目录

[1.设计模式概述 10](#_Toc150448292)

[1.1软件模式 10](#_Toc150448293)

[1.2设计模式的定义 10](#_Toc150448294)

[1.3设计模式的基本要素 11](#_Toc150448295)

[1.4设计模式的分类 11](#_Toc150448296)

[1.4.1根据其目的 11](#_Toc150448297)

[1.4.2根据范围 11](#_Toc150448298)

[1.5GOF设计模式简介 12](#_Toc150448299)

[1.6设计模式的优点 12](#_Toc150448300)

[2.简单工厂模式 13](#_Toc150448301)

[2.1模式结构 13](#_Toc150448302)

[2.2模式分析 14](#_Toc150448303)

[2.3模式实例与解析 15](#_Toc150448304)

[2.4模式优缺点 16](#_Toc150448305)

[2.5模式使用环境 17](#_Toc150448306)

[3.工厂模式 17](#_Toc150448307)

[3.1简单工厂模式的不足 17](#_Toc150448308)

[3.2模式动机 18](#_Toc150448309)

[3.3模式定义 18](#_Toc150448310)

[3.4模式结构 19](#_Toc150448311)

[3.5模式分析 19](#_Toc150448312)

[3.6模式实例 21](#_Toc150448313)

[3.7模式优缺点 22](#_Toc150448314)

[3.8适用环境 23](#_Toc150448315)

[3.9模式扩展 24](#_Toc150448316)

[3.10模式本质 25](#_Toc150448317)

[4.抽象工厂模式 25](#_Toc150448318)

[4.1模式动机 25](#_Toc150448319)

[产品等级结构： 25](#_Toc150448320)

[产品族： 25](#_Toc150448321)

[4.2模式定义 27](#_Toc150448322)

[4.3模式结构 27](#_Toc150448323)

[4.4模式分析 28](#_Toc150448324)

[4.5模式实例 30](#_Toc150448325)

[4.6模式优缺点 32](#_Toc150448326)

[4.7模式适用环境 33](#_Toc150448327)

[4.8模式扩展 34](#_Toc150448328)

[4.9模式本质 34](#_Toc150448329)

[4.10模式应用 35](#_Toc150448330)

[5.建造者模式 35](#_Toc150448331)

[5.1模式动机 35](#_Toc150448332)

[5.2模式定义 36](#_Toc150448333)

[5.4模式结构 37](#_Toc150448334)

[5.5模式分析 38](#_Toc150448335)

[5.6模式实例 39](#_Toc150448336)

[5.7模式优缺点 42](#_Toc150448337)

[优点 42](#_Toc150448338)

[缺点 42](#_Toc150448339)

[5.8模式适用环境 43](#_Toc150448340)

[5.9模式应用 43](#_Toc150448341)

[5.10模式扩展 43](#_Toc150448342)

[5.11建造者模式与抽象工厂模式的对比 44](#_Toc150448343)

[5.12模式本质 44](#_Toc150448344)

[6.原型模式 45](#_Toc150448345)

[6.1模式动机 45](#_Toc150448346)

[6.2模式定义 45](#_Toc150448347)

[6.3模式结构 46](#_Toc150448348)

[6.4模式分析 46](#_Toc150448349)

[6.5 模型实例 48](#_Toc150448350)

[6.6模式优缺点 51](#_Toc150448351)

[6.7模式适用环境 52](#_Toc150448352)

[6.8模式应用 52](#_Toc150448353)

[6.9模式扩展 52](#_Toc150448354)

[6.10模式本质 53](#_Toc150448355)

[7.单例模式 53](#_Toc150448356)

[7.1.模式动机 53](#_Toc150448357)

[7.2.模式定义 53](#_Toc150448358)

[7.3模式分析 54](#_Toc150448359)

[7.4模式实例 54](#_Toc150448360)

[7.5模式优缺点 55](#_Toc150448361)

[7.6模式适用环境 56](#_Toc150448362)

[7.7模式应用 56](#_Toc150448363)

[7.8模式扩展 56](#_Toc150448364)

[7.9模式本质 57](#_Toc150448365)

[结构型模式 58](#_Toc150448366)

[结构性模式概述 58](#_Toc150448367)

[结构型模式简介 58](#_Toc150448368)

[8.适配器模式 59](#_Toc150448369)

[8.1模式动机 59](#_Toc150448370)

[8.2模式定义 59](#_Toc150448371)

[8.3模式分析 61](#_Toc150448372)

[8.4模式优缺点 64](#_Toc150448373)

[类适配器 64](#_Toc150448374)

[对象适配器 64](#_Toc150448375)

[8.6适用环境 65](#_Toc150448376)

[8.7模式应用 65](#_Toc150448377)

[8.8模式本质 65](#_Toc150448378)

[9.桥接模式 66](#_Toc150448379)

[9.1模式动机 66](#_Toc150448380)

[9.2模式定义 66](#_Toc150448381)

[9.3模式结构 67](#_Toc150448382)

[9.4模式分析 67](#_Toc150448383)

[9.5模式实例 69](#_Toc150448384)

[9.6模式优缺点 71](#_Toc150448385)

[9.7适用环境 72](#_Toc150448386)

[9.8模式应用 72](#_Toc150448387)

[9.9模式本质 73](#_Toc150448388)

[10.组合模式 73](#_Toc150448389)

[10.1模式动机 73](#_Toc150448390)

[10.2模式定义 73](#_Toc150448391)

[10.3模式结构 74](#_Toc150448392)

[10.4模式分析 75](#_Toc150448393)

[10.5 模式实例 77](#_Toc150448394)

[10.6模式优缺点 79](#_Toc150448395)

[10.7模式适用环境 80](#_Toc150448396)

[10.8模式本质 81](#_Toc150448397)

[11.装饰模式 81](#_Toc150448398)

[11.1模式定义 81](#_Toc150448399)

[11.2模式分析 81](#_Toc150448400)

[11.3模式实例 83](#_Toc150448401)

[11.4模式优缺点 85](#_Toc150448402)

[11.5模式适用环境 86](#_Toc150448403)

[11.6模式本质 86](#_Toc150448404)

[12.外观模式 86](#_Toc150448405)

[12.1模式定义 86](#_Toc150448406)

[12.2模式分析 87](#_Toc150448407)

[12.3模式优缺点 88](#_Toc150448408)

[12.4模式适用环境 89](#_Toc150448409)

[13.享元模式 89](#_Toc150448410)

[13.1模式动机与定义 89](#_Toc150448411)

[13.2模式实例 90](#_Toc150448412)

[13.3模式优缺点 93](#_Toc150448413)

[13.4适用环境 93](#_Toc150448414)

[13.5模式本质 94](#_Toc150448415)

[14.代理模式 94](#_Toc150448416)

[14.2模式分析 94](#_Toc150448417)

[14.3模式结构 95](#_Toc150448418)

[14.4模式实例 97](#_Toc150448419)

[14.5模式优缺点 99](#_Toc150448420)

[14.6模式适用环境 100](#_Toc150448421)

[14.7 模式本质 101](#_Toc150448422)

[行为型模式 101](#_Toc150448423)

[15.职责链模式 102](#_Toc150448424)

[15.1模式动机及定义 102](#_Toc150448425)

[14.2模式结构 103](#_Toc150448426)

[14.3模式分析 104](#_Toc150448427)

[14.4模式优缺点 106](#_Toc150448428)

[14.5模式适用环境 107](#_Toc150448429)

[14.6模式本质 107](#_Toc150448430)

[15.命令模式 107](#_Toc150448431)

[15.1模式动机及定义 107](#_Toc150448432)

[15.2模式结构 108](#_Toc150448433)

[15.3模式分析 109](#_Toc150448434)

[15.4命令扩展 111](#_Toc150448435)

[命令队列 111](#_Toc150448436)

[请求日志 111](#_Toc150448437)

[15.5模式实例 111](#_Toc150448438)

[15.6模式优缺点 113](#_Toc150448439)

[15.7模式优缺点 114](#_Toc150448440)

[16.迭代器模式 114](#_Toc150448441)

[16.1模式动机及定义 114](#_Toc150448442)

[16.2模式结构 115](#_Toc150448443)

[16.3模式实例 116](#_Toc150448444)

[16.4模式优缺点 119](#_Toc150448445)

[16.5模式适用环境 120](#_Toc150448446)

[16.6模式本质 120](#_Toc150448447)

[17.中介者模式 120](#_Toc150448448)

[17.1模式动机及定义 120](#_Toc150448449)

[17.2模式结构 121](#_Toc150448450)

[17.3模式分析 121](#_Toc150448451)

[17.4模式实例 124](#_Toc150448452)

[17.5模式优缺点 127](#_Toc150448453)

[17.6模式本质 127](#_Toc150448454)

[18.备忘录模式 127](#_Toc150448455)

[18.1模式动机及定义 127](#_Toc150448456)

[18.2模式分析 129](#_Toc150448457)

[18.3模式实例 131](#_Toc150448458)

[18.4模式优缺点 133](#_Toc150448459)

[18.5模式本质 134](#_Toc150448460)

[19.观察者模式 134](#_Toc150448461)

[19.1模式动机及定义 134](#_Toc150448462)

[19.2模式分析 136](#_Toc150448463)

[19.3模式实例 138](#_Toc150448464)

[19.4 模式优缺点 139](#_Toc150448465)

[19.5模式本质 140](#_Toc150448466)

[20.状态模式 140](#_Toc150448467)

[20.2模式结构 141](#_Toc150448468)

[20.3状态模式 142](#_Toc150448469)

[20.4模式实例 144](#_Toc150448470)

[20.5模式优缺点 145](#_Toc150448471)

[20.6模式本质 146](#_Toc150448472)

[21.策略模式 146](#_Toc150448473)

[21.1模式动机及定义 146](#_Toc150448474)

[21.2模式分析 146](#_Toc150448475)

# 1.设计模式概述

## 1.1软件模式



软件模式与具体的应用领域无关，在模式发现过程中需要遵循大三律(Rule of Three)，即只有经过三个以上不同类型（或不同领域）的系统的校验，一个解决方案才能从候选模式升格为模式。

## 1.2设计模式的定义

设计模式(Design Pattern)是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结，使用设计模式是为了可复用代码、让代码更容易被他人理解、保证代码可靠性等。

## 1.3设计模式的基本要素

设计模式一般有如下几个基本要素：

模式名称、问题、目的、解决方案、效果、实例代码和相关设计模式，其中的关键元素包括以下四个方面：

* 模式名称 (Pattern name)
* 问题 (Problem)
* 解决方案 (Solution)
* 效果 (Consequences)

## 1.4设计模式的分类

1.4.1根据其目的（模式是用来做什么的）

可分为创建型(Creational)，结构型(Structural)和行为型(Behavioral)三种：

* 创建型模式主要用于创建对象。
* 结构型模式主要用于处理类或对象的组合。
* 行为型模式主要用于描述对类或对象怎样交互和怎样分配职责。

### 1.4.2根据范围

即模式主要是用于处理类之间关系还是处理对象之间的关系，可分为类模式和对象模式两种：

* 类模式处理类和子类之间的关系，这些关系通过继承建立，在编译时刻就被确定下来，是属于静态的。
* 对象模式处理对象间的关系，这些关系在运行时刻变化，更具动
* 态性。

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |

## 1.5GOF设计模式简介

GoF 最先将模式的概念引入软件工程领域，他们归纳发表了23种在软件开发中使用频率较高的设计模式，旨在用模式来统一沟通面向对象方法在分析、设计和实现间的鸿沟

## 1.6设计模式的优点

①设计模式使人们可以更加简单方便地复用成功的设计，将已证实的技术表述成设计模式会使新系统开发者更加容易理解其设计思路，并避免那些导致不可重用的设计方案。

②设计模式使得设计方案更加灵活，且易于修改。

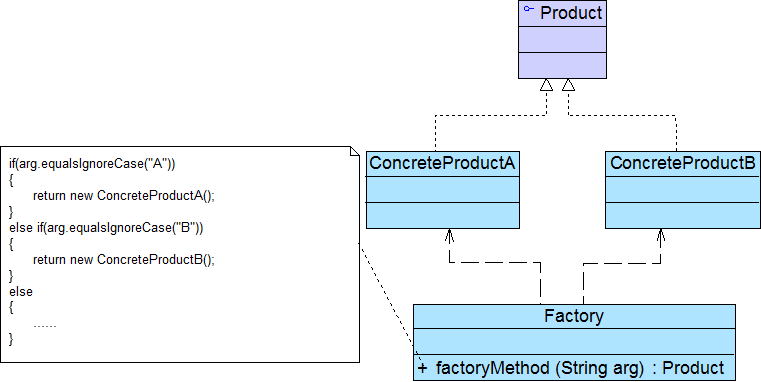
③设计模式的使用将提高软件系统的开发效率和软件质量，且在一定程度上节约设计成本。

④设计模式有助于初学者更深入地理解面向对象思想，一方面可以帮助初学者更加方便地阅读和学习现有类库与其他系统中的源代码，另一方面还可以提高软件的设计水平和代码质量。

# 2.简单工厂模式

又称为静态工厂方法(Static Factory Method)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。

## 2.1模式结构



* Factory：工厂角色，负责实现创建所有实例的内部逻辑
* Product：抽象产品角色，是所创建的所有对象的父类，负责描述所有实例所共有的公共接口
* ConcreteProduct：具体产品角色是创建目标，所有创建的对象都充当这个角色的某个具体类的实例

## 2.2模式分析

有一家生产处理器核的厂家，它只有一个工厂，能够生产两种型号的处理器核。客户需要什么样的处理器核，一定要显示地告诉生产工厂。

enum CTYPE {COREA, COREB};

class SingleCore

{

public:

virtual void Show() = 0;

};

//单核A

class SingleCoreA: public SingleCore

{

public:

void Show() { cout<<"SingleCore A"<<endl; }

};

//单核B

class SingleCoreB: public SingleCore

{

public:

void Show() { cout<<"SingleCore B"<<endl; }

};

//唯一的工厂，可以生产两种型号的处理器核，在内部判断

class Factory

{

public:

SingleCore\* CreateSingleCore(enum CTYPE ctype)

{

if(ctype == COREA) //工厂内部判断

return new SingleCoreA(); //生产核A

else if(ctype == COREB)

return new SingleCoreB(); //生产核B

else

return NULL;

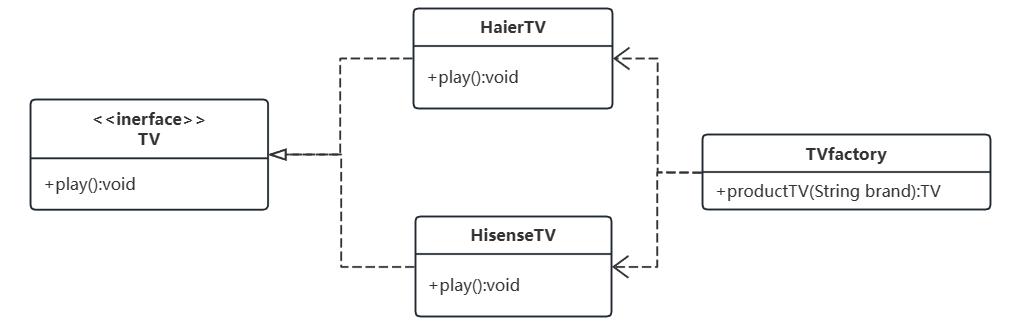
}

};

* 将对象的创建和对象本身业务处理分离可以降低系统的耦合度，使得两者修改起来都相对容易。
* 简单工厂模式最大的问题在于**工厂类的职责相对过重**，增加新的产品需要修改工厂类的判断逻辑，**这一点与开闭原则是相违背的**
* 简单工厂模式的要点在于：当你需要什么，只需要传入一个正确的参数，就可以获取你所需要的对象，而无须知道其创建细节。

## 2.3模式实例与解析

简单电视机工厂



//抽象电视类

class TV

{

public:

virtual void play() = 0;

};

//海尔电视类

class HaierTV :public TV

{

public:

void play() { cout << "海尔电视"; }

};

//海信电视

class HisenseTV :public TV

{

public:

void play() { cout << "海信电视"; }

};

//电视工厂

class Tvfactory

{

public:

static TV\* produceTV(string Brand)

{

if (Brand == "Haire")

return new HaierTV();

else if (Brand == "Hisense")

return new HisenseTV();

else

cout << "不生产该品牌" << endl;

}

};

## 2.4模式优缺点

优点：

1. 工厂类含有必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户端可以免除直接创建产品对象的责任，而仅仅“消费”产品；**简单工厂模式通过这种做法实现了对责任的分割，它提供了专门的工厂类用于创建对象。**
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。**

缺点：

1. 由于**工厂类集中了所有产品创建逻辑**，一旦不能正常工作，整个系统都要受到影响。
2. 使用简单工厂模式将会**增加系统中类的个数**，在一定程序上增加了系统的复杂度和理解难度
3. **系统扩展困难，一旦添加新产品就不得不修改工厂逻辑，在产品类型较多时**，有可能造成工厂逻辑过于复杂，不利于系统的扩展和维护。
4. 简单工厂模式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。**

## 2.5模式使用环境

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

# 3.工厂模式

## 3.1简单工厂模式的不足

简单工厂模式最大的缺点是当有新产品要加入到系统中时，必须修改工厂类，加入必要的处理逻辑，这违背了“开闭原则”。所有的产品都是由同一个工厂创建，工厂类职责较重，业务逻辑较为复杂，具体产品与工厂类之间的耦合度高，严重影响了系统的灵活性和扩展性，而工厂方法模式则可以很好地解决这一问题。

## 3.2模式动机

将具体按钮的创建过程交给专门的工厂子类去完成。定义具体的工厂类来生成圆形按钮、矩形按钮、菱形按钮等，它们实现在抽象按钮工厂类中定义的方法。这种抽象化的结果使这种结构可以在不修改具体工厂类的情况下引进新的产品，如果出现新的按钮类型，只需要为这种新类型的按钮创建一个具体的工厂类就可以获得该新按钮的实例，这一特点无疑使得工厂方法模式具有超越简单工厂模式的优越性，更加符合“开闭原则”。

