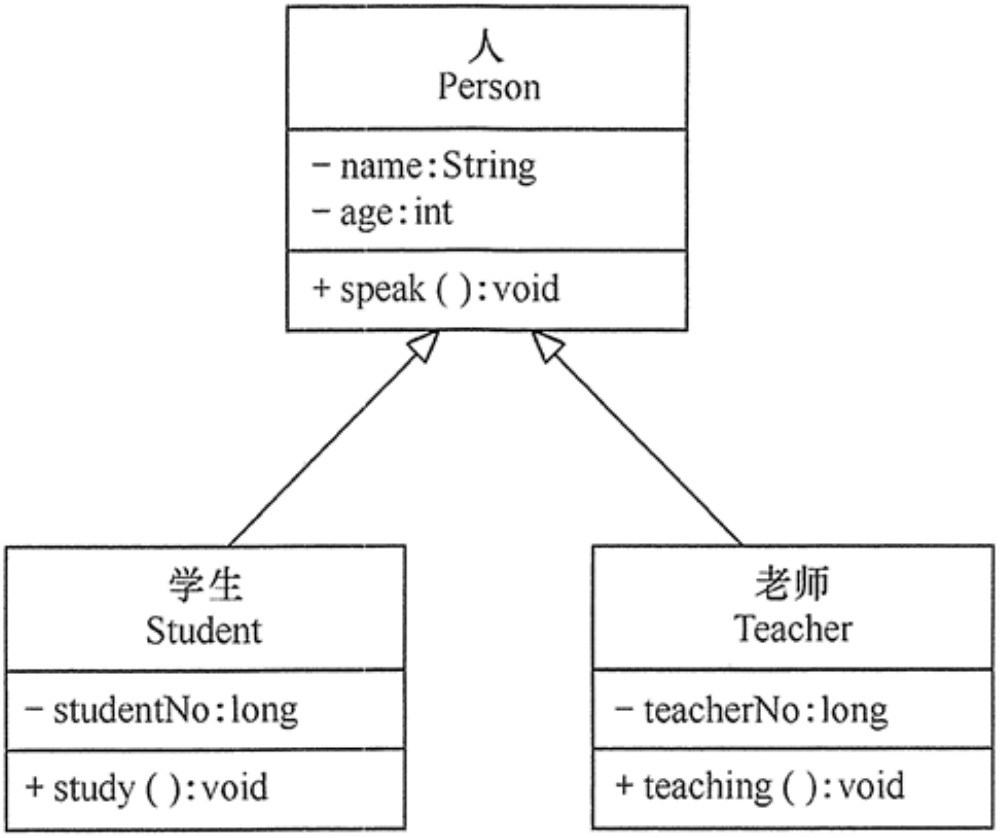


图8 泛化关系的实例

- 实现关系 实现（Realization）关系是接口与实现类之间的关系。在这种关系中，类实现了接口，类中的操作实现了接口中所声明的所有的抽象操作。 在 UML 类图中，实现关系使用**带空心三角箭头的虚线**来表示，**箭头从实现类指向接口**。例如，汽车和船实现了交通工具。

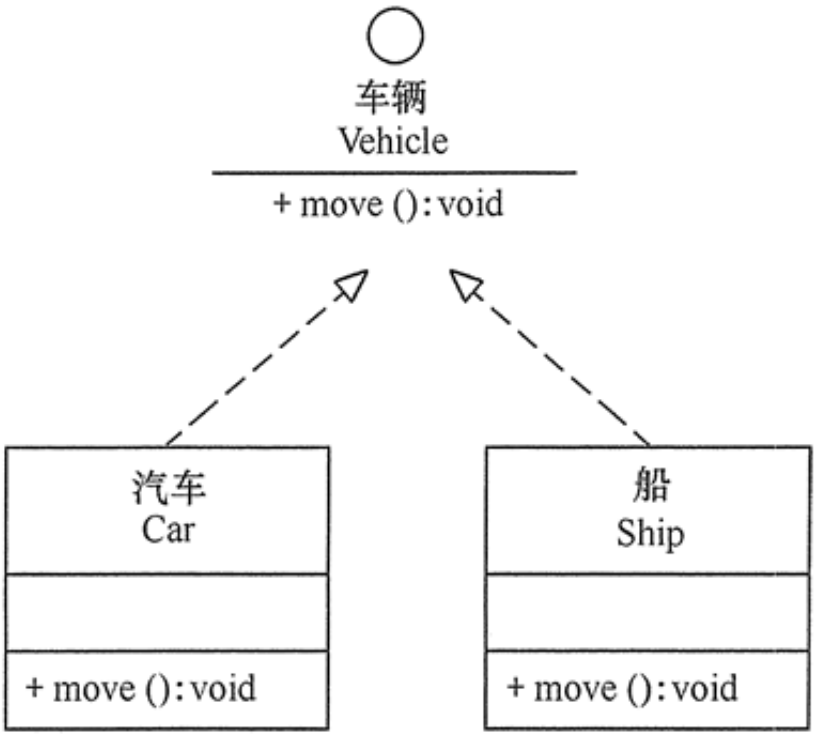


图9 实现关系的实例

设计模式

创建型模式

单例模式

单例模式（Singleton Pattern）是设计模式中最简单的形式之一，其目的是使得类的一个对象成为系统中的唯一实例。

> 这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一对象的方式，可以直接访问，不需要实例化该类的对象。
> 在计算机系统中，还有 Windows 的回收站、操作系统中的文件系统、多线程中的线程池、显卡的驱动程序对象、打印机的后台处理服务、应用程序的日志对象、数据库的连接池、网站的计数器、Web 应用的配置对象、应用程序中的对话框、系统中的缓存等常常被设计成单例。

要点

- 单例模式的要点有三个：
    - 单例类有且仅有一个实例

    - 单例类必须自行创建自己的唯一实例
    - 单例类必须给所有其他对象提供这一实例
- 从具体实现角度来说，可分为以下三点：
    - 提供一个 private 构造函数（防止外部调用而构造类的实例）
    - 提供一个该类的 static private 对象
    - 提供一个 static public 函数，用于创建或获取其本身的静态私有对象（例如：GetInstance()）
- 除此之外，还有一些关键点（需要多加注意，很容易忽视）：
    - 线程安全（双检锁 - DCL，即：double-checked locking）
    - 资源释放

多种实现方式

- 局部静态变量

```
// 单例
class Singleton{
public:
    static Singleton& GetInstance(){
        static Singleton instance;
        return instance;
    }
    void doSomething() {
        cout << "Do something" << endl;
    }
private:
    Singleton() {} // 构造函数（被保护）
    Singleton(Singleton const &); // 无需实现，实现不能复制拷贝
    Singleton& operator = (const Singleton &); // 无需实现，实现不能赋值拷贝
};
```

```
Singleton::GetInstance().doSomething(); // OK 相当于调用GetInstance()生成第一个实例，然后通过该实例调用doSomething()函数
```

```
Singleton single = Singleton::GetInstance(); // Error 不能编译通过，不允许进行拷贝构造
```

- 懒汉式/饿汉式
  Singleton 的头文件（懒汉式/饿汉式公用）

```
// singleton.h
#ifndef SINGLETON_H
#define SINGLETON_H

// 单例 - 懒汉式/饿汉式公用
class Singleton{
public:
    static Singleton* GetInstance(); //可以避免对象赋值初始化（返回的是对象指针）
private:
    Singleton() {} // 构造函数（被保护）
private:
    static Singleton *m_pSingleton; // 指向单例对象的指针
};

#endif // SINGLETON_H
```

- 懒汉式的特点：
  - Lazy 初始化
  - 非多线程安全（第一次生成对象多个线程同时访问）

优点：第一次调用才初始化，避免内存浪费。
缺点：必须加锁（在“线程安全”部分分享如何加锁）才能保证单例，但加锁会影响效率。
实现代码如下：

```
// singleton.cpp
#include "singleton.h"
// 单例 - 懒汉式
Singleton *Singleton::m_pSingleton = NULL;

Singleton *Singleton::GetInstance(){
```

```
    if (m_pSingleton == NULL)
        m_pSingleton = new Singleton(); //第一次调用才初始化
    return m_pSingleton;
}
```