## 3.3模式定义

虚拟构造器(Virtual Constructor)模式或者多态工厂(Polymorphic Factory)模式。在工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象，这样做的目的是**将产品类的实例化操作延迟到工厂子类中完成，即通过工厂子类来确定究竟应该实例化哪一个具体产品类**。

## 3.4模式结构



* Product：抽象产品是定义产品的接口，是工厂方法模式所创建对象的超类型，是产品对象的共同父类或接口
* ConcreteProduct：具体产品实现了抽象产品接口，某种类型的具体产品由专门的具体工厂创建
* Factory：抽象工厂声明了工厂方法，返回一个产品，与具体应用无关，是具体工厂的共同父类或接口
* ConcreteFactory：具体工厂实现了抽象工厂中的工厂方法，返回一个具体产品类的实例，与应用密切相关。

## 3.5模式分析

**在工厂方法模式中，核心的工厂类不再负责所有产品的创建，而是将具体创建工作交给子类去做。**这个核心类仅仅负责给出具体工厂必须实现的接口，而不负责哪一个产品类被实例化这种细节，这使得**工厂方法模式可以允许系统在不修改工厂角色的情况下引进新产品**。

当系统扩展需要添加新的产品对象时，仅仅需要添加一个具体产品对象以及一个具体工厂对象，原有工厂对象不需要进行任何修改，也不需要修改客户端，**很好地符合了“开闭原则”。**而简单工厂模式在添加新产品对象后不得不修改工厂方法，扩展性不好。**工厂方法模式退化后可以演变成简单工厂模式。**

class SingleCore

{

public:

virtual void Show() = 0;

};

//单核A

class SingleCoreA: public SingleCore

{

public:

void Show() { cout<<"SingleCore A"<<endl; }

};

//单核B

class SingleCoreB: public SingleCore

{

public:

void Show() { cout<<"SingleCore B"<<endl; }

};

class Factory

{

public:

virtual SingleCore\* CreateSingleCore() = 0;

};

//生产A核的工厂

class FactoryA: public Factory

{

public:

SingleCoreA\* CreateSingleCore() { return new SingleCoreA; }

};

//生产B核的工厂

class FactoryB: public Factory

{

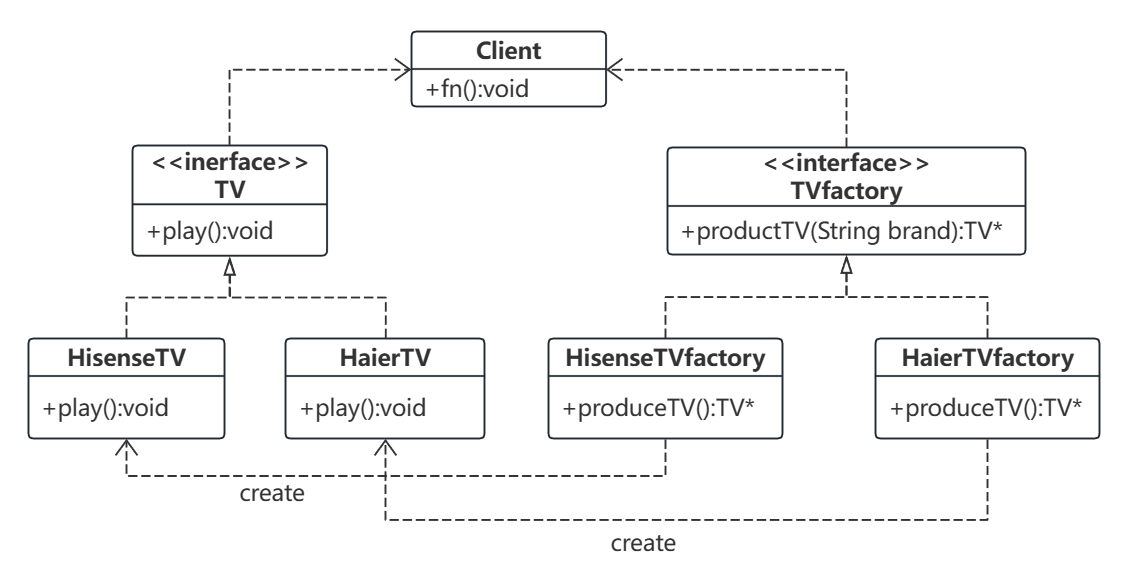
public:

SingleCoreB\* CreateSingleCore() { return new SingleCoreB; }

};

## 3.6模式实例

电视机工厂



//抽象电视类

class TV

{

public:

virtual void play() = 0;

};

//海尔电视类

class HaierTV :public TV

{

public:

void play() { cout << "海尔电视"; }

};

//海信电视

class HisenseTV :public TV

{

public:

void play() { cout << "海信电视"; }

};

//工厂类

class TVfacotry

{

public:

virtual TV\* produceTV() = 0;

};

//生产海尔电视工厂

class HaierTVfactory :public TVfacotry

{

public:

TV\* produceTV()

{

cout << "生产海尔电视" << endl;

return new HaierTV();

}

};

class HisenseTVfactory :public TVfacotry

{

public:

TV\* produceTV()

{

cout << "生产海信电视" << endl;

return new HisenseTV();

}

};

class Client

{

void fn()

{

TV\* tv;

TVfacotry\* TVfactory;

TVfactory = new HisenseTVfactory();

tv = TVfactory->produceTV();

}

};

## 3.7模式优缺点

优点：

* 在工厂方法模式中，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，用户只需要关心所需产品对应的工厂，无须关心创建细节，甚至无须知道具体产品类的类名。
* 基于工厂角色和产品角色的多态性设计是工厂方法模式的关键。它能够使工厂可以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。工厂方法模式之所以又被称为多态工厂模式，是因为所有的具体工厂类都具有同一抽象父类。
* 使用工厂方法模式的另一个优点是在系统中加入新产品时，**无须修改抽象工厂和抽象产品提供的接口，无须修改客户端，也无须修改其他的具体工厂和具体产品，而只要添加一个具体工厂和具体产品就可以了。这样，系统的可扩展性也就变得非常好，完全符合“开闭原则”。**

缺点：

* 在添加新产品时，需要编写新的具体产品类，而且还要提供与之对应的具体工厂类，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，有更多的类需要编译和运行，会给系统带来一些额外的开销。
* 由于考虑到系统的可扩展性，需要引入抽象层，在客户端代码中均使用抽象层进行定义，增加了系统的抽象性和理解难度，且在实现时可能需要用到DOM、反射等技术，增加了系统的实现难度。

## 3.8适用环境

* 一个类不知道它所需要的对象的类：在工厂方法模式中，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，具体的产品对象由具体工厂类创建；客户端需要知道创建具体产品的工厂类。
* 一个类通过其子类来指定创建哪个对象：在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。
* 将创建对象的任务委托给多个工厂子类中的某一个，客户端在使用时可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定，可将具体工厂类的类名存储在配置文件或数据库中。

## 3.9模式扩展

多态性的丧失和模式的退化：

如果工厂仅仅返回一个具体产品对象，便违背了工厂方法的用意，发生退化，此时就不再是工厂方法模式了。一般来说，工厂对象应当有一个抽象的父类型，如果工厂等级结构中只有一个具体工厂类的话，抽象工厂就可以省略，也将发生了退化。当只有一个具体工厂，在具体工厂中可以创建所有的产品对象，并且工厂方法设计为静态方法时，工厂方法模式就退化成简单工厂模式。

## 3.10模式本质

**延迟到子类来选择实现**。

工厂方法模式很好的体现了“依赖倒置原则”。

# 4.抽象工厂模式

## 4.1模式动机

我们需要一个工厂可以提供多个产品对象，而不是单一的产品对象。

### 产品等级结构：

产品等级结构即产品的继承结构，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，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。



当系统所提供的工厂所需生产的具体产品并**不是一个简单的对象**，而是多个位于不同产品等级结构中属于不同类型的具体产品时需要使用抽象工厂模式。

抽象工厂模式是所有形式的工厂模式中最为抽象和最具一般性的一种形态。

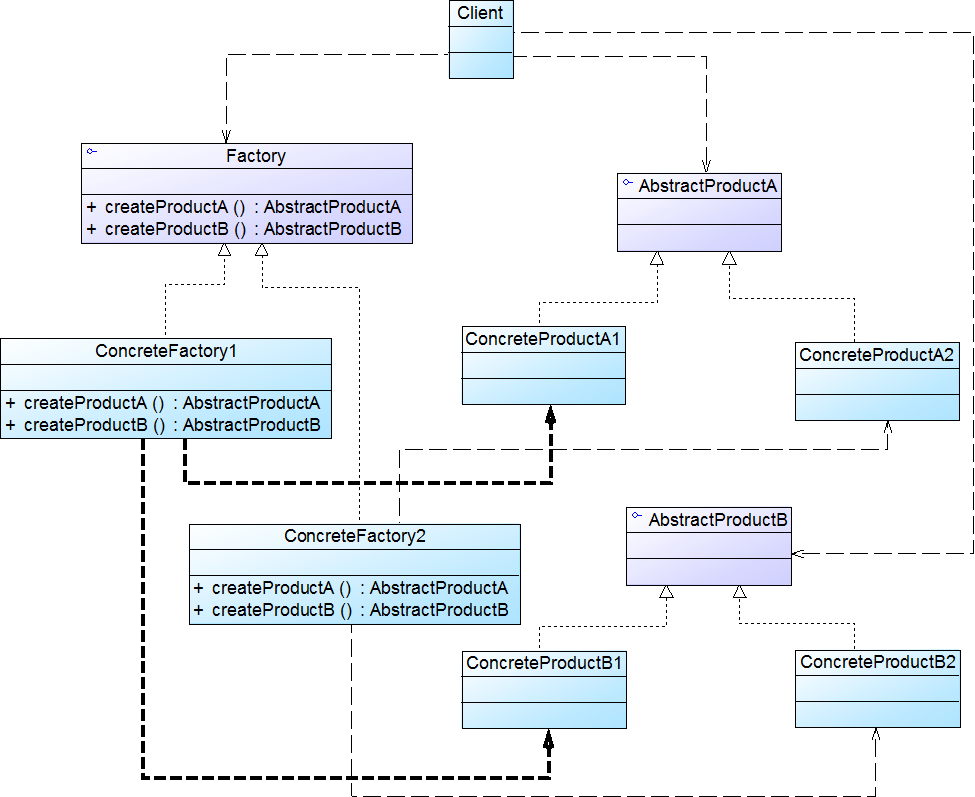
**抽象工厂模式与工厂方法模式最大的区别在于**，工厂方法模式针对的是一个产品等级结构，而抽象工厂模式则需要面对多个产品等级结构，一个工厂等级结构可以负责多个不同产品等级结构中的产品对象的创建 。当一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象时，抽象工厂模式比工厂方法模式更为简单、有效率。



## 4.2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**。

## 4.3模式结构



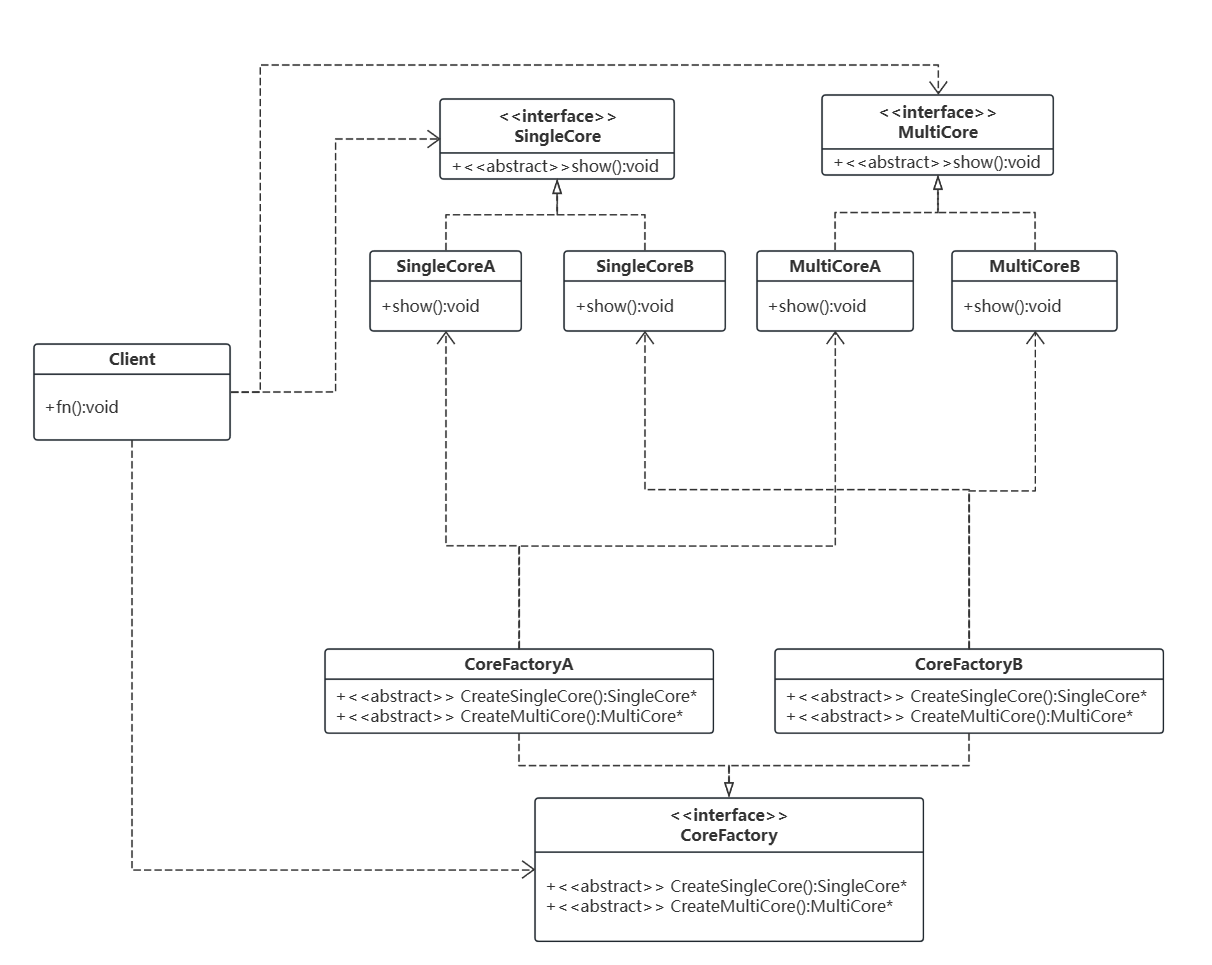
**AbstractFactory**：抽象工厂用于声明生成抽象产品的方法，在一个抽象工厂中可以定义一组方法，每一个方法对应一个产品等级结构

**ConcreteFactory**：具体工厂实现了抽象工厂声明的生成抽象产品的方法，生成一组具体产品，这些产品构成了一个产品族，每一个产品都位于某个产品等级结构中

**AbstractProduct**：抽象产品为每种产品声明接口，在抽象产品中定义了产品的抽象业务方法

**Product**：具体产品定义具体工厂生产的具体产品对象，实现抽象产品接口中定义的业务方法。

## 4.4模式分析



//单核

class SingleCore

{

public:

virtual void Show() = 0;

};

class SingleCoreA: public SingleCore

{

public:

void Show() { cout<<"Single Core A"<<endl; }

};

class SingleCoreB :public SingleCore

{

public:

void Show() { cout<<"Single Core B"<<endl; }

};

//多核

class MultiCore

{

public:

virtual void Show() = 0;

};

class MultiCoreA : public MultiCore

{

public:

void Show() { cout<<"Multi Core A"<<endl; }

};

class MultiCoreB : public MultiCore

{

public:

void Show() { cout<<"Multi Core B"<<endl; }

};

//工厂

class CoreFactory

{

public:

virtual SingleCore\* CreateSingleCore() = 0;

virtual MultiCore\* CreateMultiCore() = 0;

};

//工厂A，专门用来生产A型号的处理器

class FactoryA :public CoreFactory

{

public:

SingleCore\* CreateSingleCore() { return new SingleCoreA(); }

MultiCore\* CreateMultiCore() { return new MultiCoreA(); }

};

//工厂B，专门用来生产B型号的处理器

class FactoryB : public CoreFactory

{

public:

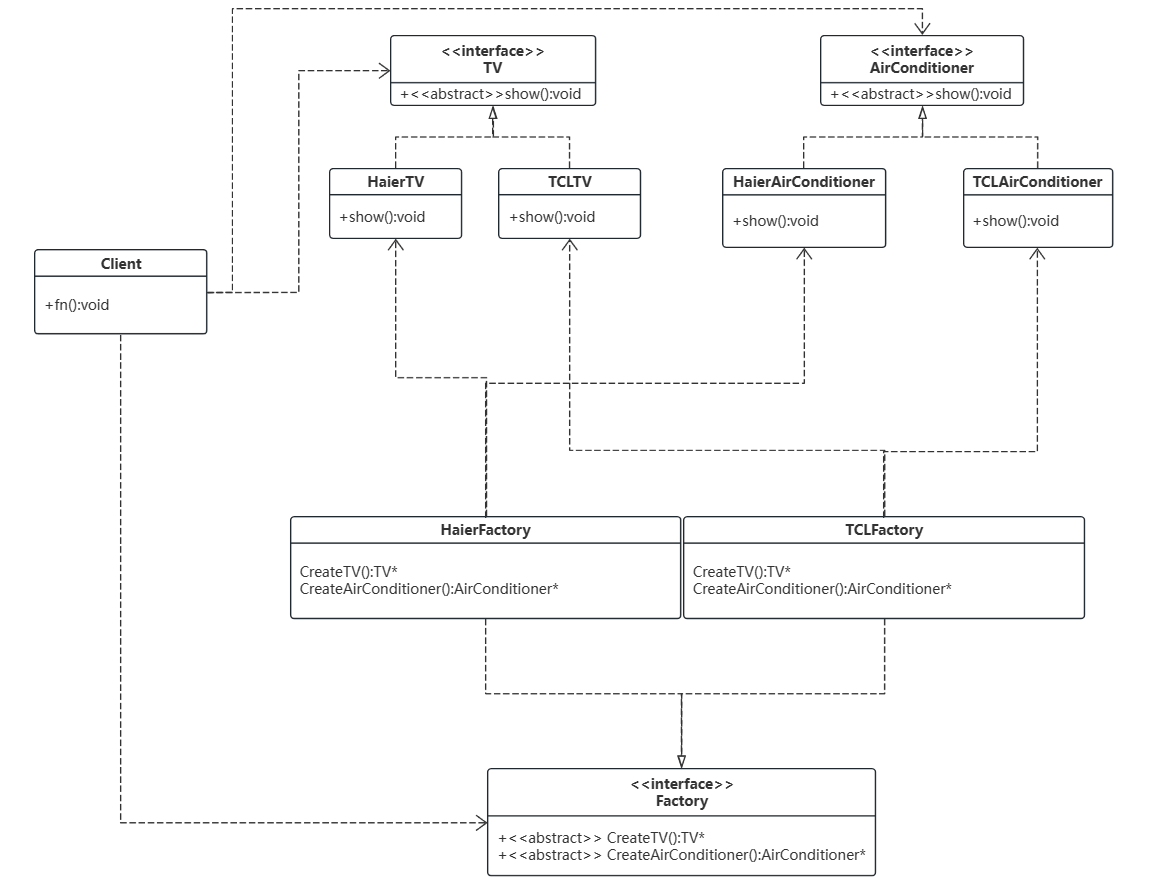
SingleCore\* CreateSingleCore() { return new SingleCoreB(); }

MultiCore\* CreateMultiCore() { return new MultiCoreB(); }

};

## 4.5模式实例

电器工厂



//电视类

class TV

{

public:

virtual void show() = 0;

};

class HaierTV :public TV

{

public:

void show() { cout << "Haier电视" << endl; }

};

class TCL\_TV :public TV

{

public:

void show() { cout << "TCL电视" << endl; }

};

//空调类

class Airconditioner

{

public:

virtual void show() = 0;

};

class HaierAirconditioner :public Airconditioner

{

public:

void show() { cout << "Haier空调" << endl; }

};

class TCLAirconditioner :public Airconditioner

{

public:

void show() { cout << "TCL空调" << endl; }

};

//工厂类

class Factory

{

public:

virtual TV\* CreateTV() = 0;

virtual Airconditioner\* CreateAirConditioner() = 0;

};

class HaierFactory :public Factory

{

TV\* CreateTV()

{

cout << "生产海尔电视" << endl;

return new HaierTV();

}

Airconditioner\* CreateAirConditioner()

{

cout << "生产海尔空调" << endl;

return new HaierAirconditioner();

}

};

class TCLFactory :public Factory

{

TV\* CreateTV()

{

cout << "生产TCL电视" << endl;

return new TCL\_TV();

}

Airconditioner\* CreateAirConditioner()

{

cout << "生产TCL空调" << endl;

return new TCLAirconditioner();

}

};

class Client

{

void fn()

{

TV\* TV;

Airconditioner\* AirConditioner;

Factory\* Factory;

Factory = new TCLFactory();

TV = Factory->CreateTV();

AirConditioner = Factory->CreateAirConditioner();

}

};

## 4.6模式优缺点

优点：

* 抽象工厂模式隔离了具体类的生成，使得客户并不需要知道什么被创建。由于这种隔离，更换一个具体工厂就变得相对容易。所有的具体工厂都实现了抽象工厂中定义的那些公共接口，因此只需改变具体工厂的实例，就可以在某种程度上改变整个软件系统的行为。另外，应用抽象工厂模式可以实现高内聚低耦合的设计目的，因此抽象工厂模式得到了广泛的应用。
* 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。这对一些需要根据当前环境来决定其行为的软件系统来说，是一种非常实用的设计模式。
* 增加新的具体工厂和产品族很方便，无须修改已有系统，符合“开闭原则”。

缺点：

* 在添加新的产品对象时，难以扩展抽象工厂来生产新种类的产品，这是因为在抽象工厂角色中规定了所有可能被创建的产品集合，要支持新种类的产品就意味着要对该接口进行扩展，而这将涉及到对抽象工厂角色及其所有子类的修改，显然会带来较大的不便。
* 开闭原则的倾斜性（增加新的工厂和产品族容易，增加新的产品等级结构麻烦）

## 4.7模式适用环境

* 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，这对于所有类型的工厂模式都是重要的。
* 系统中有多于一个的产品族，而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。
* 系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于具体实现。

## 4.8模式扩展

“开闭原则”的倾斜性：

“开闭原则”要求系统对扩展开放，对修改封闭，通过扩展达到增强其功能的目的。对于涉及到多个产品族与多个产品等级结构的系统，其功能增强包括两方面：

(1) 增加产品族：对于增加新的产品族，工厂方法模式很好的支持了“开闭原则”，对于新增加的产品族，只需要对应增加一个新的具体工厂即可，对已有代码无须做任何修改。

(2) 增加新的产品等级结构：对于增加新的产品等级结构，需要修改所有的工厂角色，包括抽象工厂类，在所有的工厂类中都需要增加生产新产品的方法，不能很好地支持“开闭原则”。

抽象工厂模式的这种性质称为“开闭原则”的倾斜性，抽象工厂模式以一种倾斜的方式支持增加新的产品，它为新产品族的增加提供方便，但不能为新的产品等级结构的增加提供这样的方便。

## 4.9模式本质

* 工厂模式的退化

当抽象工厂模式中每一个具体工厂类只创建一个产品对象，也就是只存在一个产品等级结构时，抽象工厂模式退化成工厂方法模式；当工厂方法模式中抽象工厂与具体工厂合并，提供一个统一的工厂来创建产品对象，并将创建对象的工厂方法设计为静态方法时，工厂方法模式退化成简单工厂模式。

* 抽象工厂模式的本质是：**选择产品簇的实现**。
* 工厂方法是选择单个产品的实现，虽然一个类里面可以有多个工厂方法，但是这些方法之间一般是没有联系的，即使看起来像有联系。
* 但是抽象工厂着重的就是为一个产品簇选择实现，定义在抽象工厂里面的方法通常是有联系的，它们是产品的某一部分或者相互依赖的。如果抽象工厂里面只定义一个方法，直接创建产品，那么就退化称为工厂方法了。

## 4.10模式应用

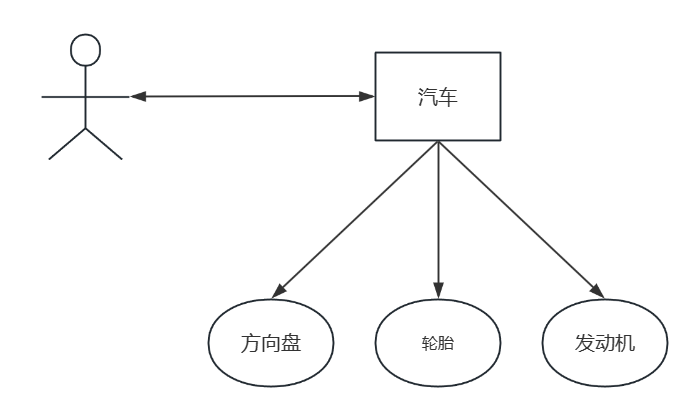
1. Java SE AWT（抽象窗口工具包）

2. 在很多软件系统中需要更换界面主题，要求界面中的按钮、文本框、背景色等一起发生改变时，可以使用抽象工厂模式进行设计。

# 5.建造者模式

## 5.1模式动机

**建造者模式可以将部件和其组装过程分开，一步一步创建一个复杂的对象**。用户只需要指定复杂对象的类型就可以得到该对象，而无须知道其内部的具体构造细节。



在软件开发中，也存在大量类似汽车一样的复杂对象，**它们拥有一系列成员属性，这些成员属性中有些是引用类型的成员对象**。而且在这些复杂对象中，还可能存在一些限制条件，如某些属性没有赋值则复杂对象不能作为一个完整的产品使用；有些属性的赋值必须按照某个顺序，一个属性没有赋值之前，另一个属性可能无法赋值等。

复杂产品对象相当于一辆有待建造的汽车，而对象的属性相当于汽车的部件，建造产品的过程就相当于组合部件的过程。由于组合部件的过程很复杂，因此，这些部件的组合过程往往被“外部化”到一个称作建造者的对象里，**建造者返还给客户端的是一个已经建造完毕的完整产品对象**，而用户无须关心该对象所包含的属性以及它们的组装方式，这就是建造者模式的模式动机。

## 5.2模式定义

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**。

建造者模式是一步一步创建一个复杂的对象，**它允许用户只通过指定复杂对象的类型就可以构建它们，用户不需要知道内部的具体构建细节**。建造者模式属于对象创建型模式。根据中文翻译的不同，建造者模式又可以称为**生成器模式**。

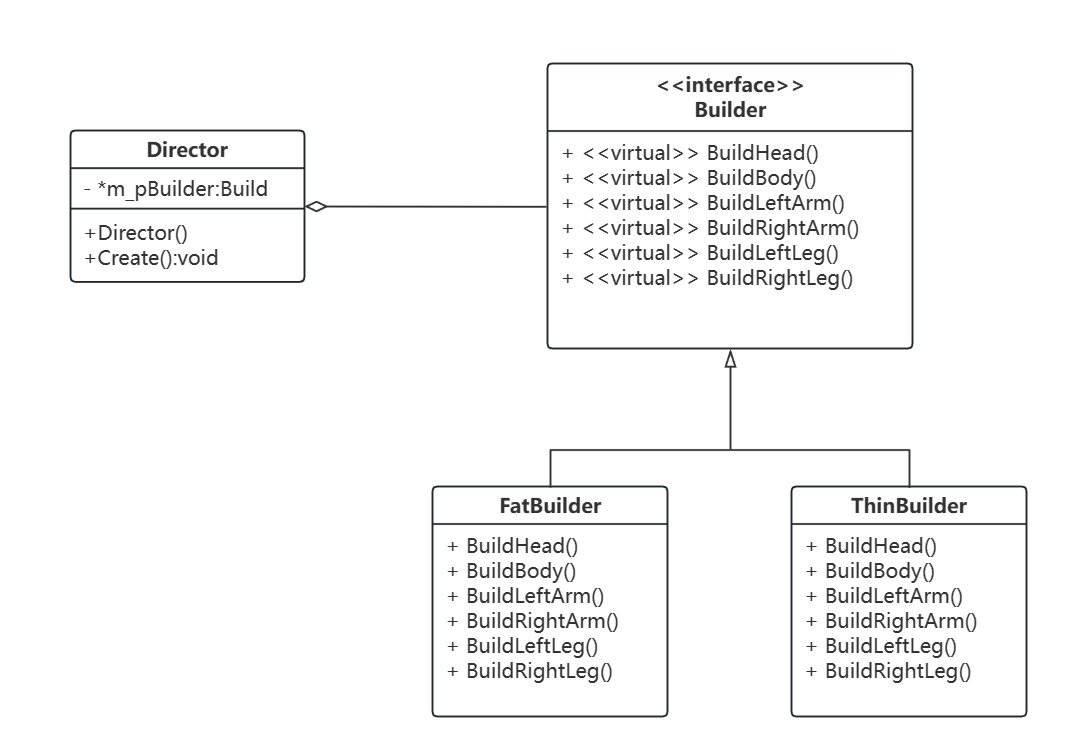
## 5.4模式结构



* Builder：抽象建造者为创建一个产品对象的各个部件指定抽象接口
* ConcreteBuilder：具体建造者实现了抽象建造者接口，实现各个部件的构造和装配方法，定义并明确它所创建的复杂对象，也可以提供一个方法返回创建好的复杂产品对象
* Director：指挥者负责安排复杂对象的建造次序，指挥者与抽象建造者之间存在关联关系，可以在其construct方法中调用建造者对象的部件构造与装配方法，完成复杂对象的建造
* Product：产品角色是被构建的复杂对象，包含多个组成部件

## 5.5模式分析

对于客户来说，只需知道导向者就可以了，通过导向者，客户就能构造复杂的对象，而不需要知道具体的构造过程。下面给出小人例子的代码实现。



class Builder

{

public:

virtual void BuildHead() {}

virtual void BuildBody() {}

virtual void BuildLeftArm(){}

virtual void BuildRightArm() {}

virtual void BuildLeftLeg() {}

virtual void BuildRightLeg() {}

};

//构造瘦人

class ThinBuilder : public Builder

{

public:

void BuildHead() { cout<<"build thin body"<<endl; }

void BuildBody() { cout<<"build thin head"<<endl; }

void BuildLeftArm() { cout<<"build thin leftarm"<<endl; }

void BuildRightArm() { cout<<"build thin rightarm"<<endl; }

void BuildLeftLeg() { cout<<"build thin leftleg"<<endl; }

void BuildRightLeg() { cout<<"build thin rightleg"<<endl; }

};

//构造胖人

class FatBuilder : public Builder

{

public:

void BuildHead() { cout<<"build fat body"<<endl; }

void BuildBody() { cout<<"build fat head"<<endl; }

void BuildLeftArm() { cout<<"build fat leftarm"<<endl; }

void BuildRightArm() { cout<<"build fat rightarm"<<endl; }

void BuildLeftLeg() { cout<<"build fat leftleg"<<endl; }

void BuildRightLeg() { cout<<"build fat rightleg"<<endl; }

};

//构造的指挥官

class Director

{

private:

Builder \*m\_pBuilder;

public:

Director(Builder \*builder) { m\_pBuilder = builder; }

void Create(){

m\_pBuilder->BuildHead();

m\_pBuilder->BuildBody();

m\_pBuilder->BuildLeftArm();

m\_pBuilder->BuildRightArm();

m\_pBuilder->BuildLeftLeg();

m\_pBuilder->BuildRightLeg();

}

};

int main()

{

FatBuilder fat;

Director director(&fat);

director.Create();

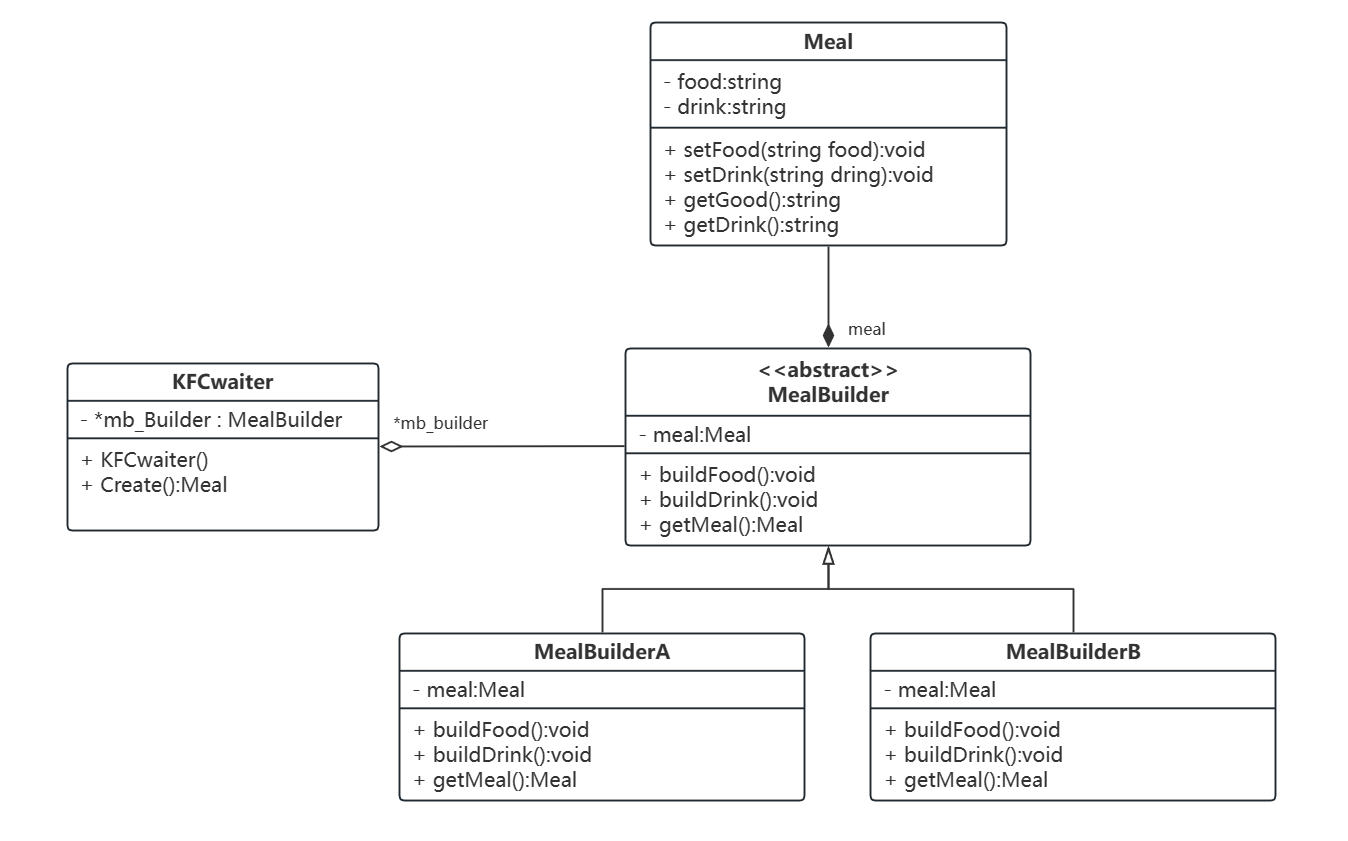
return 0;

}

## 5.6模式实例

KFC套餐

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。



class MealBuilder;

class Meal;

class KFCwaiter;