- 饿汉式的特点：
  - 非 Lazy 初始化
  - 多线程安全

优点：没有加锁，执行效率会提高。
缺点：类加载时就初始化，浪费内存。

```
// singleton.cpp
#include "singleton.h"
// 单例 - 饿汉式
Singleton *Singleton::m_pSingleton = new Singleton();  //类加载时就初始化
Singleton *Singleton::GetInstance(){
    return m_pSingleton;
}
```

- 线程安全
  在懒汉式下，如果使用多线程，会出现线程安全隐患（多个线程同时发出调用申请的情况）。为了解决这个问题，可以引入双检锁 - DCL 机制。

```
//singleton.h
// 单例 - 懒汉式/饿汉式公用
class Singleton{
public:
    static Singleton* GetInstance();
private:
    Singleton() {} // 构造函数（被保护）
private:
    static Singleton *m_pSingleton; // 指向单例对象的指针
    static mutex m_mutex; // 锁
};
```

```cpp
// singleton.cpp
#include "singleton.h"
// 单例 - 懒汉式（双检锁 DCL 机制）
Singleton *Singleton::m_pSingleton = NULL;
mutex Singleton::m_mutex;
Singleton *Singleton::GetInstance(){
    if (m_pSingleton == NULL) {
        std::lock_guard<std::mutex> lock(m_mutex); // 自解锁，当离开{}时会调用析构函数，自动解锁
        if (m_pSingleton == NULL) {
            m_pSingleton = new Singleton();
        }
    }
    return m_pSingleton;
}
```

- 资源释放
  有内存申请，就要有对应的释放，可以采用下述两种方式：
- 主动释放（手动调用接口来释放资源）
- 自动释放（由程序自己释放）

要手动释放资源，添加一个 static 接口，编写需要释放资源的代码：

```cpp
// 单例 - 主动释放
static void DestoryInstance(){
    if (m_pSingleton != NULL) {
        delete m_pSingleton;
        m_pSingleton = NULL;
    }
}
```

然后在需要释放的时候，手动调用该接口：

```cpp
Singleton::GetInstance()->DestoryInstance();
```

自动释放的方法

```
```

```cpp
// 单例 - 自动释放
class Singleton{
public:
    static Singleton* GetInstance();
private:
    Singleton() {} // 构造函数（被保护）
private:
    static Singleton *m_pSingleton; // 指向单例对象的指针

    // GC 机制
    class GC {
    public:
        ~GC(){
            // 可以在这里销毁所有的资源，例如：db 连接、文件句柄等
            if (m_pSingleton != NULL) {
                cout << "Here destroy the m_pSingleton..." << endl;
//实际窗口是打印不出来的，因为这是程序最后结束进行回收的
                delete m_pSingleton;
                m_pSingleton = NULL;
            }
        }
        static GC gc; // 用于释放单例
    };
};
```

```cpp
// main.cpp
#include "singleton.h"
Singleton::GC Singleton::GC::gc; // 重要，对gc的初始化，会调用gc的构造函数

int main(){
    Singleton *pSingleton1 = Singleton::GetInstance();
    Singleton *pSingleton2 = Singleton::GetInstance();
    cout << (pSingleton1 == pSingleton2) << endl;
    return 0;
}
```

在程序运行结束时，系统会调用 Singleton 的静态成员 GC 的析构函数，该析构函数会进行资源的释放。这种方式的最大优点就是在“不知不觉”中进行，所以，对我们来

说，尤为省心。

个人理解

模型反正就是那个类只有一个实体，并且是通过调用该类的静态public函数来进行访问的。实现单例的方式有：局部静态变量，懒汉模式（开始没初始化实体，后面调用时生成，存在线程安全问题），饿汉模式（初始化时就生成实体，不存在线程安全问题）。其中针对线程安全问题引入了自解锁（离开作用区域时自动解掉互斥锁）。还有资源自动释放方法，引入GC机制，实现程序结束时调用GC类中的一个静态实体析构函数对申请的资源进行释放。

原型模式

在有些系统中，存在大量相同或相似对象的创建问题，如果用传统的构造函数来创建对象，会比较复杂且耗时耗资源，用原型模式生成对象就很高效，就像孙悟空拔下猴毛轻轻一吹就变出很多孙悟空一样简单。

定义与特点

原型（Prototype）模式的定义如下：用一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型相同或相似的新对象。在这里，原型实例指定了要创建的对象的种类。
用这种方式创建对象非常高效，根本无须知道对象创建的细节。例如，Windows 操作系统的安装通常较耗时，如果复制就快了很多。

模式结构与实现

原型模式包含以下主要角色。

- 抽象原型类：规定了具体原型对象必须实现的接口。
- 具体原型类：实现抽象原型类的 clone() 方法，它是可被复制的对象。
- 访问类：使用具体原型类中的 clone() 方法来复制新的对象。

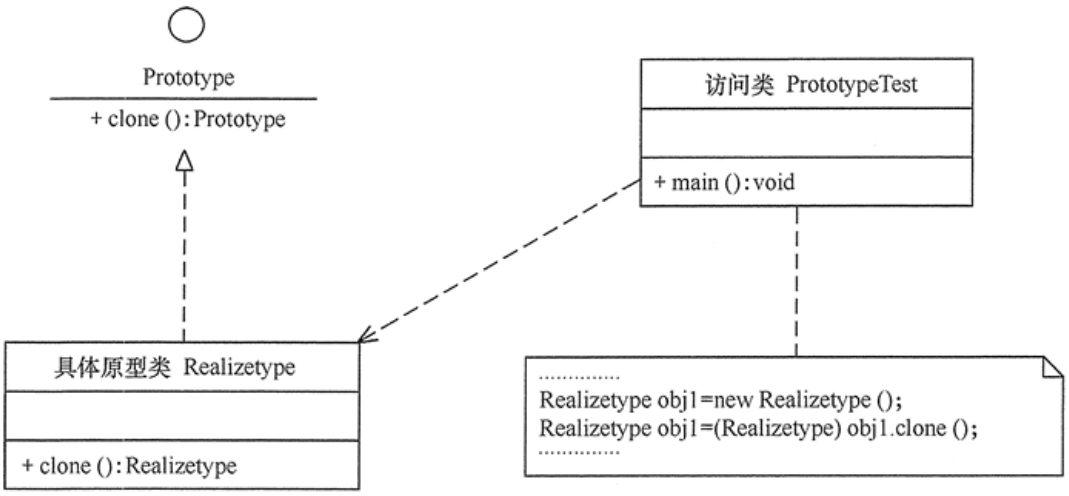


图1 原型模式的结构图

工厂方法模式

工厂方法（FactoryMethod）模式的定义：定义一个创建产品对象的工厂接口，将产品对象的实际创建工作推迟到具体子工厂类当中。这满足创建型模式中所要求的“创建与使用相分离”的特点。

> 被创建的对象称为“产品”，把创建产品的对象称为“工厂”。
> 如果要创建的产品不多，只要一个工厂类就可以完成，这种模式叫“简单工厂模式”，它不属于 GoF 的 23 种经典设计模式，它的缺点是增加新产品时会违背“开闭原则”。
> 本节介绍的“工厂方法模式”是对简单工厂模式的进一步抽象化，其好处是可以使系统在不修改原来代码的情况下引进新的产品，即满足开闭原则。

- 工厂方法模式的主要优点有：
  1、用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程；
  2、在系统增加新的产品时只需要添加具体产品类和对应的具体工厂类，无须对原工厂进行任何修改，满足开闭原则；