//最后得到的产品 Meal

class Meal

{

public:

void setFood(string food) { this->food = food; }

void setDrink(string drink) { this->drink = drink; }

string getFood() { return food; }

string getDrink() { return drink; }

private:

string food;

string drink;

};

//Builder抽象类

class MealBuilder

{

public:

virtual void buildFood() = 0;

virtual void buildDrink() = 0;

Meal getMeal() { return meal; }

protected:

Meal meal;

};

//具体BuilderA类

class MealBuilderA:public MealBuilder

{

void buildFood() { meal.setFood("香辣鸡腿堡"); }

void buildDrink() { meal.setDrink("可乐"); }

};

//具体BuilderB类

class MealBuilderB :public MealBuilder

{

void buildFood() { meal.setFood("奥尔良鸡腿堡"); }

void buildDrink() { meal.setDrink("雪碧"); }

};

//指挥者类

class KFCwaiter

{

public:

KFCwaiter(MealBuilder\* mb) { mb\_Builder = mb; }

Meal Create()

{

mb\_Builder->buildFood();

mb\_Builder->buildDrink();

return mb\_Builder->getMeal();

}

private:

MealBuilder\* mb\_Builder;

};

int main()

{

MealBuilderA mbA;

KFCwaiter kfcWaiter(&mbA);

Meal meal = kfcWaiter.Create();

cout << meal.getFood();

cout << meal.getDrink();

}

## 5.7模式优缺点

### 优点

* 在建造者模式中，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，而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，**用户使用不同的具体建造者即可得到不同的产品对象**。
* 可以更加精细地控制产品的创建过程。将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰，也更方便使用程序来控制创建过程。
* **增加新的具体建造者无须修改原有的代码，指挥者类针对抽象建造者类编程，系统扩展方便，符合“开闭原则”。**

### 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，则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，可能会导致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。

## 5.8模式适用环境

* **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属性**。
* 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对象。

## 5.9模式应用

(1) JavaMail（一步一步构造一个完整的邮件对象，然后发送）

(2) 在很多游戏软件中，地图包括天空、地面、背景等组成部分，人物角色包括人体、服装、装备等组成部分，可以使用建造者模式对其进行设计，通过不同的具体建造者创建不同类型的地图或人物。

## 5.10模式扩展

建造者模式的简化

* 省略抽象建造者角色：如果系统中只需要一个具体建造者的话，可以省略掉抽象建造者。
* 省略指挥者角色：在具体建造者只有一个的情况下，如果抽象建造者角色已经被省略掉，那么还可以省略指挥者角色。
* 把指挥者类和抽象建造者进行合并，简化了系统结构，但同时也加重了抽象建造者类的职责，也不符合单一职责原则，如果construct()过于复杂，建议还是封装到指挥者类中。

## 5.11建造者模式与抽象工厂模式的对比

* 与抽象工厂模式相比，**建造者模式返回一个组装好的完整产品，而抽象工厂模式返回一系列相关的产品**，这些产品位于不同的产品等级结构，构成了一个产品族。
* 在抽象工厂模式中，客户端实例化工厂类，然后调用工厂方法获取所需产品对象，而在建造者模式中，客户端可以不直接调用建造者的相关方法，而是通过指挥者类来指导如何生成对象，包括对象的组装和建造过程，它侧重于一步步构造一个复杂对象，返回一个完整的对象。
* 如果将抽象工厂模式看成汽车配件生产工厂，生产一个产品族的产品，那么建造者模式就是一个汽车组装工厂，通过对部件的组装可以返回一辆完整的汽车。

## 5.12模式本质

建造者模式的本质：**分离整体构建算法和部件构造。**

构建一个复杂的对象，本来就有构建的过程，以及构建过程中具体的实现，建造者模式就是用来分离这两个部分，从而使得程序结构更松散、扩展更容易、复用性更好，同时也会使得代码更清晰，意图更明确。

# 6.原型模式

## 6.1模式动机

* 在面向对象系统中，使用原型模式来复制一个对象的自身，从而克隆出多个与原型对象一模一样的对象。
* 在软件系统中，有些对象的创建过程较为复杂，而且有时候需要频繁创建，原型模式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，更方便、更高效且节省资源，这就是原型模式的意图所在。
* 对比拷贝构造，通过原型对象创建新的对象，就不再需要关心/知道这个对象本身的类型，如复印机。
* 对比拷贝构造，原型模式有助于符合里氏替换原则，如水果篮里放入水果副本。

## 6.2模式定义

**原型模式(Prototype Pattern)：**原型模式是一种对象创建型模式，用原型实例指定创建对象的种类，并且通过复制这些原型创建新的对象。原型模式允许一个对象再创建另外一个可定制的对象，无须知道任何创建的细节。

原型模式的基本工作原理是通过将一个原型对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型对象拷贝原型自己来实现创建过程。

## 6.3模式结构



Prototype：抽象原型类是定义具有克隆自己的方法的接口

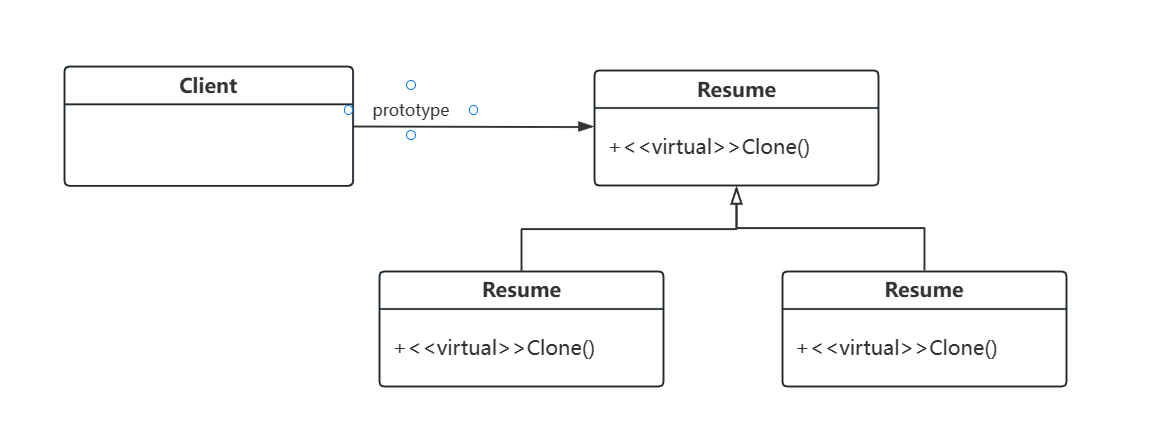
ConcretePrototype：具体原型类实现具体的克隆方法，在克隆方法中返回自己的一个克隆对象

Client：在客户类中只需要直接实例化或通过工厂方法等方式创建一个对象，再通过调用该对象的克隆方法复制得到多个相同的对象

## 6.4模式分析

我们只需手写一份，然后利用打印设备复印多份即可。如果要修改简历中的某项，那么修改原始的版本就可以了，然后再复印。原始的那份手写稿相当于是一个原型，有了它，就可以通过复印（拷贝）创造出更多的新简历。

Clone函数即C++拷贝构造函数。



class Resume

{

protected:

char\* name;

public:

Resume() { };

virtual ~Resume() { };

virtual Resume\* Clone() { return NULL; }

virtual void Set(string name) { };

virtual void Show() { };

};

class ResumeA :public Resume

{

public:

ResumeA(const char\* str);

ResumeA(const ResumeA& r);

~ResumeA();

ResumeA\* Clone();

void Show();

};

ResumeA::ResumeA(const char\* str)

{

if (str == NULL)

{

name = new char[1];

name[0] = '\0';

}

else

{

name = new char[strlen(str) + 1];

strcpy(name, str);

}

}

ResumeA::ResumeA(const ResumeA& r)

{

name = new char[strlen(r.name) + 1];

strcpy(name, r.name);

}

ResumeA::~ResumeA()

{

delete[] name;

}

ResumeA\* ResumeA::Clone()

{

return new ResumeA(\*this);

}

void ResumeA::Show()

{

cout << "ResumeA:" << name << endl;

}

int main()

{

ResumeA\* r1 = new ResumeA("A");

ResumeA\* r2 = new ResumeA("B");

ResumeA\* r3 = r1->Clone();

ResumeA\* r4 = r2->Clone();

r1->Show();

r2->Show();

delete r1, r2;

r1 = r2 = nullptr;

r3->Show();

r4->Show();

delete r3, r4;

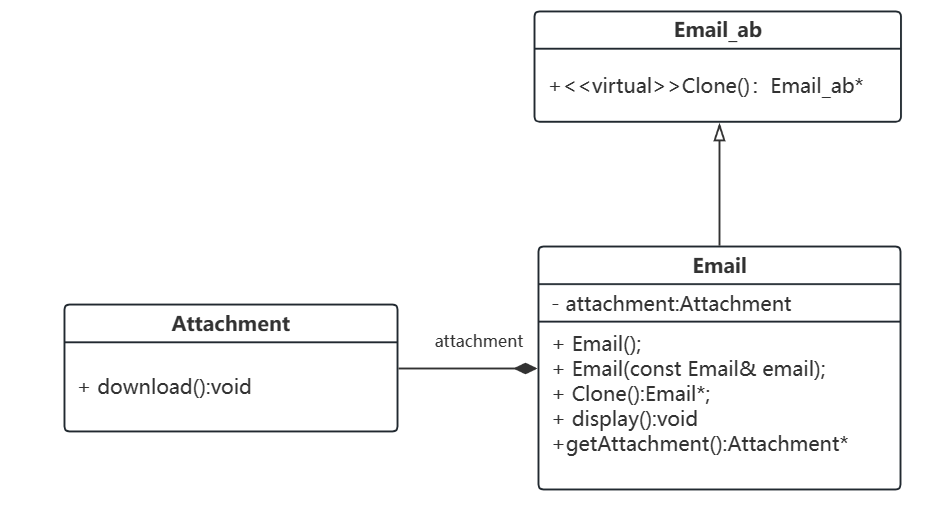
r3 = r4 = nullptr;

}

## 6.5 模型实例

邮件复制（浅拷贝）

不复制附件

class Attachment

{

public:

void download() { cout << "download Attachment" << endl; }

};

class Email\_ab

{

public:

Email\_ab() {};

virtual ~Email\_ab() {};

virtual Email\_ab\* clone() { return nullptr; };

virtual void display() {};

virtual Attachment\* getAttachment() { return nullptr; };

protected:

Attachment \*attachment;

};

class Email :public Email\_ab

{

public:

Email() = default;

Email(Attachment& attachment);

Email(const Email& email);

~Email() { delete attachment; }

Email \* clone();

void display();

Attachment\* getAttachment();

};

Email::Email(Attachment& attachment)

{

this->attachment = new Attachment();

\*this->attachment = attachment;

}

Email::Email(const Email& email)

{

attachment = email.attachment;

}

Email\* Email::clone()

{

return new Email(\*this);

}

void Email::display()

{

cout << "查看邮件" << endl;

}

Attachment\* Email::getAttachment()

{

return attachment;

}

int main()

{

Attachment attachment;

Email\* email = new Email(attachment);

Email\* copy\_email;

copy\_email = email->clone();

cout << (email == copy\_email) << endl;

cout << (email->getAttachment() == copy\_email->getAttachment()) << endl;

}

邮件复制（深拷贝）

复制附件

Email::Email(const Email& email)

{

attachment = new Attachment();

\*attachment = \*email.attachment;

}

## 6.6模式优缺点

优点：

* 当创建新的对象实例较为复杂时，使用原型模式可以简化对象的创建过程，通过一个已有实例可以提高新实例的创建效率。
* 扩展性较好，由于在原型模式中提供了抽象原型类，在客户端可以针对抽象原型类进行编程，而将具体原型类写在配置文件中，增加或减少产品类对原有系统都没有任何影响。
* 原型模式提供了简化的创建结构，工厂方法模式常常需要有一个与产品类等级结构相同的工厂等级结构，而原型模式就不需要这样，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，使用原型模式将对象复制一份并将其状态保存起来，以便在需要的时候使用（如恢复到某一历史状态），可辅助实现撤销操作。

缺点：

* 需要为每一个类配备一个克隆方法，而且这个克隆方法需要对类的功能进行通盘考虑，这对全新的类来说不是很难，但对已有的类进行改造时，不一定是件容易的事，必须修改其源代码，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。

## 6.7模式适用环境

* 需要避免创建一个与产品类层次平行的工厂类层次时，并且类的实例对象只能有几个不同状态组合中的一种时，建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。
* 比如在画图工具里，要画圆只需要拖动工具条的画圆工具到绘图区即可，而不需要从头开始一点一点的画一个圆，而且如果需要不同大小和颜色的圆，只需要复制几个圆，然后再修改他们的大小和颜色即可。

## 6.8模式应用

（1）复制粘贴

（2）在Struts2中为了保证线程的安全性，**Action对象的创建使用了原型模式。**

（3）在Spring中，用户也可以**采用原型模式来创建新的bean实例**。

## 6.9模式扩展

1. 带原型管理器的原型模式

2. 相似对象的复制

## 6.10模式本质

原型模式的本质：**克隆生成对象**。

克隆是手段，目的还是生成新的对象实例。正是因为原型的目的是为了生成新的对象实例，原型模式通常是被归类为创建型的模式。

# 7.单例模式

## 7.1.模式动机

**对于系统中的某些类来说，只有一个实例很重要**，例如，一个系统只有一个文件系统、一个任务管理器。

**定义一个全局变量可以确保对象随时都可以被访问，但不能防止我们实例化多个对象**。

**一个更好的解决办法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被创建，并且它可以提供一个访问该实例的方法。这就是单例模式的模式动机。

## 7.2.模式定义

**单例模式**(Singleton Pattern)：单例模式**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。

单例模式有三个要点：一是**某个类只能有一个实例**；二是**它必须自行创建这个实例**；三是它必须**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。单例模式又名单件模式或单态模式。

## 7.3模式分析

class Singleton

{

public:

static Singleton\* GetInstance();

private:

Singleton()=default;

static Singleton\* singleton;

};

Singleton\* Singleton::singleton = nullptr;

Singleton\* Singleton::GetInstance()

{

if (singleton == nullptr)

singleton = new Singleton();

return singleton;

}

int main()

{

Singleton\* a = Singleton::GetInstance();

Singleton\* b = Singleton::GetInstance();

cout << (a == b) << endl;

}

## 7.4模式实例

身份证号码

class IdentityCardNo

{

public:

static IdentityCardNo\* getInstance();

string getIdentityCardNo();

private:

string No;

static IdentityCardNo\* instance;

IdentityCardNo() {};

void setIdentityCardNo(string N0);

};

IdentityCardNo\* IdentityCardNo::instance = nullptr;

IdentityCardNo\* IdentityCardNo::getInstance()

{

if (instance == nullptr)

{

cout << "first time" << endl;

instance = new IdentityCardNo();

instance->setIdentityCardNo("1111111111");

}

else

cout << "you have made a IDcard";

return instance;

}

## 7.5模式优缺点

优点：

* 提供了对唯一实例的受控访问。因为单例类封装了它的唯一实例，所以它可以严格控制客户怎样以及何时访问它，并为设计及开发团队提供了共享的机制。
* 由于在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销毁的对象，单例模式无疑可以提高系统的性能。
* 允许可变数目的实例。我们可以基于单例模式进行扩展，使用与单例控制相似的方法来获得指定个数的对象实例。

缺点：

* 由于单例模式中没有抽象层，因此单例类的扩展有很大的困难。
* 单例类的职责过重，在一定程度上违背了“单一职责原则”。因为单例类既充当了工厂角色，提供了工厂方法，同时又充当了产品角色，包含一些业务方法，将产品的创建和产品的本身的功能融合到一起。
* 滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出。

## 7.6模式适用环境

* 系统只需要一个实例对象，如系统要求提供一个唯一的序列号生成器，或者需要考虑资源消耗太大而只允许创建一个对象。
* 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。
* 在一个系统中要求一个类只有一个实例时才应当使用单例模式。反过来，如果一个类可以有几个实例共存，就需要对单例模式进行改进，使之成为多例模式。

## 7.7模式应用

(1) java.lang.Runtime类

(2)数据库主键生成器

（3）默认情况下Spring会通过单例模式创建bean实例

## 7.8模式扩展

懒汉式



饿汉式



## 7.9模式本质

**控制实例数目。**

# 结构型模式

## 结构性模式概述

结构型模式(Structural Pattern)**描述如何将类或者对象结合在一起形成更大的结构**，就像搭积木，可以通过简单积木的组合形成复杂的、功能更为强大的结构。

结构型模式可以分为**类结构型模式**和**对象结构型模式**：

* 类结构型模式关心类的组合，由多个类可以组合成一个更大的系统，在类结构型模式中一般只存在继承关系和实现关系。
* 对象结构型模式关心类与对象的组合，通过关联关系使得在一个类中定义另一个类的实例对象，然后通过该对象调用其方法。根据“合成复用原则”，在系统中尽量使用关联关系来替代继承关系，因此大部分结构型模式都是对象结构型模式。

## 结构型模式简介

适配器模式(Adapter)

桥接模式(Bridge)

组合模式(Composite)

装饰模式(Decorator)

外观模式(Facade)

享元模式(Flyweight)

代理模式(Proxy)

# 8.适配器模式

## 8.1模式动机

通常情况下，**客户端可以通过目标类的接口访问它所提供的服务**。有时，现有的类可以满足客户类的功能需要，但是它所提供的接口不一定是客户类所期望的，这可能是因为现有类中方法名与目标类中定义的方法名不一致等原因所导致的。

在这种情况下，现有的接口需要转化为客户类期望的接口，这样保证了对现有类的重用。如果不进行这样的转化，客户类就不能利用现有类所提供的功能，**适配器模式可以完成这样的转化**。

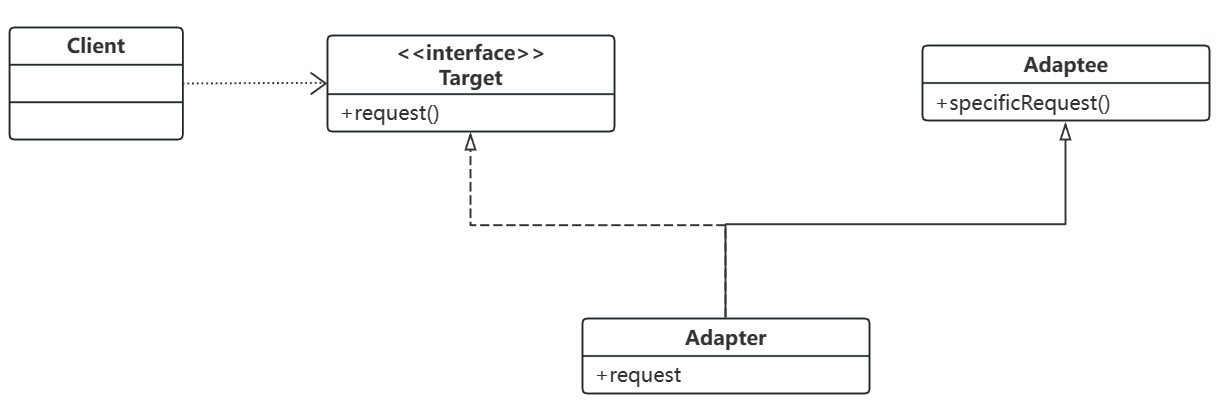
在适配器模式中可以定义一个包装类，包装不兼容接口的对象，这个包装类指的就是**适配器(Adapter)**，它所包装的对象就是适**配者者(Adaptee)**，即被适配的类。

适配器提供客户类需要的接口，**适配器的实现就是把客户类的请求转化为对适配者的相应接口的调用**。也就是说：**当客户类调用适配器的方法时，在适配器类的内部将调用适配者类的方法，而这个过程对客户类是透明的，客户类并不直接访问适配者类。**因此，**适配器可以使由于接口不兼容而不能交互的类可以一起工作**。这就是适配器模式的模式动机。

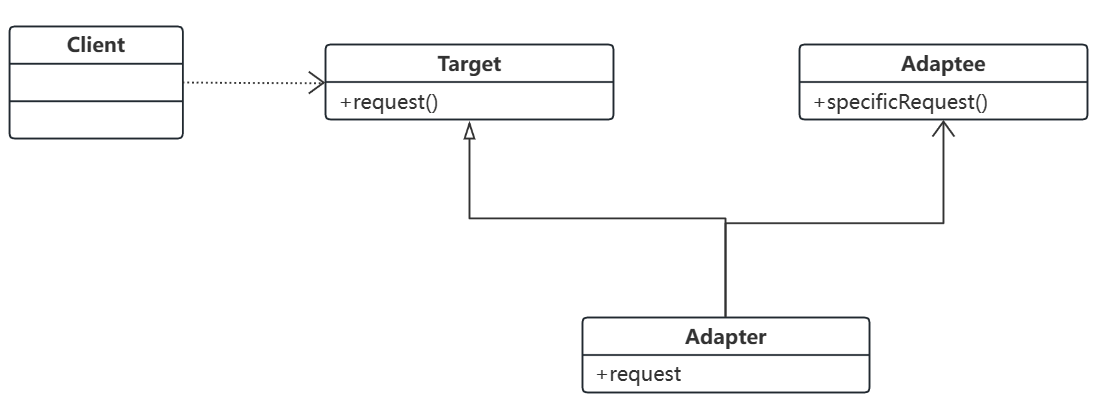
## 8.2模式定义

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，其别名为包装器(Wrapper)。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。

类适配器：



对象适配器：



Target：目标抽象类定义客户所需接口，可以是一个抽象类或接口，也可以是具体类

Adapter：适配器类可以调用另一个接口，作为一个转换器，对Adaptee和Target进行适配，适配器类是适配器模式的核心，在对象适配器中，它通过继承Target并关联一个Adaptee对象使二者产生联系

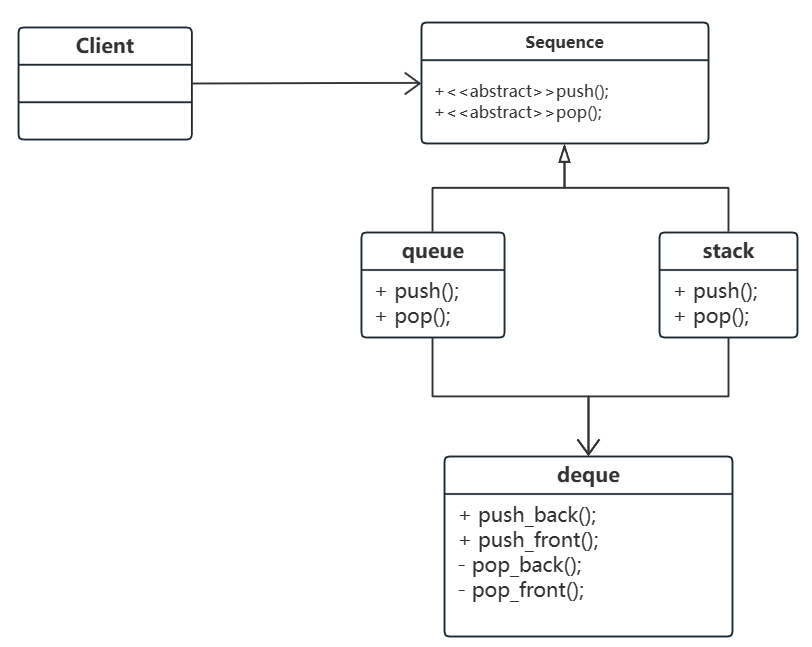
Adaptee：适配者类即被适配的角色，它定义了一个已经存在的接口，这个接口需要适配，适配者类一般是一个具体类，包含了客户希望使用的业务方法，在某些情况下可能没有适配者类的源代码

Client：客户类

## 8.3模式分析

对象适配器：

STL中栈和队列是用deque实现的。



//适配者

class Deque

{

public:

void push\_back(int x) { cout << "Deque push\_back" << endl; }

void push\_front(int x) { cout << "Deque push\_front" << endl; }

void pop\_back() { cout << "Deque pop\_back" << endl; }

void pop\_front() { cout << "Deque pop\_front" << endl; }

};

//适配器类

class Sequence

{

public:

virtual void push(int x) = 0;

virtual void pop() = 0;

};

class Stack :public Sequence

{

public:

void push(int x) { deque.push\_back(x); }

void pop() { deque.pop\_back(); }

private:

Deque deque;

};

class deque :public Sequence

{

public:

void push(int x) { deque.push\_back(x); }

void pop() { deque.pop\_front(); }

private:

Deque deque;

};

int main()

{

Sequence\* s1 = new Stack();

Sequence\* s2 = new deque();

s1->push(1); s1->pop();

s2->push(1); s2->pop();

delete s1, s2;

return 0;

}

类适配器：

现需要设计一个可以模拟各种动物行为的机器人，在机器人中定义了一系列方法，如机器人叫喊方法cry()、机器人移动方法move()等。如果希望在不修改已有代码的基础上使得机器人能够像狗一样叫，像狗一样跑，使用适配器模式进行系统设计。

//目标类

class Robot

{

public:

virtual void cry() = 0;

virtual void run() = 0;

};

//适配者类

class Dog

{

public:

void wang() { cout << "dog wang" << endl; }

void run() { cout << "dog run" << endl; }

};

class Bird

{

public:

void tweedle() { cout << "bird tweedle" << endl; }

void fly() { cout << "bird fly" << endl; }

};

//适配器类

class dogAdapter :public Robot,Dog

{

public:

void cry() { cout << "robot like:"; Dog::wang(); }

void run() { cout << "robot like:"; Dog::run(); }

};

class birdAdapter :public Robot, Bird

{

public:

void cry() { cout << "robot like:"; Bird::tweedle(); }

void run() { cout << "robot like:"; Bird::fly(); }

};

int main()

{

Robot\* dog = new dogAdapter();

Robot\* bird = new birdAdapter();

dog->cry(); dog->run();

bird->cry(); bird->run();

return 0;

}

## 8.4模式优缺点

优点：

* **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，而无须修改原有代码。
* **增加了类的透明性和复用性**，将具体的实现封装在适配者类中，对于客户端类来说是透明的，而且提高了适配者的复用性。
* **灵活性和扩展性都非常好**，通过使用配置文件，可以很方便地更换适配器，也可以在不修改原有代码的基础上增加新的适配器类，完全符合“开闭原则”。

### 类适配器

优点：

由于适配器类是适配者类的子类，因此可以在适配器类中置换一些适配者的方法，使得适配器的灵活性更强。

缺点：

对于Java、C#等不支持多重继承的语言，一次最多只能适配一个适配者类，而且目标抽象类只能为抽象类，不能为具体类，**其使用有一定的局限性**，不能将一个适配者类和它的子类都适配到目标接口。

### 对象适配器

优点：

一个对象适配器可以把多个不同的适配者适配到同一个目标，也就是说，**同一个适配器可以把适配者类和它的子类都适配到目标接口**。

缺点：

与类适配器模式相比，**要想置换适配者类的方法就不容易**。如果一定要置换掉适配者类的一个或多个方法，就只好先做一个适配者类的子类，将适配者类的方法置换掉，然后再把适配者类的子类当做真正的适配者进行适配，实现过程较为复杂.

## 8.6适用环境

* 系统需要使用现有的类，而这些类的接口不符合系统的需要。
* 想要建立一个可以重复使用的类(适配器类)，用于与一些彼此之间没有太大关联的一些类(目标类和适配者类)，包括一些可能在将来引进的类(适配者类的子类)一起工作。

## 8.7模式应用

(1) JDBC

(2) 在JDK类库中也定义了一系列适配器类，如在com.sun.imageio.plugins.common包中定义的InputStreamAdapter类，用于包装ImageInputStream接口及其子类对象。

## 8.8模式本质

转换匹配，复用功能。

# 9.桥接模式

## 9.1模式动机

假设要开发一个软件，绘制矩形、圆形、椭圆、正方形，我们至少需要4个形状类，同时如果绘制的图形还需要填充不同的颜色，如白色、灰色、黑色等，此时至少有如下两种设计方案：

* 第一种设计方案是为每一种形状都提供一套各种颜色的版本。
* 第二种设计方案是根据实际需要对形状和颜色进行组合。

方案一扩展麻烦，无论是增加新的图形还是新的颜色，都需要增加大量的具体类，导致系统变得非常庞大，增大运行和维护开销。究其原因，是将图形的形状和颜色两种职责集中在一个具体类中，违反了“单一职责原则”，任意一个职责发生改变都需要一个新类。

方案二采用关联关系解耦，类个数更少，扩展更方便。

## 9.2模式定义

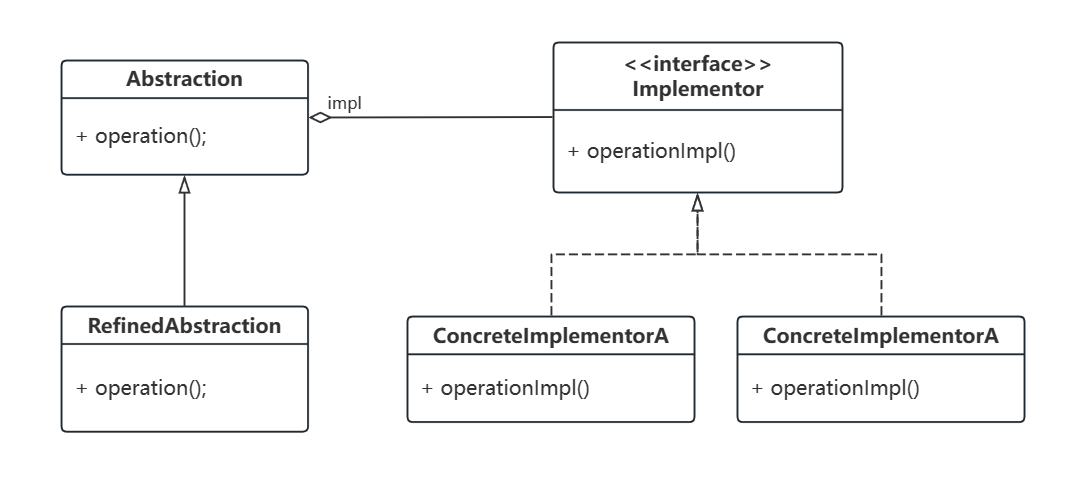
**桥接模式**(Bridge Pattern)：将抽象部分与它的实现部分分离，使它们都可以独立地变化。它是一种对象结构型模式，又称为柄体(Handle and Body)模式或接口(Interface)模式。

如果软件系统中某个类存在两个独立变化的维度，通过桥接模式可以将这两个维度分离出来，使两者可以独立扩展，让系统更加符合“单一职责原则”。

与多重/多层继承方案不同，桥接模式将两个独立变化的维度设计为两个独立的继承等级结构，并且在抽象层建立一个抽象关联，该关联关系类似一条连接两个独立继承结构的桥，故名桥接模式。

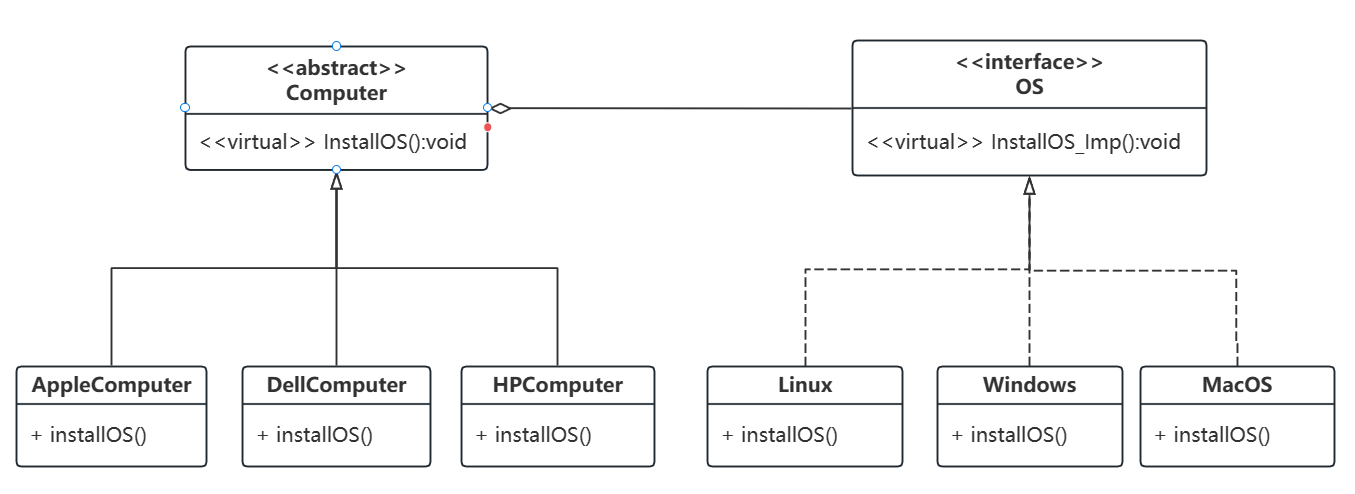
桥接模式用抽象关联取代了传统的多重/多层继承，将类之间的静态继承关系转换为动态的对象组合关系。

## 9.3模式结构



## 9.4模式分析

可以将操作系统和计算机分别抽象出来，让它们各自发展，减少它们的耦合度。当然了，两者之间有标准的接口。



class OS

{

public:

virtual void InstallOS\_Imp() = 0;

};

class Windows :public OS

{

public:

void InstallOS\_Imp() { cout << "Install Windows" << endl; }

};

class Linux :public OS

{

public:

void InstallOS\_Imp() { cout << "Install Linux" << endl; }

};

class MacOS :public OS

{

public:

void InstallOS\_Imp() { cout << "Install MacOS" << endl; }

};

class Computer

{

public:

virtual void InstallOS(OS\* os) {};

};

class DellCompuer:public Computer

{

public:

void InstallOS(OS\* os) { cout << "Dell"; os->InstallOS\_Imp(); };

};

class HPCompuer :public Computer

{

public:

void InstallOS(OS\* os) { cout << "HP"; os->InstallOS\_Imp(); };

};

class AppleCompuer :public Computer

{

public:

void InstallOS(OS\* os) { cout << "Apple"; os->InstallOS\_Imp(); };

};

int main()

{

OS\* os1 = new Windows();

OS\* os2 = new MacOS();

Computer\* computer1 = new AppleCompuer();

Computer\* computer2 = new DellCompuer();

computer1->InstallOS(os2);

computer2->InstallOS(os1);