  > 其缺点是：每增加一个产品就要增加一个具体产品类和一个对应的具体工厂类，这增加了系统的复杂度。

结构与实现

模式的结构

- 工厂方法模式的主要角色如下。
  抽象工厂（Abstract Factory）：提供了创建产品的接口，调用者通过它访问具体

工厂的工厂方法 newProduct() 来创建产品。

具体工厂（ConcreteFactory）：主要是实现抽象工厂中的抽象方法，完成具体产品的创建。

抽象产品（Product）：定义了产品的规范，描述了产品的主要特性和功能。

具体产品（ConcreteProduct）：实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间一一对应。

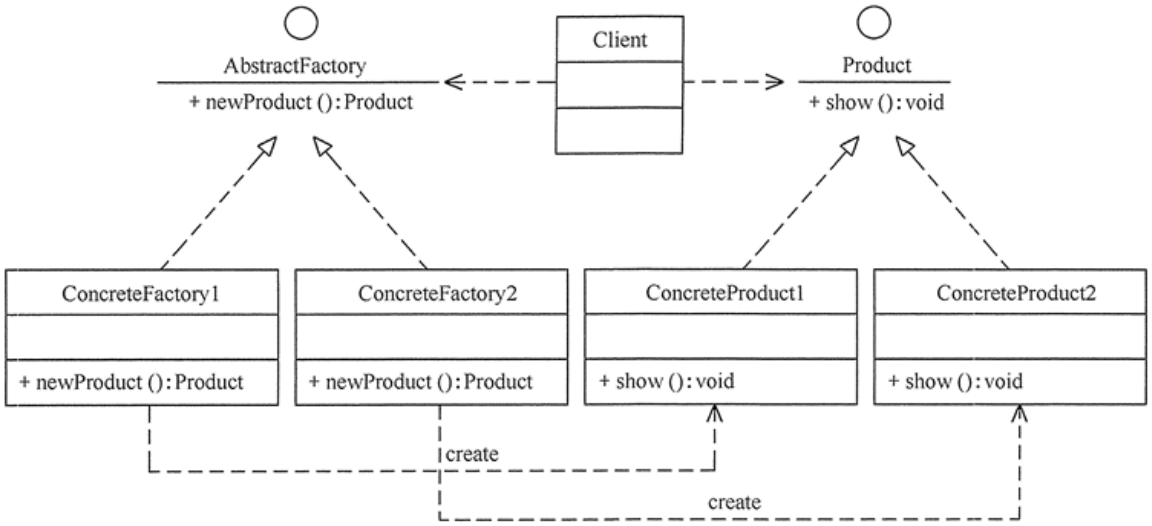


图1 工厂方法模式的结构图

抽象工厂模式

工厂方法模式只考虑生产同等级的产品，但是在现实生活中许多工厂是综合型的工厂，能生产多等级（种类）的产品，如农场里既养动物又种植物，电器厂既生产电视机又生产洗衣机或空调。抽象工厂模式将考虑多等级产品的生产，将同一个具体工厂所生产的位于不同等级的一组产品称为一个产品族

抽象工厂模式（Abstract Factory Pattern）是所有形态的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式是指当有多个抽象角色时，使用的一种工厂模式。抽象工厂模式可以向客户端提供一个接口，使客户端在不必指定产品的具体的情况下，创建多个产品族中的产品对象。

- Factory（抽象工厂）：声明一个用于创建抽象产品的接口
- ConcreteFactory（具体工厂）：用于创建具体的产品
- Product（抽象产品）：声明一个产品对象类型的接口
- ConcreteProduct（具体产品）：由具体工厂创建的具体产品

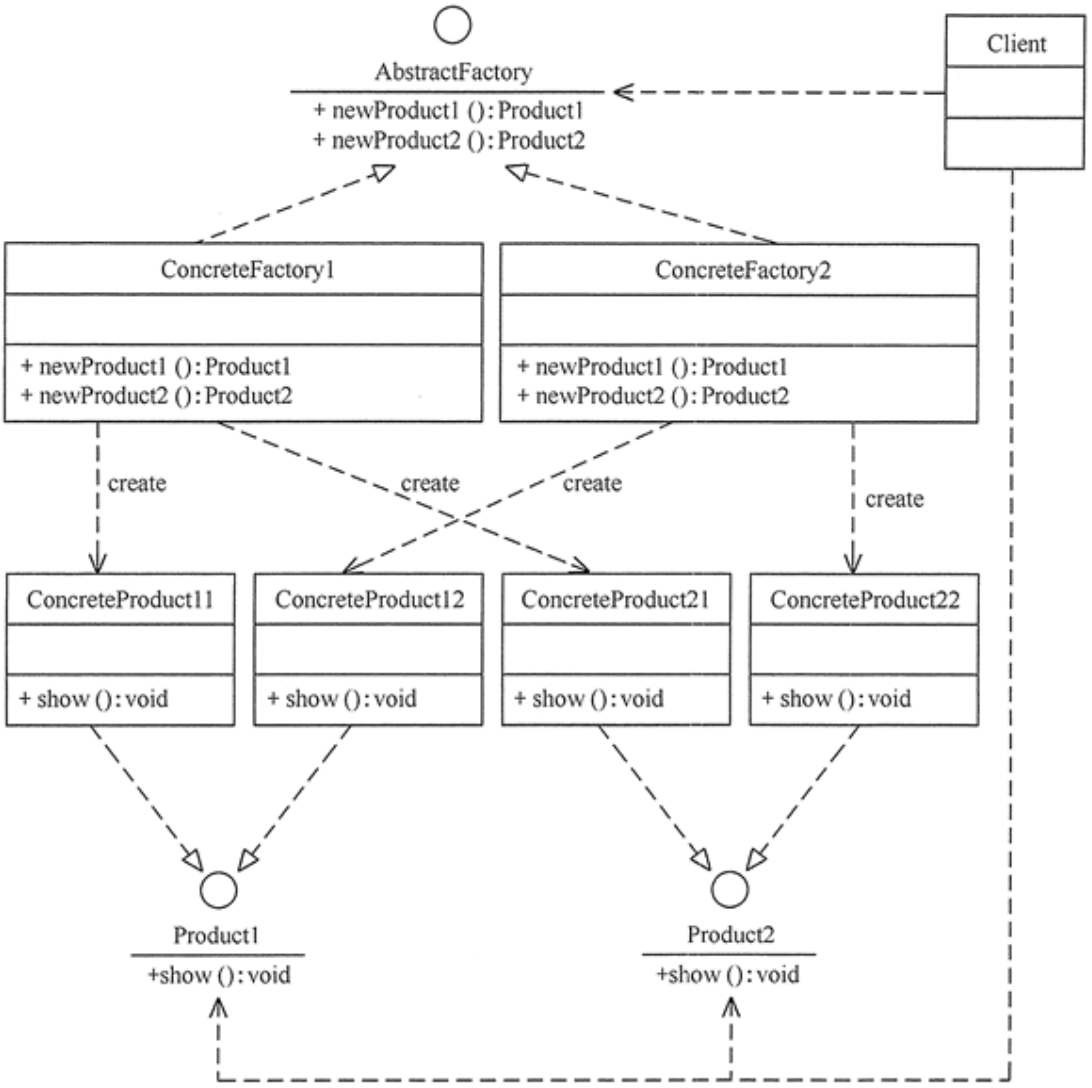


图2 抽象工厂模式的结构图

优缺点及适用场景

优点：

- 封装了产品的创建，使得不需要知道具体是哪种产品，只需要知道是哪个工厂即可。
- 可以支持不同类型的产品，使得模式灵活性更强。
- 可以非常方便的使用一族中的不同类型的产品。

缺点：

- 结构过于臃肿，如果产品类型较多或产品族较多，会非常难于管理。
- 每次如果添加一组产品，那么所有的工厂类都必须添加一个方法，这样违背了开放-封闭原则。所以一般适用于产品组合产品族变化不大的情况。
- 在不必指定产品的具体的情况下，创建多个产品族中的产品对象。