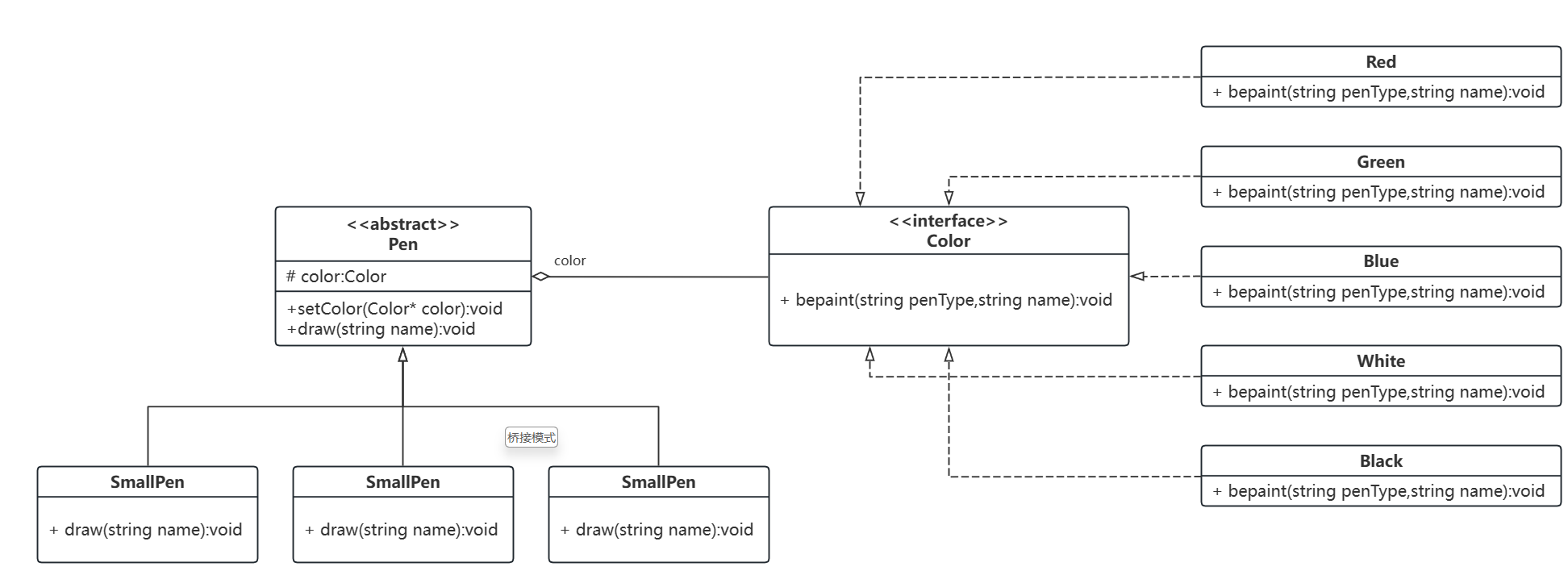
delete os1, os2, computer1, computer2;

}

## 9.5模式实例

模拟毛笔

现需要提供大中小3种型号的画笔，能够绘制5种不同颜色，如果使用蜡笔，我们需要准备3\*5=15支蜡笔，也就是说必须准备15个具体的蜡笔类。而如果使用毛笔的话，只需要3种型号的毛笔，外加5个颜料盒，用3+5=8个类就可以实现15支蜡笔的功能。本实例使用桥接模式来模拟毛笔的使用过程



class Color

{

public:

virtual void bepaint(string penType, string name) = 0;

};

class Red :public Color

{

void bepaint(string penType, string name)

{

cout << "red " << penType <<name << endl;

}

};

class Green :public Color

{

void bepaint(string penType, string name)

{

cout << "red " << penType <<name << endl;

}

};

class Pen

{

public:

void setColor(Color\* color) { this->color = color; }

virtual void draw(string name) {};

protected:

Color\* color;

};

class SmallPen:public Pen

{

public:

void draw(string name)

{

string penType = "小号毛笔绘制 ";

color->bepaint(penType, name);

}

};

class MiddlePen :public Pen

{

public:

void draw(string name)

{

string penType = "中号毛笔绘制 ";

color->bepaint(penType, name);

}

};

class BigPen :public Pen

{

public:

void draw(string name)

{

string penType = "大号毛笔绘制 ";

color->bepaint(penType, name);

}

};

int main()

{

Color\* color = new Red();

Pen\* middlePen = new MiddlePen();

middlePen->setColor(color);

middlePen->draw("Hello world");

delete color, middlePen;

}

## 9.6模式优缺点

优点：

* 分离抽象接口及其实现部分。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。
* 在很多情况下，桥接模式可以取代多重/多层继承方案，多重/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。

## 9.7适用环境

* 一个类存在两个（或多个）独立变化的维度，且这两个（或多个）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重/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。

## 9.8模式应用

(1) Java语言通过Java虚拟机实现了平台的无关性。

(2) Java为AWT中的每一个GUI构件都提供了一个Peer构件，在AWT中的Peer架构就使用了桥接模式。

## 9.9模式本质

桥接模式的本质：**分离抽象和实现**。

# 10.组合模式

## 10.1模式动机

对于树形结构，当容器对象（如文件夹）的某一个方法被调用时，将遍历整个树形结构，寻找也包含这个方法的成员对象（可以是容器对象，也可以是叶子对象，如子文件夹和文件）并调用执行。（递归调用）

由于容器对象和叶子对象在功能上的区别，在使用这些对象的客户端代码中必须有区别地对待容器对象和叶子对象，而实际上大多数情况下我们希望一致地处理它们，因为对于这些对象的区别对待将会使得程序非常复杂。

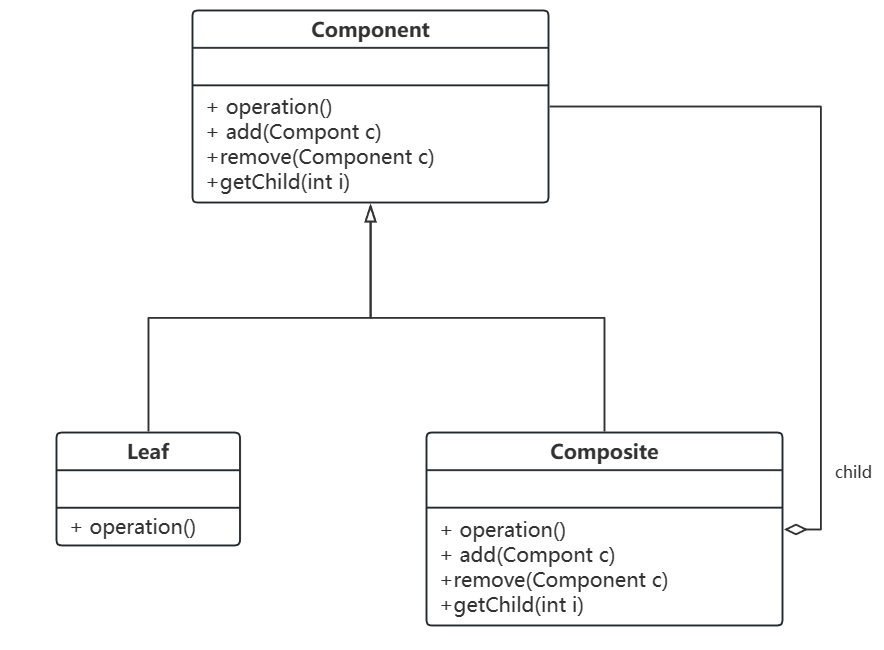
组合模式就是为解决此类问题而诞生，它通过一种巧妙的设计方案使得客户端可以一致性地处理整个树形结构或者树形结构的一部分，也可以一致性地处理树形结构中的叶子节点（不包含子节点的节点）和容器节点（包含子节点的节点）。

## 10.2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“整体-部分”关系的层次结构。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。

组合模式又可以称为“整体-部分”(Part-Whole)模式。

## 10.3模式结构



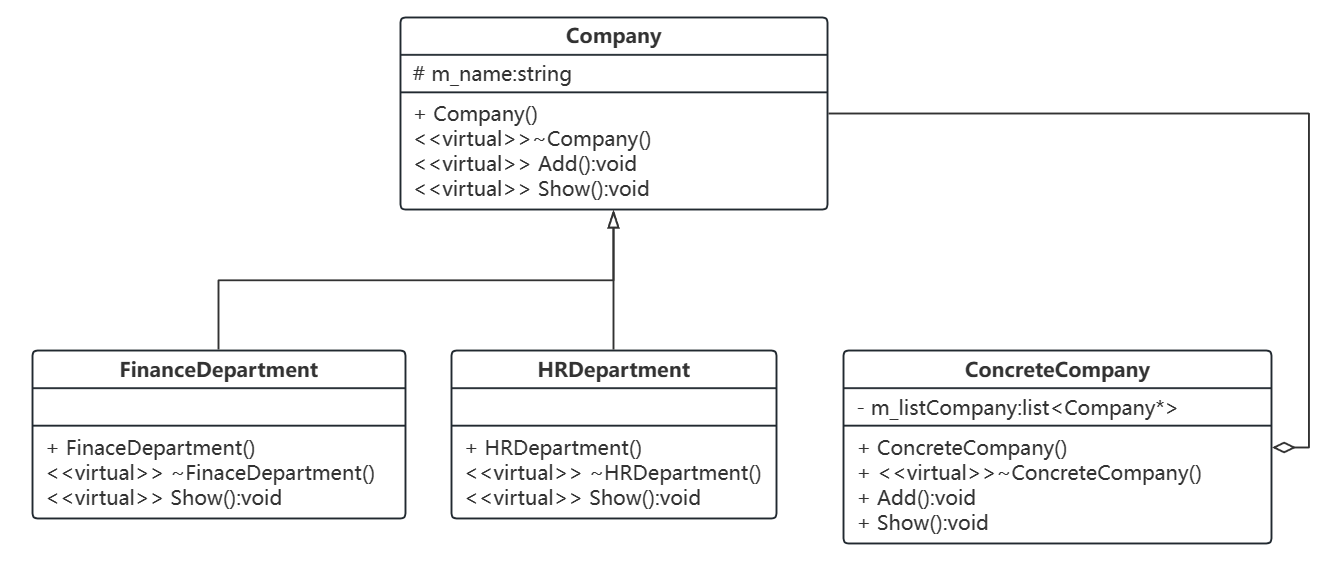
Component：抽象组件，可以是接口或抽象类，为叶子组件和容器组件对象声明接口，在该角色中可以包含所有子类共有行为的声明和实现。在抽象组件中定义了访问及管理它的子组件的方法，如增加子组件、删除子组件、获取子组件等。

Leaf：叶子组件，在组合结构中表示叶子节点对象，叶子节点没有子节点，它实现了在抽象组件中定义的行为。对于那些访问及管理子组件的方法，可以通过异常等方式进行处理。

Composite：容器组件，在组合结构中表示容器节点对象，容器节点包含子节点，其子节点可以是叶子节点，也可以是容器节点，它提供一个集合用于存储子节点，实现了在抽象组件中定义的行为，包括那些访问及管理子组件的方法，在其业务方法中可以递归调用其子节点的业务方法。

## 10.4模式分析

比如一个集团公司，它有一个母公司，下设很多家子公司。不管是母公司还是子公司，都有各自直属的财务部、人力资源部、销售部等。对于母公司来说，不论是子公司，还是直属的财务部、人力资源部，都是它的部门。整个公司的部门拓扑图就是一个树形结构。



class Company

{

public:

Company(string name) { m\_name = name; }

virtual ~Company(){}

virtual void Add(shared\_ptr<Company> pCom) {};

virtual void Show(int depth) {};

protected:

string m\_name;

};

//具体公司

class ConcreteCompany :public Company

{

public:

ConcreteCompany(string name) :Company(name) {}

virtual ~ConcreteCompany() {};

void Add(shared\_ptr<Company> pCom) { m\_listCompany.push\_back(pCom); }

void Show(int depth)

{

for (int i = 0; i < depth; i++)

cout << "-";

cout << m\_name;

auto iter = m\_listCompany.begin();

for (iter; iter != m\_listCompany.end(); iter++)

(\*iter)->Show(depth + 2);

}

private:

list<shared\_ptr<Company>> m\_listCompany;

};

//具体的部分，财务部

class FinanceDepartment :public Company

{

public:

FinanceDepartment(string name):Company(name){}

virtual ~FinanceDepartment() {};

virtual void Show(int depth)//已经是叶子节点，无需添加

{

for (int i = 0; i < depth; i++)

cout << "-";

cout << m\_name << endl;

}

};

//具体的部门，人力资源部

class HRDepartment :public Company

{

public:

HRDepartment(string name) :Company(name) {}

virtual ~HRDepartment() {};

virtual void Show(int depth)//已经是叶子节点，无需添加

{

for (int i = 0; i < depth; i++)

cout << "-";

cout << m\_name << endl;

}

};

int main()

{

shared\_ptr<Company> root = make\_shared<ConcreteCompany>("总公司");

shared\_ptr<Company> leaf1 = make\_shared<FinanceDepartment>("财务部");

shared\_ptr<Company> leaf2 = make\_shared<HRDepartment>("人力资源部");

root->Add(leaf1);

root->Add(leaf2);

//分公司A

shared\_ptr<Company> mid1 = make\_shared<ConcreteCompany>("分公司A");

shared\_ptr<Company> leaf3 = make\_shared<FinanceDepartment>("财务部");

shared\_ptr<Company> leaf4 = make\_shared<HRDepartment>("人力资源部");

mid1->Add(leaf3);

mid1->Add(leaf4);

//分公司B

shared\_ptr<Company> mid2 = make\_shared<ConcreteCompany>("分公司B");

shared\_ptr<Company> leaf5 = make\_shared<FinanceDepartment>("财务部");

shared\_ptr<Company> leaf6 = make\_shared<HRDepartment>("人力资源部");

mid2->Add(leaf5);

mid2->Add(leaf6);

root->Add(mid1);

root->Add(mid2);

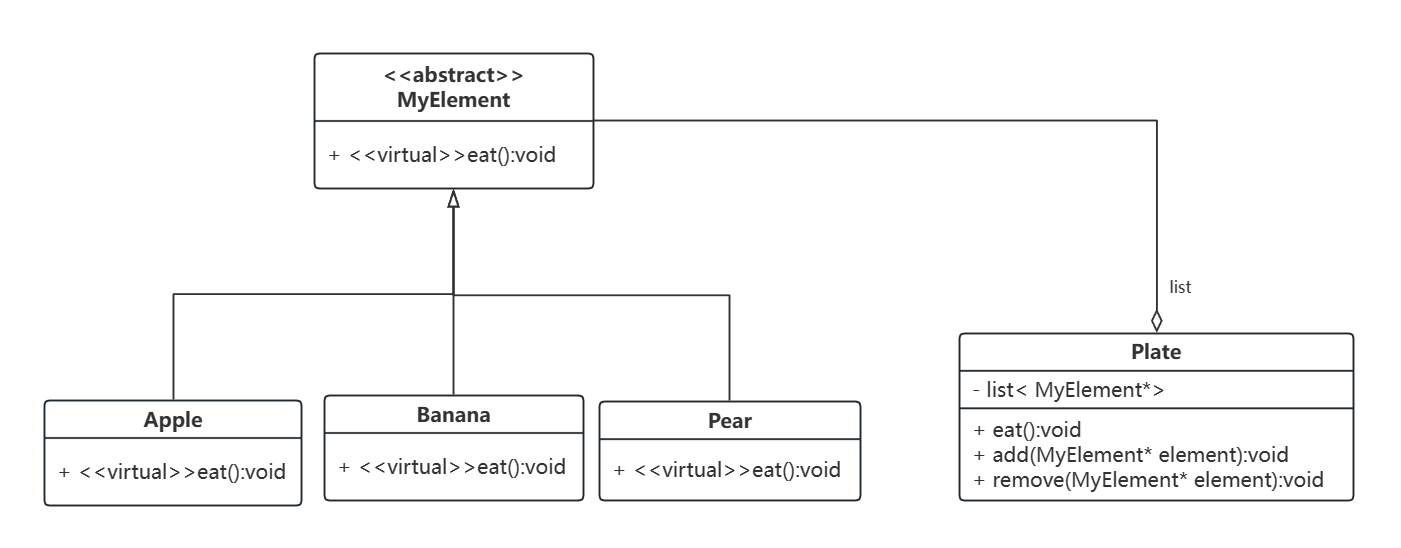
root->Show(0);

}

## 10.5 模式实例

水果盘

在水果盘(Plate)中有一些水果，如苹果(Apple)、香蕉(Banana)、梨子(Pear)，当然大水果盘中还可以有小水果盘，现需要对盘中的水果进行遍历（吃），当然如果对一个水果盘执行“吃”方法，实际上就是吃其中的水果。使用组合模式模拟该场景。



class MyElement

{

public:

virtual void eat() = 0;

virtual void add(shared\_ptr<MyElement> element) {};

virtual void Delete(shared\_ptr<MyElement> element) {};

};

//叶子节点 水果

class Apple :public MyElement

{

public:

void eat() { cout << "吃苹果" << endl;}

};

class Banana :public MyElement

{

public:

void eat() { cout << "吃香蕉" << endl; }

};

class Pear :public MyElement

{

public:

void eat() { cout << "吃梨" << endl; }

};

//容器节点 水果盘

class FruitPlate :public MyElement

{

public:

void add(shared\_ptr<MyElement> element)

{

m\_list.push\_back(element);

}

void Delete(shared\_ptr<MyElement> element)

{

auto iter = m\_list.begin();

for (iter; iter != m\_list.end(); iter++)

{

if (\*iter == element)

{

m\_list.erase(iter);

cout << "Delete successfully" << endl;

return;

}

}

cout << "Delete fail" << endl;

}

void eat()

{

for (auto iter : m\_list)

iter->eat();

}

private:

list<shared\_ptr<MyElement>> m\_list;

};

int main()

{

shared\_ptr<MyElement> apple = make\_shared<Apple>();

shared\_ptr<MyElement> banana = make\_shared<Banana>();

shared\_ptr<MyElement> pear = make\_shared<Pear>();

shared\_ptr<MyElement> fruitPlate1 = make\_shared<FruitPlate>();

fruitPlate1->add(apple);

fruitPlate1->add(banana);

fruitPlate1->add(pear);

fruitPlate1->eat();

}

## 10.6模式优缺点

优点：

* 定义了包含基本对象和组合对象的类层次结构，在组合模式中，基本对象可以被组合成更复杂的组合对象，而组合对象又可以组合成更复杂的组合对象，可以不断地递归组合下去，从而构成一个统一的组合对象的类层次结构。
* 统一了组合对象和叶子对象，在组合模式中，可以把叶子对象当作特殊的组合对象看待，为它们定义统一的父类，从而把组合对象和叶子对象的行为统一起来。
* 简化了客户端调用，组合模式通过统一组合对象和叶子对象，使得客户端在使用它们的时候，就不需要再去区分它们，客户不关心使用的到底是什么类型的对象，这就大大简化了客户端的使用。
* 更容易扩展，由于客户端是统一的面对Component来操作，因此，新定义的Composite或Leaf子类能够很容易的与已有的结构一起工作，而客户端不需要为增添了新的组件类而改变。

缺点：

很难限制组合中的组件类型，容易增加新的组件也会带来一些问题，比如很难限制组合中的组件类型。这在需要检测组件类型的时候，使得我们不能依靠编译期的类型约束来完成，必须在运行期间动态检测。