案例分析

> 工厂方法模式 要求产品必须为同一类型，也就是说，BBA 只能生产汽车，要生产其他产品（例如：自行车）是不行的，这显然限制了产品的扩展。为了解决这个问题，抽象工厂模式出现了 - 将产品归类分组，然后将好几组产品构成一族。每个工厂负责生产一族产品，而工厂中的每个方法负责生产一种类型的产品。
> 这样，客户端只需要创建具体工厂的实例，然后调用工厂对象的工厂方法就可以得到所需要的产品对象。

代码实现

- 创建抽象产品
  在我们的示例中，需要有两个产品 - 汽车和自行车：

```
// product.h
#include <string>
using namespace std;

// 汽车接口
class ICar{
public:
    virtual string Name() = 0; // 汽车名称
};
// 自行车接口
class IBike{
public:
    virtual string Name() = 0; // 自行车名称
};
```

- 创建具体产品
  有了抽象产品，继续创建一些具体的产品：

```
// concrete_product.h
#include "product.h"
/********** 汽车 **********/
// 奔驰
class BenzCar : public ICar{
public:
    string Name() {
```

```
        return "Benz Car";
    }
};
// 宝马
class BmwCar : public ICar{
public:
    string Name() {
        return "Bmw Car";
    }
};
// 奥迪
class AudiCar : public ICar{
public:
    std::string Name() {
        return "Audi Car";
    }
};

/********** 自行车 **********/
// 奔驰
class BenzBike : public IBike{
public:
    string Name() {
        return "Benz Bike";
    }
};
// 宝马
class BmwBike : public IBike{
public:
    string Name() {
        return "Bmw Bike";
    }
};
// 奥迪
class AudiBike : public IBike{
public:
    string Name() {
        return "Audi Bike";
    }
};
```

这样，为汽车和自行车都准备好了所有的具体类。

- 创建抽象工厂
  产品有了，当然要有相应的制造商与其相关联，所以呢，要有具体的工厂。但在这之前，需要一个抽象工厂：

```
// factory.h
#include "product.h"

// 抽象工厂
class AFactory{
public:
    enum FACTORY_TYPE {
        BENZ_FACTORY, // 奔驰工厂
        BMW_FACTORY, // 宝马工厂
        AUDI_FACTORY // 奥迪工厂
    };
    virtual ICar* CreateCar() = 0; // 生产汽车
    virtual IBike* CreateBike() = 0; // 生产自行车
    static AFactory* CreateFactory(FACTORY_TYPE factory); // 创建工厂
};
```

```
// factory.cpp
#include "factory.h"
#include "concrete_factory.h"

// 创建工厂
AFactory* AFactory::CreateFactory(FACTORY_TYPE factory){
    AFactory *pFactory = NULL;
    switch (factory) {
    case FACTORY_TYPE::BENZ_FACTORY: // 奔驰工厂
        pFactory = new BenzFactory();
        break;
    case FACTORY_TYPE::BMW_FACTORY: // 宝马工厂
        pFactory = new BmwFactory();
        break;
    case FACTORY_TYPE::AUDI_FACTORY: // 奥迪工厂
        pFactory = new AudiFactory();
```

```
        break;
    default:
        break;
    }
    return pFactory;
}
```

- 创建具体工厂
  为每个制造商创建具体的工厂：

```
// concrete_factory.h
#include "factory.h"
#include "concrete_product.h"

// 奔驰工厂
class BenzFactory : public AFactory{
public:
    ICar* CreateCar() {
        return new BenzCar();
    }
    IBike* CreateBike() {
        return new BenzBike();
    }
};

// 宝马工厂
class BmwFactory : public AFactory{
public:
    ICar* CreateCar() {
        return new BmwCar();
    }
    IBike* CreateBike() {
        return new BmwBike();
    }
};

// 奥迪工厂
class AudiFactory : public AFactory{
public:
```

```
    ICar* CreateCar() {
        return new AudiCar();
    }
    IBike* CreateBike() {
        return new AudiBike();
    }
};
```

具体的产品就与其制造商关联起来了。

- 创建客户端
  当一切准备就绪，就可以实现客户端了，利用相关产品的这种层次结构来创建产品。

```
// main.cpp
#include "factory.h"
#include "product.h"
#include <iostream>
using namespace std;
#define SAFE_DELETE(p) { if(p){delete(p); (p)=NULL;} }

int main(){
    // 奔驰
    AFactory *pFactory = AFactory::CreateFactory(AFactory::FACTORY_TYPE::BENZ_FACTORY);
 //AFactory为抽象工厂基类指针，传入的参数为工厂类型参数
    ICar *pCar = pFactory->CreateCar(); //调用奔驰工厂创建车对象
    IBike *pBike = pFactory->CreateBike();

    cout << "Benz factory - Car: " << pCar->Name() << endl;
    cout << "Benz factory - Bike: " << pBike->Name() << endl;
    SAFE_DELETE(pCar);  SAFE_DELETE(pBike);  SAFE_DELETE(pFactory);

    // 宝马
    pFactory = AFactory::CreateFactory(AFactory::FACTORY_TYPE::BMW_FACTORY);
    pCar = pFactory->CreateCar();
    pBike = pFactory->CreateBike();

    cout << "Bmw factory - Car: " << pCar->Name() << endl;
```

```
    cout << "Bmw factory - Bike: " << pBike->Name() << endl;
    SAFE_DELETE(pCar);  SAFE_DELETE(pBike);  SAFE_DELETE(pFactory);

    // 奥迪
    pFactory =
AFactory::CreateFactory(AFactory::FACTORY_TYPE::AUDI_FACTORY);
    pCar = pFactory->CreateCar();
    pBike = pFactory->CreateBike();

    cout << "Audi factory - Car: " << pCar->Name() << endl;
    cout << "Audi factory - Bike: " << pBike->Name() << endl;
    SAFE_DELETE(pCar);   SAFE_DELETE(pBike);  SAFE_DELETE(pFactory);
    getchar();
    return 0;
}
```

到这里，抽象工厂模式的基本框架已经有了，我们仅输出了具体产品的名字，其实还可以包含更多的信息，请根据需要自行扩展。

个人理解

工厂模式相当于实现了多个工厂（比如奔驰、宝马），以及每个工厂对应生产的产品（汽车、自行车，这里直接调用具体产品类）。在主程序中只需创建指定的工厂，比如宝马工厂，直接调用生产车辆函数时，宝马工厂就直接生成出宝马车了。

建造者模式

在软件开发过程中有时需要创建一个复杂的对象，这个复杂对象通常由多个子部件按一定的步骤组合而成。如游戏中的不同角色，其性别、个性、能力、脸型、体型、服装、发型等特性都有所差异；还有汽车中的方向盘、发动机、车架、轮胎等部件也多种多样。
以上所有这些产品都是由多个部件构成的，各个部件可以灵活选择，但其创建步骤都大同小异。这类产品的创建无法用前面介绍的工厂模式描述，只有建造者模式可以很好地描述该类产品的创建。

定义与特点

- 建造者（Builder）模式的定义：
  指将一个复杂对象的构造与它的表示分离，使同样的构建过程可以创建不同的表

示，这样的设计模式被称为建造者模式。
它是将一个复杂的对象分解为多个简单的对象，然后一步一步构建而成。
它将变与不变相分离，即产品的组成部分是不变的，但每一部分是可以灵活选择的。

- 该模式的主要优点如下：
  各个具体的建造者相互独立，有利于系统的扩展。
  客户端不必知道产品内部组成的细节，便于控制细节风险。
- 其缺点如下：
  产品的组成部分必须相同，这限制了其使用范围。
  如果产品的内部变化复杂，该模式会增加很多的建造者类。

建造者（Builder）模式和工厂模式的关注点不同：**建造者模式注重零部件的组装过程**，而**工厂方法模式更注重零部件的创建过程**，但两者可以结合使用。

结构与实现

模式的结构

- 建造者（Builder）模式的主要角色如下。
  产品角色（Product）：它是包含多个组成部件的复杂对象，由具体建造者来创建其各个滅部件。
  抽象建造者（Builder）：它是一个包含创建产品各个子部件的抽象方法的接口，通常还包含一个返回复杂产品的方法 getResult()。
  具体建造者(Concrete Builder）：实现 Builder 接口，完成复杂产品的各个部件的具体创建方法。
  指挥者（Director）：它调用建造者对象中的部件构造与装配方法完成复杂对象的创建，在指挥者中不涉及具体产品的信息。

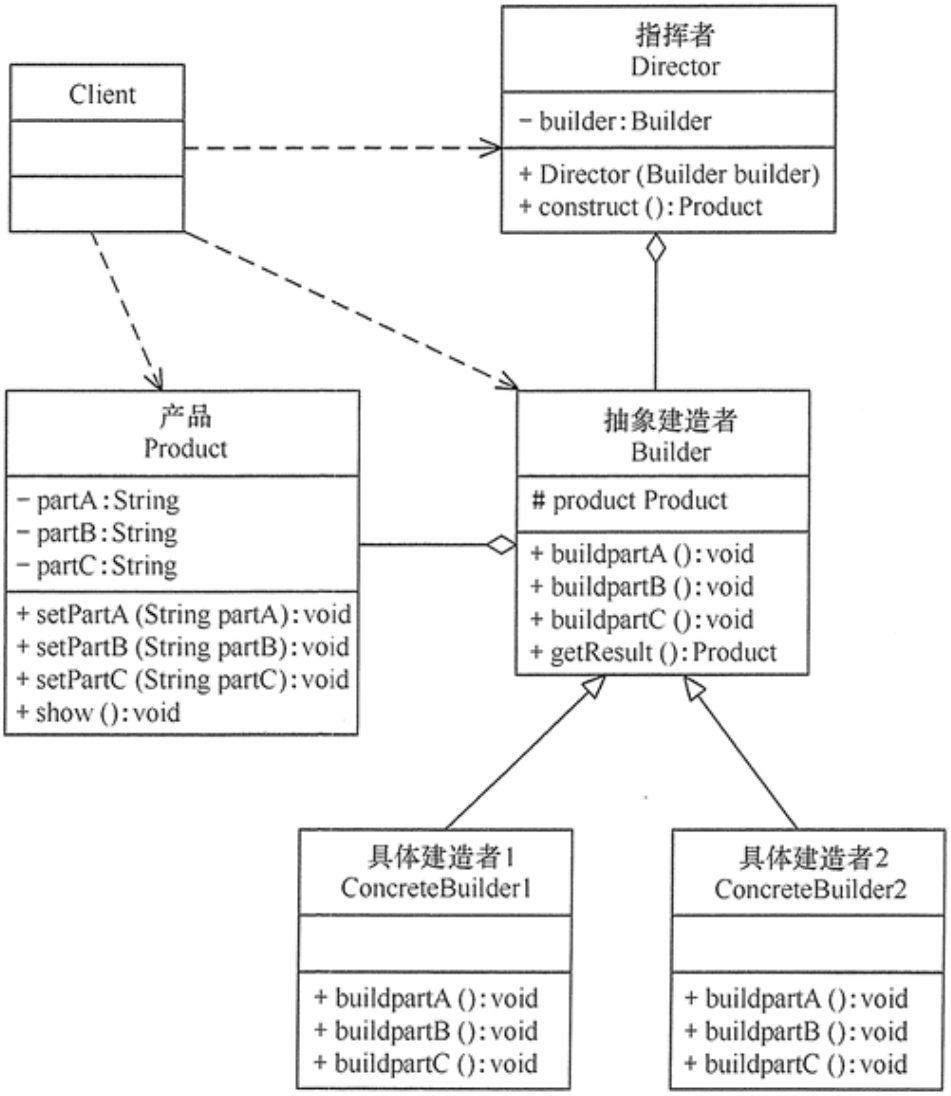


图1 建造者模式的结构图

结构性模式

代理模式

在软件设计中，使用代理模式的例子也很多，例如，要访问的远程对象比较大（如视频或大图像等），其下载要花很多时间。还有因为安全原因需要屏蔽客户端直接访问真实对象，如某单位的内部数据库等。

- 代理模式的定义与特点
  - 代理模式的定义：
    由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。
  - 代理模式的主要优点有：
    代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作

用；
代理对象可以扩展目标对象的功能；
代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度；

- 主要缺点是：
  在客户端和目标对象之间增加一个代理对象，会造成请求处理速度变慢；
  增加了系统的复杂度；

- 代理模式的主要角色如下
  抽象主题（Subject）类：通过接口或抽象类声明真实主题和代理对象实现的业务方法。
  真实主题（Real Subject）类：实现了抽象主题中的具体业务，是代理对象所代表的真实对象，是最终要引用的对象。
  代理（Proxy）类：提供了与真实主题相同的接口，其内部含有对真实主题的引用，它可以访问、控制或扩展真实主题的功能。

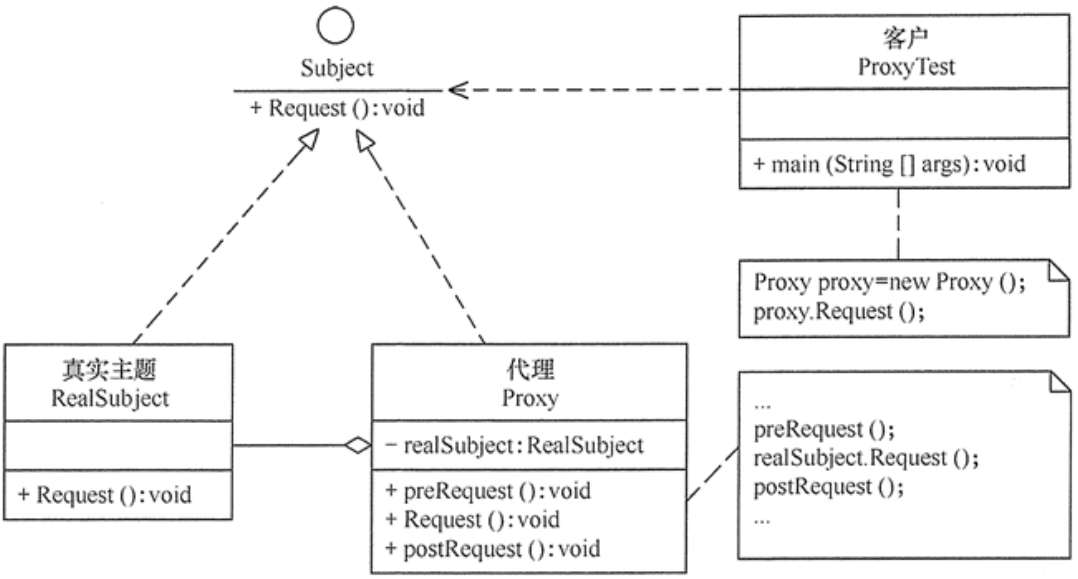


图1 代理模式的结构图

二、观察者模式

观察者模式（Observer Pattern），定义了对象间的一对多的依赖关系，让多个观察者对象同时监听某一个主题对象（被观察者）。当主题对象的状态发生更改时，会通知所有观察者，让它们能够自动更新。

背景

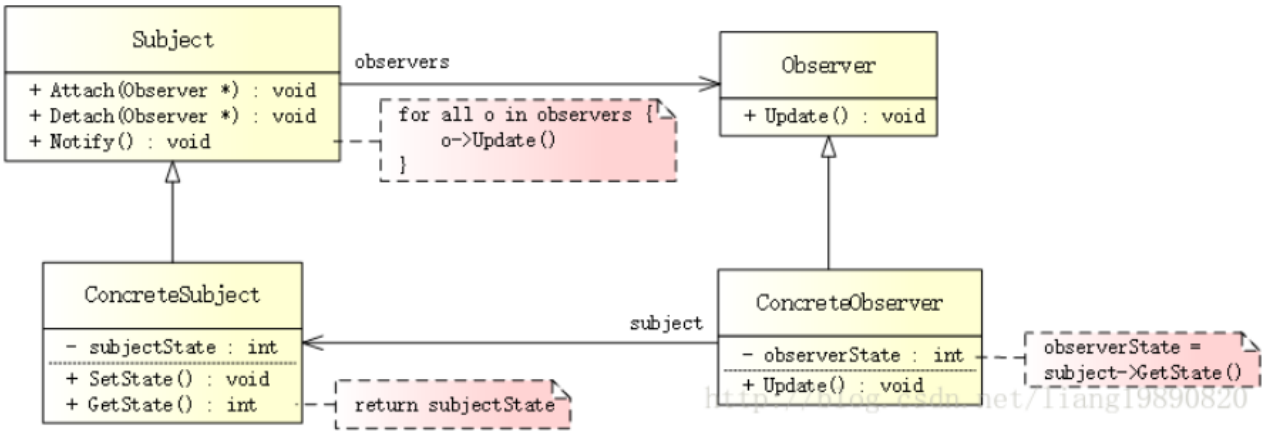
很多时候，在应用程序的一部分发生更改时，需要同时更新应用程序的其他部分。有一种方法是：让接收者反复检查发送者来进行更新，但是这种方法存在两个主要问题：

- 占用大量的 CPU 时间来检查新的状态
- 依赖于检测更新的时间间隔，可能不会立即获得更新

对于这个问题，有一个简单的解决方案 - 观察者模式。

模式结构

UML 结构图：



Subject（抽象主题）：跟踪所有观察者，并提供添加和删除观察者的接口。
Observer（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进行自我更新。
ConcreteSubject（具体主题）：将有关状态存入各 ConcreteObserver 对象。当具体主题的状态发生任何更改时，通知所有观察者。
ConcreteObserver（具体观察者）：实现 Observer 所要求的更新接口，以便使本身的状态与主题的状态相协调。

优缺点及适用场景

优点：

- 观察者和被观察者是抽象耦合的
- 建立一套触发机制

缺点：

- 如果一个被观察者对象有很多的直接和间接的观察者，将**所有的观察者都通知**到会花费很多时间。
- 如果在观察者和观察目标之间有**循环依赖**的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。
- 观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，

而仅仅只是知道观察目标发生了变化。

- 适用场景
  - 有多个子类共有的方法，且逻辑相同。
  - 重要的、复杂的方法，可以考虑作为模板方法。

案例分析：

> 自从有了滴滴、快滴、Uber、神舟等各大打车平台，广大市民的出行便利了不少。但自从合并以后，补助少了，价格也上涨了很多，不 * XX 倍甚至打不到车。。。
> 滴滴：好，第一个月，价格上调至 12.5。。。
> 过了不久，心里想着：纳尼，都垄断了，还不多涨涨，果断 15.0。。。
> 合并就是为了垄断，再无硝烟四起的价格战，整合成统一价格模式，用户也就没有了自由选择权。
> 在这里，滴滴相当于主题，司机相当于观察者。

代码实现

- 创建抽象主题
  提供关于注册、注销、通知观察者的接口：

```
// subject.h

class IObserver;
// 抽象主题
class ISubject{
public:
    virtual void Attach(IObserver *) = 0; // 注册观察者
    virtual void Detach(IObserver *) = 0; // 注销观察者
    virtual void Notify() = 0; // 通知观察者
};
```

- 创建具体主题
  抽象主题的具体实现，用于管理所有的观察者：

```
// concrete_subject.h
#include "subject.h"
```

```
#include "observer.h"
#include <iostream>
#include <list>
using namespace std;
// 具体主题
class ConcreteSubject : public ISubject{
public:
    ConcreteSubject() { m_fPrice = 10.0; }
    void SetPrice(float price) {
        m_fPrice = price;
    }

    void Attach(IObserver *observer) {
        m_observers.push_back(observer);
    }
    void Detach(IObserver *observer) {
        m_observers.remove(observer);
    }

    void Notify() {
        list<IObserver *>::iterator it = m_observers.begin();
        while (it != m_observers.end()) {
            (*it)->Update(m_fPrice);
            ++it;
        }
    }

private:
    list<IObserver *> m_observers; // 观察者列表
    float m_fPrice; // 价格
};
```

- 创建抽象观察者
  提供一个 Update() 接口，用于更新价格：

```
// observer.h
// 抽象观察者
class IObserver{
public:
```

```
    virtual void Update(float price) = 0; // 更新价格
};
```

- 创建具体观察者
  抽象观察者的具体实现，当接收到通知后，调整对应的价格：

```
// concrete_observer.h
#include "observer.h"
#include <iostream>
#include <string>
using namespace std;

// 具体观察者
class ConcreteObserver : public IObserver{
public:
    ConcreteObserver(string name) { m_strName = name; }
    void Update(float price) {
        cout << m_strName << " - price: " << price << "\n";
    }
private:
     string m_strName; // 名字
};
```

- 创建客户端主程序
  创建主题以及对应的观察者，并添加观察者并更新价格：

```
// main.cpp
#include "concrete_subject.h"
#include "concrete_observer.h"

SAFE_DELETE(p) { if(p) {delete(p); (p)=NULL;} }

int main(){
    // 创建主题、观察者
    ConcreteSubject *pSubject = new ConcreteSubject();
    IObserver *pObserver1 = new ConcreteObserver("Jack Ma");
    IObserver *pObserver2 = new ConcreteObserver("Pony");

```

```
    // 注册观察者
    pSubject->Attach(pObserver1);
    pSubject->Attach(pObserver2);

    // 更改价格，并通知观察者
    pSubject->SetPrice(12.5);
    pSubject->Notify();

    // 注销观察者
    pSubject->Detach(pObserver2);
    // 再次更改状态，并通知观察者
    pSubject->SetPrice(15.0);
    pSubject->Notify();

    SAFE_DELETE(pObserver1);
    SAFE_DELETE(pObserver2);
    SAFE_DELETE(pSubject);

    getchar();
    return 0;
}
```

个人理解

观察者模式举了滴滴平台和司机的例子。平台相当于主题，司机相当于观察者，分别都有一个抽象类和实体类。
主题中包含加入观察者的指针，删除观察者的指针，当平台需要修改价格的时候，通过观察者的指针调用更新更新价格的函数，实现修改观察者的价格。
总之，我理解的是这个模式适合单向的老师不断通知所有学生做各种事情。

三、适配器模式

适配器模式（Adapter Pattern）是一种补救模式，将一个类的接口转换成客户希望的另外一个接口，从而使原本由于接口不兼容而不能一起工作的类可以一起工作。

类适配器和对象适配器

从实现层面上划分，适配器模式分为两种：

- 类适配器（多继承方式）

- 对象适配器（对象组合方式）

实际应用中如何在二者之间进行选择？
类适配器包含以下特点：

- 由于 Adapter 直接继承自 Adaptee 类，所以，在 Adapter 类中可以对 Adaptee 类的方法进行重定义。
- 如果在 Adaptee 中添加了一个抽象方法，那么 Adapter 也要进行相应的改动，这样就带来高耦合。
- 如果 Adaptee 还有其它子类，而在 Adapter 中想调用 Adaptee 其它子类的方法时，使用类适配器是无法做到的。

对象适配器包含以下特点：

- 有的时候，你会发现，去构造一个 Adaptee 类型的对象不是很容易。
- 当 Adaptee 中添加新的抽象方法时，Adapter 类不需要做任何调整，也能正确的进行动作。
- 可以使用多态的方式在 Adapter 类中调用 Adaptee 类子类的方法。

由于对象适配器的耦合度比较低，所以在很多的书中都建议使用对象适配器。在我们实际项目中，也是如此，能使用对象组合的方式，就不使用多继承的方式。（面试中，这一块可以讲讲组合和继承之间的选择）

优缺点及适用场景

优点：

- 可以让任何两个没有关联的类一起运行
- 提高了类的复用
- 增加了类的透明度
- 灵活性好

缺点：

- 过多地使用适配器，会让系统非常零乱，不利于整体把控。
- 例如，看到调用的是 A 接口，内部却被适配成了 B 接口的实现，系统如果出现太多类似情况，无异于一场灾难。因此，如果不是很必要，可以不使用适配器，而是直接对系统进行重构。
- 适用场景

- 当想使用一个已存在的类，而它的接口不符合需求时。
- 你想创建一个可复用的类，该类可以与其他不相关的类或不可预见的类协同工作。
- 你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口，对象适配器可以适配它的父接口。

案例分析：

> 莫斯科、圣彼得堡。。。作为俄罗斯的热门旅游景点，每年都会迎来成百上千万的游客，而中国稳居其第一大客源国。
> 要去俄罗斯旅游，手机必不可少，然而，让人头疼的是**如何给手机充电**！
> 世界各国插座标准都不尽相同，甚至同一国家的不同地区也可能不一样。例如，中国一般使用两脚扁型，而俄罗斯使用的是双脚圆形。那么，如果去俄罗斯旅游，就会出现一个问题：我们带去的充电器为两脚扁型，而他们提供的插座为双脚圆形，如何给手机充电呢？总不能为了旅客而随意更改墙上的插座吧，而且俄罗斯人一直都这么使用，并且用的很好。俗话说入乡随俗，那么只能自己想办法解决了。
> 其实这个问题的解决方式很简单 - 适配器模式，只需要提供一个电源转化器即可。该转化器的一端符合俄罗斯标准，可以插到俄罗斯的插座上，另一端符合中国标准，可以供我们的手机充电器使用。

代码实现（对象适配器）

- 创建目标接口
  俄罗斯提供的插座：

```
// target.h
#include <iostream>
// 俄罗斯提供的插座
class IRussiaSocket{
public:
    // 使用双脚圆形充电（暂不实现）
    virtual void Charge() = 0;
};
```

- 创建适配者
  再来看看我们自带的充电器：

```cpp
// adaptee.h
#include <iostream>
using namespace std;

// 自带的充电器 - 两脚扁型
class OwnCharger{
public:
    void ChargeWithFeetFlat() {
        cout << "OwnCharger::ChargeWithFeetFlat" << endl;
    }
};
```

- 创建适配器
  定义一个电源适配器，并使用我们自带的充电器充电：

```cpp

// adapter.h
#include "target.h"
#include "adaptee.h"
#define SAFE_DELETE(p) { if(p){delete(p); (p)=NULL;} }

// 电源适配器
class PowerAdapter : public IRussiaSocket{
public:
    PowerAdapter() : m_pCharger( new OwnCharger() ) {}
    ~PowerAdapter() {
        SAFE_DELETE(m_pCharger);
    }
    void Charge() {
        // 使用自带的充电器（两脚扁型）充电
        m_pCharger->ChargeWithFeetFlat();
    }
private:
    OwnCharger *m_pCharger; // 持有需要被适配的接口对象（自带的充电器）
};
```

- 创建客户端
  最终，客户端实现如下

```cpp

```

```cpp
// main.cpp
#include "adapter.h"
int main(){
    // 创建适配器
    IRussiaSocket *pAdapter = new PowerAdapter();  //pAdapter 是基类俄罗斯插口指针，PowerAdapter是继承俄罗斯插口的适配器。（并且在适配器中创建了中国插头对象）
    // 充电
    pAdapter->Charge(); //用指向俄罗斯插口的基类指针，调用适配器中charge函数来实现调用中国插头。
    SAFE_DELETE(pAdapter);
    getchar();
    return 0;
}
```

这样适配器起作用了，现在可以使用两脚扁型插孔充电了。我们并没有改变俄罗斯提供的插座，只提供了一个适配器就能使用中国的标准插口充电。

> 这就是适配器模式的魅力：不改变原有接口，却还能使用新接口的功能。

- 类适配器
  Target 和 Adaptee 保持不变，只需要将 Adapter 变为**多继承的方式**即可：

```cpp
#include "target.h"
#include "adaptee.h"
// 电源适配器
class PowerAdapter : public IRussiaSocket, OwnCharger{
public:
    PowerAdapter() {}
    void Charge() {
        // 使用自带的充电器（两脚扁型）充电
        ChargeWithFeetFlat();
    }
};
```

除此之外，其他用法和“对象适配器”一致。

个人理解

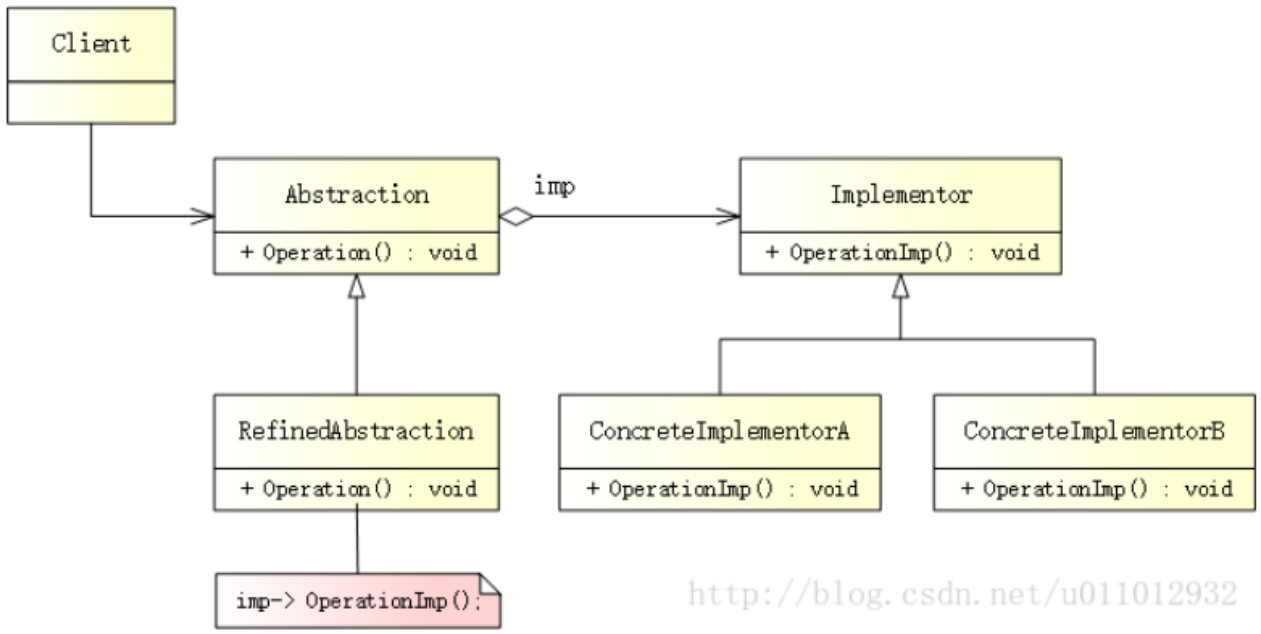
对象适配器理解：继承+组合
适配器继承了俄罗斯插头类，在适配器中也创建了中国插头对象，并实现了中国插头充电的方法。
简单来说，就是我不改变原有俄罗斯和中国的接口，通过继承俄罗斯插口类，放入中国插口对象，实现通过俄罗斯插口指针和适配器对象来调用中国插口。
类适配器理解：继承+继承

四、桥接模式

桥接模式（Bridge Pattern）是将抽象部分与它的实现部分分离，使它们都可以独立地变化。

模式结构

UML 结构图：



- Abstraction（抽象类）
  用于定义抽象类的接口，并且维护一个指向 Implementor 实现类的指针。它与 Implementor 之间具有关联关系。
- RefinedAbstraction（扩充抽象类）
  扩充由 Abstraction 定义的接口，在 RefinedAbstraction 中可以调用在 Implementor 中定义的业务方法。
- Implementor（实现类接口）
  定义实现类的接口，这个接口不一定要与 Abstraction 的接口完全一致，事实上这两个接口可以完全不同。

- ConcreteImplementor（具体实现类）
  实现了 Implementor 定义的接口，在不同的 ConcreteImplementor 中提供基本操作的不同实现。在程序运行时，ConcreteImplementor 对象将替换其父类对象，提供给 Abstraction 具体的业务操作方法。

优缺点及适用场景

优点：

- 分离抽象和实现部分。桥接模式使用“对象间的关联关系”解耦了抽象和实现之间固有的绑定关系，使得抽象和实现可以沿着各自的维度来变化。所谓抽象和实现沿着各自维度的变化，也就是说抽象和实现不再在同一个继承层次结构中，而是“子类化”它们，使它们各自都具有自己的子类，以便任何组合子类，从而获得多维度组合对象。
- 在很多情况下，桥接模式可以取代多层继承方案，多层继承方案违背了“单一职责原则”，复用性较差，且类的个数非常多，桥接模式是比多层继承方案更好的解决方法，它极大减少了子类的个数。
- 桥接模式提高了系统的可扩展性，在两个变化维度中任意扩展一个维度，都不需要修改原有系统，符合“开闭原则”。

缺点：

- 桥接模式的使用会增加系统的理解与设计难度，由于关联关系建立在抽象层，要求开发者一开始就针对抽象层进行设计与编程。
- 桥接模式要求正确识别出系统中两个独立变化的维度，因此其使用范围具有一定的局限性，如何正确识别两个独立维度也需要一定的经验积累。
- 适用场景
  - 如果一个系统需要在抽象化和具体化之间增加更多的灵活性，避免在两个层次之间建立静态的继承关系，通过桥接模式可以使它们在抽象层建立一个关联关系。
  - “抽象部分”和“实现部分”可以以继承的方式独立扩展而互不影响，在程序运行时可以动态将一个抽象化子类的对象和一个实现化子类的对象进行组合，即系统需要对抽象化角色和实现化角色进行动态耦合。
  - 一个系统存在多个（≥ 2）独立变化的维度，且这多个维度都需要独立进行扩展。
  - 对于那些**不希望使用继承或因为多层继承**导致系统类的个数急剧增加的系

统，桥接模式尤为适用。

案例分析：

> 开关和电器
> 电器是现代生活必不可少的东西，几乎每家每户都有，电视、风扇、电灯。。。
> 无论什么电器，都由开关控制。开关种类众多，有拉链式开关、两位开关、调光开关。。。
> 不管任何时候，都可以在不触及其它东西的情况下更换设备。例如，可以在不更换开关的情况下换掉灯泡，也可以在不接触灯泡或风扇的情况下更换开关，甚至可以在不接触开关的情况下将灯泡和风扇互换。
> 这看起来很自然，当然也应该这样！当不同的事物联系到一起时，它们应该在一个可以变更或者替换的系统中，以便不相互影响或者影响尽可能的小，这样才能更方便、更低成本地去管理系统。想象一下，如果要换房间里的一个灯泡，得要求把开关也换了，你会考虑使用这样的系统吗？

代码实现

- 创建实现类接口
  所有电器都有一些共性，可以被打开和关闭：

```
// implementor.h
// 电器
class IElectricalEquipment{
public:
    virtual ~IElectricalEquipment() {}
    // 打开
    virtual void PowerOn() = 0;
    // 关闭
    virtual void PowerOff() = 0;
};
```

- 创建具体实现类
  接下来，是真正的电器 - 电灯和风扇，它们实现了 IElectricalEquipment 接口：

```
// concrete_implementor.h
#include "implementor.h"
#include <iostream>
```

```cpp

// 电灯
class Light : public IElectricalEquipment{
public:
    // 开灯
    virtual void PowerOn() override {
        std::cout << "Light is on." << std::endl;
    }
    // 关灯
    virtual void PowerOff() override {
        std::cout << "Light is off." << std::endl;
    }
};

// 风扇
class Fan : public IElectricalEquipment{
public:
    // 打开风扇
    virtual void PowerOn() override {
        std::cout << "Fan is on." << std::endl;
    }
    // 关闭风扇
    virtual void PowerOff() override {
        std::cout << "Fan is off." << std::endl;
    }
};
```

- 创建抽象类
  对于开关来说，它并不知道电灯和风扇的存在，只知道自己可以控制（打开/关闭）某个电器。也就是说，每个 ISwitch 应该持有一个 IElectricalEquipment 对象：

```cpp

// abstraction.h
#include "implementor.h"
// 开关
class ISwitch{
public:
    ISwitch(IElectricalEquipment *ee) { m_pEe = ee;}
    virtual ~ISwitch() {}
```

```
    // 打开电器
    virtual void On() = 0;
    // 关闭电器
    virtual void Off() = 0;
protected:
    IElectricalEquipment *m_pEe;
};
```

- 创建扩充抽象类
  特定类型的开关很多，比如拉链式开关、两位开关：

```
// refined_abstraction.h
#include "abstraction.h"
#include <iostream>

// 拉链式开关
class PullChainSwitch : public ISwitch{
public:
    PullChainSwitch(IElectricalEquipment *ee) : ISwitch(ee) {}
    // 用拉链式开关打开电器
    virtual void On() override {
        std::cout << "Switch on the equipment with a pull chain switch." << std::endl;
        m_pEe->PowerOn();
    }
    // 用拉链式开关关闭电器
    virtual void Off() override {
        std::cout << "Switch off the equipment with a pull chain switch." << std::endl;
        m_pEe->PowerOff();
    }
};

// 两位开关
class TwoPositionSwitch : public ISwitch{
public:
    TwoPositionSwitch(IElectricalEquipment *ee) : ISwitch(ee) {}
    // 用两位开关打开电器
    virtual void On() override {
```

```
        std::cout << "Switch on the equipment with a two-position switch." << std::endl;
        m_pEe->PowerOn();
    }
    // 用两位开关关闭电器
    virtual void Off() override {
        std::cout << "Switch off the equipment with a two-position switch." << std::endl;
        m_pEe->PowerOff();
    }
};
```

- 创建客户端
  很好，是时候将开关和电器关联起来了：

```
// main.cpp
#include "refined_abstraction.h"
#include "concrete_implementor.h"
#define SAFE_DELETE(p) { if(p){delete(p); (p)=NULL;} }

int main(){
    // 创建电器 - 电灯、风扇
    IElectricalEquipment *light = new Light(); //电器基类指针指向电器派生类对象，可实现多态的作用
    IElectricalEquipment *fan = new Fan();

    /**
     * 创建开关 - 拉链式开关、两位开关
     * 将拉链式开关和电灯关联起来，将两位开关和风扇关联起来
    **/
    ISwitch *pullChain = new PullChainSwitch(light); //抽象基类指针指向开关派生类对象，实现多态的作用。（同时放入电器对象的基类指针）
    ISwitch *twoPosition= new TwoPositionSwitch(fan);

    // 开灯、关灯
    pullChain->On();
    pullChain->Off();

    // 打开风扇、关闭风扇
    twoPosition->On();
```

```
    twoPosition->Off();

    SAFE_DELETE(twoPosition);
    SAFE_DELETE(pullChain);
    SAFE_DELETE(fan);
    SAFE_DELETE(light);
    getchar();
    return 0;
}
```

个人理解

桥接模式相当于继承原来的开关抽象类和电器实现类，继承抽象类中加入多种开关，并在每种开关初始化的时候传入电器基类指针。继承电器实现类中加入了多种电器的开和关。
主程序中，首先通过电器基类指针指向新创建的灯、冰箱，然后通过开关抽象基类指针指向新创建的两位开关（也可以是拉链开关），并传入电器的基类指针，通过电器的基类指针操纵对应电器的开关。
相当于把接口集中在扩充抽象类中，电器要更换开关时，只需要把电器指针传给新开关就可以了。