## 10.7模式适用环境

* 如果想表示对象的整体-部分层次结构，可以选用组合模式，把整体和部分的操作统一起来，使得层次结构实现更简单，从外部来使用这个层次结构也简单。
* 如果希望统一的使用组合结构中的所有对象，可以选用组合模式，这正是组合模式提供的主要功能。

## 10.8模式本质

组合模式的本质：统一叶子对象和组合对象

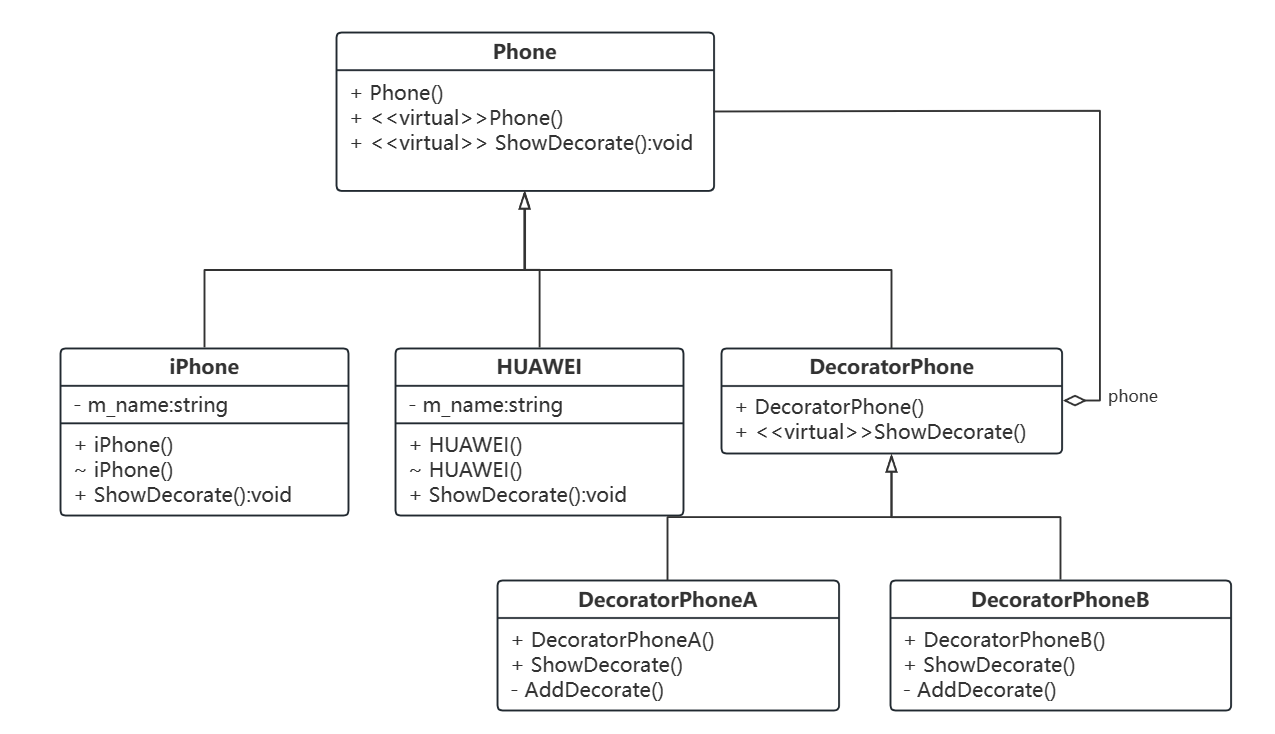
# 11.装饰模式

## 11.1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(Responsibility)，就增加对象功能来说，装饰模式比生成子类实现更为灵活。其别名也可以称为包装器(Wrapper)，与适配器模式的别名相同，但它们适用于不同的场合。根据翻译的不同，装饰模式也有人称之为“油漆工模式”，它是一种对象结构型模式。

## 11.2模式分析

比如有一个手机，允许你为手机添加特性，比如增加挂件、屏幕贴膜等。一种灵活的设计方式是，将手机嵌入到另一对象中，由这个对象完成特性的添加，我们称这个嵌入的对象为装饰。



class Phone

{

public:

Phone() {};

virtual ~Phone() {};

virtual void ShowDecorate() = 0;

};

class iPhone :public Phone

{

public:

iPhone() { m\_name = "iPhone"; }

~iPhone() {}

void ShowDecorate() { cout << m\_name << "的装饰" << endl; }

private:

string m\_name; //手机名称

};

class HUAWEI :public Phone

{

public:

HUAWEI() { m\_name = "HUAWEI"; }

~HUAWEI() {}

void ShowDecorate() { cout << m\_name << "的装饰" << endl; }

private:

string m\_name; //手机名称

};

//装饰类

class DecoratorPhone :public Phone

{

public:

DecoratorPhone(Phone \*phone) { m\_phone = phone; }

virtual void ShowDecorate() { m\_phone->ShowDecorate(); }

private:

Phone \*m\_phone;//要装饰的手机

};

class DecoratorPhoneA :public DecoratorPhone

{

public:

DecoratorPhoneA(Phone\* m\_phone) :DecoratorPhone(m\_phone) {}

void ShowDecorate() { DecoratorPhone::ShowDecorate(); AddDecorate(); }

private:

void AddDecorate() { cout << "增加挂件" << endl; }

};

class DecoratorPhoneB :public DecoratorPhone

{

public:

DecoratorPhoneB(Phone\* m\_phone) :DecoratorPhone(m\_phone) {}

void ShowDecorate() { DecoratorPhone::ShowDecorate(); AddDecorate(); }

private:

void AddDecorate() { cout << "屏幕贴膜" << endl; }

};

int main()

{

shared\_ptr<Phone> iphone = make\_shared<iPhone>();

shared\_ptr<Phone> dpa = make\_shared<DecoratorPhoneA>(iphone.get());

shared\_ptr<Phone> dpb = make\_shared<DecoratorPhoneB>(iphone.get());

dpa->ShowDecorate();

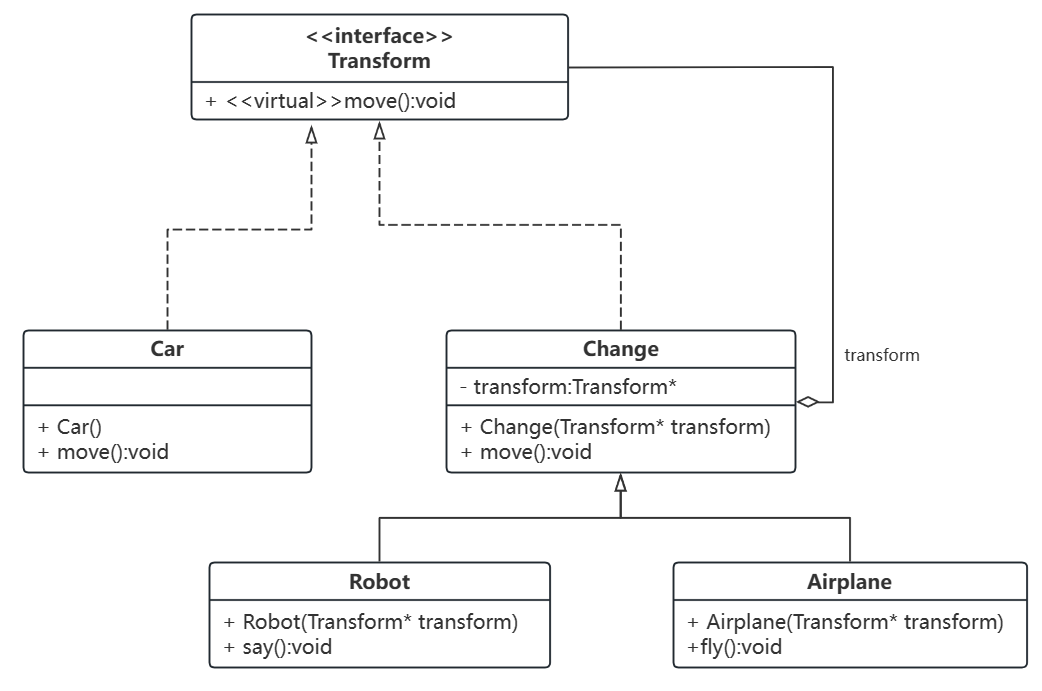
dpb->ShowDecorate();

return 0;

}

## 11.3模式实例

变形金刚



class Transform

{

public:

virtual void move() = 0;

};

class Car :public Transform

{

public:

Car() { cout << "变形金刚是一辆车" << endl; }

void move() { cout << "在地上移动" << endl; }

};

class Change :public Transform

{

public:

Change(shared\_ptr<Transform> transform) :m\_transform(transform) {}

void move() { m\_transform->move(); }

private:

shared\_ptr<Transform> m\_transform;

};

class Robot :public Change

{

public:

Robot(shared\_ptr<Transform> transform) :Change(transform) { cout << "变成机器人" << endl; }

void say() { cout << "说话" << endl; }

};

class Airplane :public Change

{

public:

Airplane(shared\_ptr<Transform> transform) :Change(transform) { cout << "变成飞机" << endl; }

void move() { fly(); }

private:

void fly() { cout << "飞" << endl; }

};

## 11.4模式优缺点

优点：

* 比继承更灵活，从为对象添加功能的角度来看，装饰模式比继承来得更灵活。
* 更容易复用功能，装饰模式把一系列复杂的功能，分散到每个装饰器当中，一般一个装饰器只实现一个功能，这样实现装饰器变得简单，更重要的是这样有利于装饰器功能的复用，可以给一个对象增加多个同样的装饰器，也可以把一个装饰器用来装饰不同的对象，从而复用装饰器的功能。
* 简化高层定义，装饰模式可以通过组合装饰器的方式，给对象增添任意多的功能，

缺点：

* 装饰模式是把一系列复杂的功能，分散到每个装饰器当中，一般一个装饰器只实现一个功能，这样会产生很多细粒度的对象，而且功能越复杂，需要的细粒度对象越多。
* 装饰模式提供了一种比继承更加灵活机动的解决方案，但同时也意味着比继承更加易于出错，排错也很困难，

## 11.5模式适用环境

* 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。不能采用继承的情况主要有两类：第一类是系统中存在大量独立的扩展，为支持每一种扩展或者扩展之间的组合将产生大量的子类，使得子类数目呈爆炸性增长；第二类是因为类已定义为不能被继承。

## 11.6模式本质

动态组合

# 12.外观模式

## 12.1模式定义

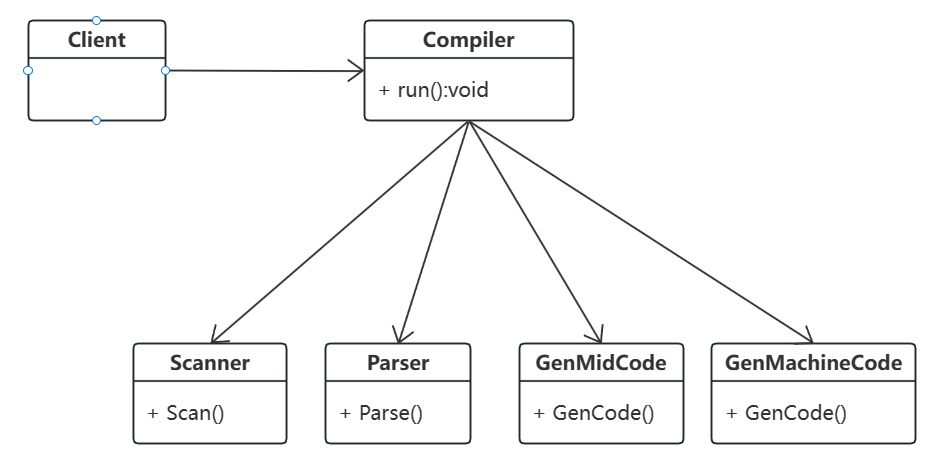
外观模式(Facade Pattern)：门面模式，为子系统中的一组接口提供一个统一的入口。外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

外观模式中，一个子系统的外部与其内部的通信通过一个统一的外观类进行，外观类将客户类与子系统的内部复杂性分隔开，使得客户类只需要与外观角色打交道，而不需要与子系统内部的很多对象打交道。

外观模式是迪米特原则的一种具体实现。

## 12.2模式分析

举个编译器的例子，假设编译一个程序需要经过四个步骤：词法分析、语法分析、中间代码生成、机器码生成。学过编译都知道，每一步都很复杂。对于编译器这个系统，就可以使用外观模式。



class Scanner

{

public:

void Scan() { cout << "词法分析" << endl; }

};

class Parser

{

public:

void Parse() { cout << "语法分析" << endl; }

};

class Compiler

{

public:

void run()

{

Scanner scanner;

Parser parse;

scanner.Scan();

parse.Parse();

}

};

int main()

{

Compiler comp;

comp.run();

}

## 12.3模式优缺点

优点：

* 外观模式对客户端屏蔽了子系统组件，减少了客户端所需处理的对象数目，并使得子系统使用起来更加容易。
* 外观模式实现了子系统与客户端之间的松耦合关系，这使得子系统的变化不会影响到调用它的客户端，只需要调整外观类即可。
* 只是提供了一个访问子系统的统一入口，并不影响用户直接使用子系统类。
* 通过合理使用外观模式，可以帮助我们更好的划分访问的层次。有些方法是提供系统外使用的，有些方法是系统内部使用的。把需要暴露给外部的功能集中到外观中。

缺点：

* 外观模式不能很好地限制客户端直接使用子系统类，如果对客户端访问子系统类做太多的限制则减少了可变性和灵活性。
* 如果设计不当，增加新的子系统可能需要修改外观类的源代码，违背了开闭原则。

## 12.4模式适用环境

* 当要为一个复杂子系统提供一个简单接口时可以使用外观模式。该接口可以满足大多数用户的需求，而且用户也可以越过外观类直接访问子系统。
* 客户端程序与多个子系统之间存在很大的依赖性。引入外观类可以将子系统与客户端解耦，从而提高子系统的独立性和可移植性。
* 在层次化结构中，可以使用外观模式定义系统中每一层的入口，层与层之间不直接产生联系，而通过外观类建立联系，降低层之间的耦合度。

12.5模式本质

外观模式是实现代码重构以达到“迪米特法则”要求的一个强有力的武器。

封装交互，简化调用。

# 13.享元模式

## 13.1模式动机与定义

享元模式实现相同或相似对象的重用。在逻辑上每一个出现的字符都有一个对象与之对应，然而在物理上它们却共享同一个享元对象，这个对象可以出现在一个字符串的不同地方，相同的字符对象都指向同一个实例。

在享元模式中，存储这些共享实例对象的地方称为享元池(Flyweight Pool)。我们可以针对每一个不同的字符创建一个享元对象，将其放在享元池中，需要时再从享元池取出。

享元模式(Flyweight Pattern)：运用共享技术有效地支持大量细粒度对象的复用。系统只使用少量的对象，而这些对象都很相似，状态变化很小，可以实现对象的多次复用。由于享元模式要求能够共享的对象必须是细粒度对象，因此它又称为轻量级模式。

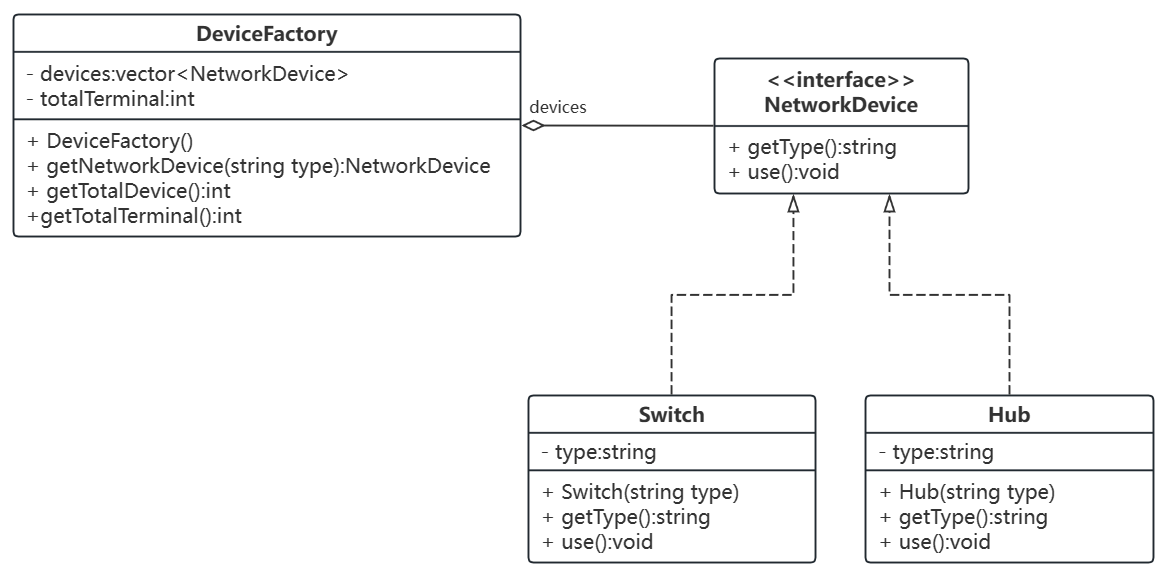
享元模式一般结合工厂模式一起使用，在它的结构图中包含了一个享元工厂类。

享元工厂类的作用在于提供一个用于存储享元对象的享元池，当用户需要对象时，首先从享元池中获取，如果享元池中不存在，则创建一个新的享元对象返回给用户，并在享元池中保存该新增对象。

## 13.2模式实例

共享网络设备（无外部状态）

很多网络设备都是支持共享的，如交换机、集线器等，多台终端计算机可以连接同一台网络设备，并通过该网络设备进行数据转发。



class NetworkDevice

{

public:

virtual string getType() =0;

virtual void use() =0;

};

class Switch:public NetworkDevice

{

public:

Switch(string type):type(type){}

string getType() { return type; }

void use() { cout << "Linked by switch,type is" << type << endl;; }

private:

string type;

};

class Hub :public NetworkDevice

{

public:

Hub(string type) :type(type) {}

string getType() { return type; }

void use() { cout << "Linked by Hub,type is" << type<<endl; }

private:

string type;

};

class DeviceFactory

{

public:

DeviceFactory()

{

shared\_ptr<NetworkDevice> nd1 = make\_shared<Switch>("Cisco-WS-c295024");

shared\_ptr<NetworkDevice> nd2 = make\_shared<Hub>("TP-link-HF8M");

devices.push\_back(nd1);

devices.push\_back(nd2);

}

shared\_ptr<NetworkDevice> getNetworkDevice(string type)

{

transform(type.begin(), type.end(), type.begin(), ::tolower);

if (type == "cisco")

{

totalTerminal++;

return devices[0];

}

else if (type == "tp")

{

totalTerminal++;

return devices[1];

}

else

return nullptr;

}

int getTotalDevice() { return devices.size(); }

int getTotalTerminal() { return totalTerminal; }

private:

vector<shared\_ptr<NetworkDevice>> devices;

int totalTerminal = 0;

};

int main()

{

shared\_ptr<NetworkDevice> nd1, nd2, nd3, nd4, nd5;

DeviceFactory df;

nd1 = df.getNetworkDevice("cisco");

nd1->use();

nd2 = df.getNetworkDevice("cisco");

nd2->use();

nd3 = df.getNetworkDevice("cisco");

nd3->use();

nd4 = df.getNetworkDevice("tp");

nd4->use();

nd5 = df.getNetworkDevice("tp");

nd5->use();

cout << "Total Device" << df.getTotalDevice() << endl;;

cout << "TotalTerminal" << df.getTotalTerminal() << endl;;

}

## 13.3模式优缺点

优点：

* 可以极大减少内存中对象的数量，使得相同或相似对象在内存中只保存一份，从而可以节约系统资源，提高系统性能。
* 享元模式的外部状态相对独立，而且不会影响其内部状态，从而使得享元对象可以在不同的环境中被共享。

缺点：

* 享元模式使得系统变得复杂，需要分离出内部状态和外部状态，这使得程序的逻辑复杂化。
* 为了使对象可以共享，享元模式需要将享元对象的部分状态外部化，而读取外部状态将使得运行时间变长。

## 13.4适用环境

* 一个系统有大量相同或者相似的对象，造成内存的大量耗费。
* 对象的大部分状态都可以外部化，可以将这些外部状态传入对象中。
* 在使用享元模式时需要维护一个存储享元对象的享元池，而这需要耗费一定的系统资源，因此，应当在需要多次重复使用享元对象时才值得使用享元模式。

## 13.5模式本质

**分离与共享**

# 14.代理模式

14.1模式动机及定义

代理模式(Proxy Pattern) ：给某一个对象提供一个代理，并由代理对象控制对原对象的引用。代理模式的英文叫做Proxy或Surrogate，它是一种对象结构型模式。

## 14.2模式分析

(1) 远程代理(Remote Proxy)

(2) 虚拟代理(Virtual Proxy)

(3) 保护代理(Protect Proxy)

(4) 缓冲代理(Cache Proxy)

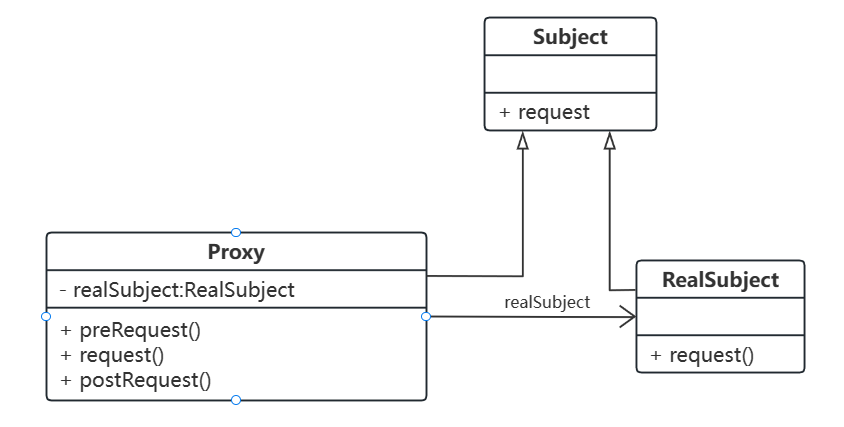
(5) 智能引用代理(Smart Reference Proxy)

(6) 防火墙代理(Firewall Proxy)

(7) 同步代理(Synchronization Proxy)

(8) 写入时复制代理(Copy-On-Write Proxy)

## 14.3模式结构

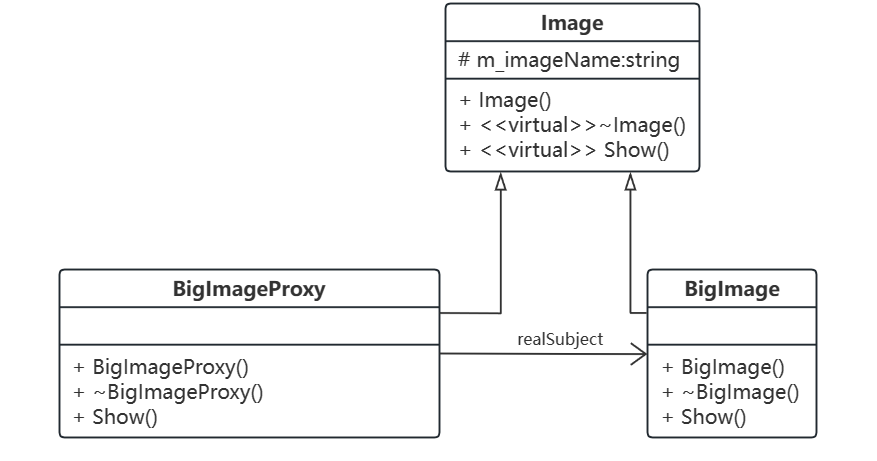


Subject：抽象主题角色，声明了真实主题和代理主题的共同接口，这样一来在任何使用真实主题的地方都可以使用代理主题，客户端通常需要针对抽象主题角色进行编程。

RealSubject：真实主题角色，定义了代理角色所代表的真实对象，在真实主题角色中实现了真实的业务操作，客户端可以通过代理主题角色间接调用真实主题角色中定义的操作。

Proxy：代理主题角色，它包含了对真实主题的引用，从而可以在任何时候操作真实主题对象。

有些图形对象的创建开销很大。但是打开文档必须很迅速，因此我们在打开文档时应避免一次性创建所有开销很大的对象。这里就可以运用代理模式，在打开文档时，并不打开图形对象，而是打开图形对象的代理以替代真实的图形。待到真正需要打开图形时，仍由代理负责打开。



class Image

{

public:

Image(string name):m\_imageName(name){}

virtual ~Image() {}

virtual void Show() {}

protected:

string m\_imageName;

};

class BigImage :public Image

{

public:

BigImage(string name):Image(name){}

~BigImage() {};

void Show() { cout << "Big Image: " << m\_imageName << endl; }

};

class BigImageProxy :public Image

{

public:

BigImageProxy(string name):Image(name),m\_bigImage(nullptr){}

~BigImageProxy(){}

void Show()

{

if (m\_bigImage == nullptr)

m\_bigImage = make\_shared<BigImage>(m\_imageName);

m\_bigImage->Show();

}

private:

shared\_ptr<BigImage> m\_bigImage;

};

int main()

{

shared\_ptr<Image> image = make\_shared<BigImageProxy>("proxy.jpg");

image->Show();

return 0;

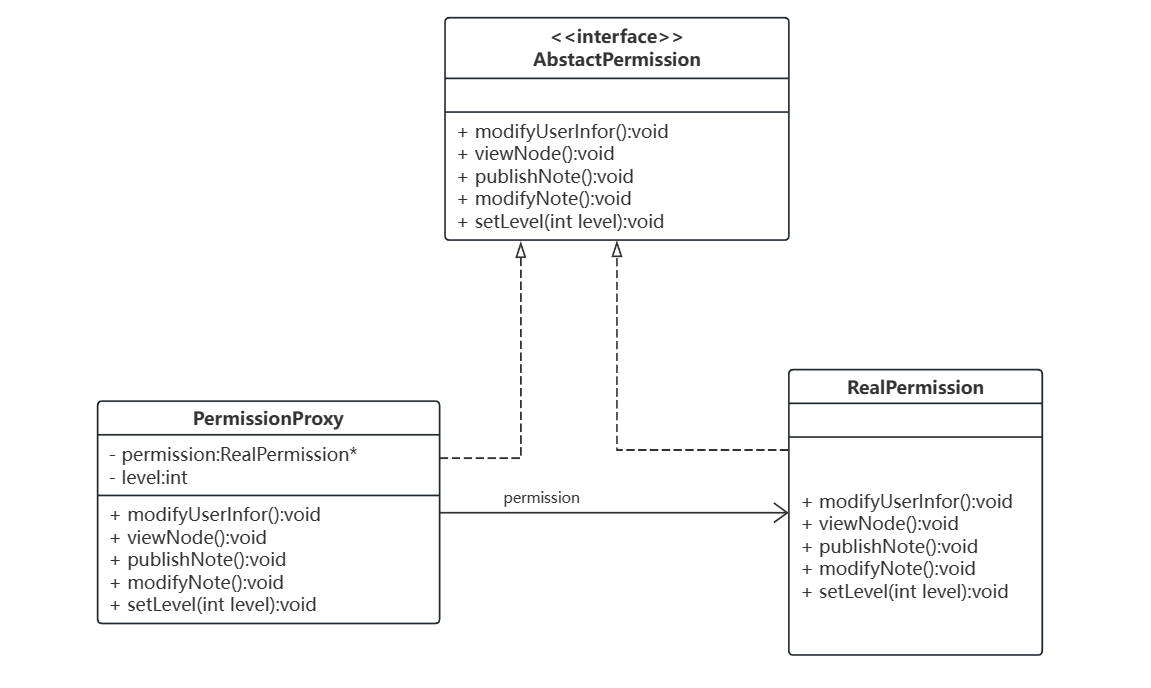
}

## 14.4模式实例

论坛权限控制代理

在一个论坛中已注册用户和游客的权限不同，已注册的用户拥有发帖、修改自己的注册信息、修改自己的帖子等功能；而游客只能看到别人发的帖子，没有其他权限。使用代理模式来设计该权限管理模块。

在本实例中我们使用代理模式中的保护代理，该代理用于控制对一个对象的访问，可以给不同的用户提供不同级别的使用权限。



class AbstractPermission

{

public:

virtual void modifyUserInfo() = 0;

virtual void viewNote() = 0;

virtual void publishNote() = 0;

virtual void modifyNote() = 0;

virtual void setLevel(int level) = 0;

};

class RealPermission :public AbstractPermission

{

virtual void modifyUserInfo(){ cout << "修改用户信息" << endl; }

virtual void viewNote() { cout << "查看帖子" << endl; }

virtual void publishNote() { cout << "发布新帖子" << endl; }

virtual void modifyNote() { cout << "修改发帖内容" << endl; }

virtual void setLevel(int level) {}

};

class PermissionProxy :public AbstractPermission

{

public:

virtual void modifyUserInfo()

{

if (level == 0)

cout << "没有权限" << endl;

else

permission->modifyNote();

}

virtual void viewNote()

{

permission->viewNote();

}

virtual void publishNote()

{

if (level == 0)

cout << "没有权限" << endl;

else

permission->publishNote();

}

virtual void modifyNote()

{

if (level == 0)

cout << "没有权限" << endl;

else

permission->modifyNote();

}

virtual void setLevel(int level)

{

this->level = level;

}

private:

shared\_ptr<AbstractPermission> permission = make\_shared<RealPermission>();

int level = 0;

};

int main()

{

shared\_ptr<AbstractPermission> permission = make\_shared<PermissionProxy>();

permission->modifyUserInfo();

permission->viewNote();

permission->publishNote();

permission->modifyNote();

cout << "-----------------------------------" << endl;

permission->setLevel(1);

permission->modifyUserInfo();

permission->viewNote();

permission->publishNote();

permission->modifyNote();

}

## 14.5模式优缺点

优点：

* 代理模式能够协调调用者和被调用者，在一定程度上降低了系统的耦合度。
* 客户端可以针对抽象主题进行编程，增加和更换代理类无须修改源代码，符合开闭原则，系统具有较好的灵活性和可扩展性。

此外，不同类型的代理模式也具有独特的优点，例如：

* 远程代理为位于两个不同地址空间对象的访问提供了一种实现机制，可以将一些消耗资源较多的对象和操作移至性能更好的计算机上，提高系统的整体运行效率。
* 虚拟代理通过一个消耗资源较少的对象来代表一个消耗资源较多的对象，在一定程度上节省系统的运行开销。
* 缓冲代理为某一个操作的结果提供临时的缓存存储空间，以便在后续使用中能够共享这些结果，优化系统性能，缩短执行时间。
* 保护代理可以控制对一个对象的访问权限，为不同用户提供不同级别的使用权限。

缺点：

* 由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。
* 实现代理模式需要额外的工作，有些代理模式的实现非常复杂

## 14.6模式适用环境

* 当需要为某一个被频繁访问的操作结果提供一个临时存储空间，以供多个客户端共享访问这些结果时可以使用缓冲代理。通过使用缓冲代理，系统无须在客户端每一次访问时都重新执行操作，只需直接从临时缓冲区获取操作结果即可。
* 当需要控制对一个对象的访问，为不同用户提供不同级别的访问权限时可以使用保护代理。
* 当需要为一个对象的访问（引用）提供一些额外的操作时可以使用智能引用代理。

## 14.7 模式本质

代理模式的本质：**控制对象访问。**

# 行为型模式

* 行为型模式(Behavioral Pattern)是对在不同的对象之间划分责任和算法的抽象化。
* 行为型模式不仅仅关注类和对象的结构，而且重点关注它们之间的相互作用。

职责链模式(Chain of Responsibility)

命令模式(Command)

解释器模式(Interpreter)

迭代器模式(Iterator)

中介者模式(Mediator)

备忘录模式(Memento)

观察者模式(Observer)

状态模式(State)

策略模式(Strategy)

模板方法模式(Template Method)

访问者模式(Visitor)

# 15.职责链模式

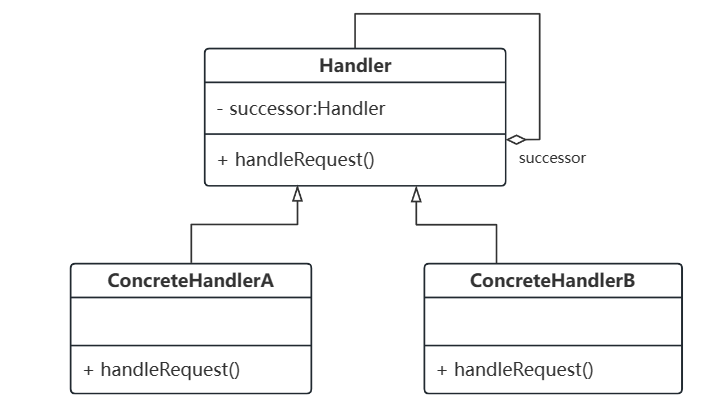
## 15.1模式动机及定义

在设计模式中，也有一种专门用于处理这种请求链式传递的模式，它就是将要介绍的职责链模式。

职责链可以是一条直线、一个环或者一个树形结构，最常见的职责链是直线型，即沿着一条单向的链来传递请求。

**职责链模式**(Chain of Responsibility Pattern)：避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。职责链模式是一种对象行为型模式。由于英文翻译的不同，职责链模式又称为责任链模式。

## 14.2模式结构

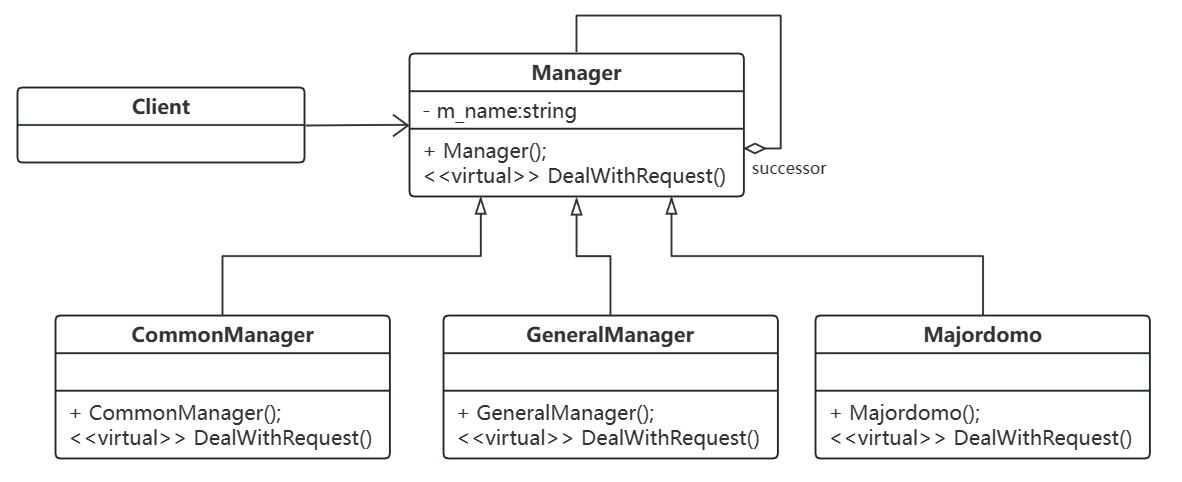


Handler：抽象处理者角色，它定义了一个处理请求的接口，一般设计为抽象类，由于不同的具体处理者处理请求的方式不同，因此在其中定义了抽象请求处理方法。因为每一个处理者的下家还是一个处理者，因此在抽象处理者中定义了一个抽象处理者类型的对象（如结构图中的successor），作为其对下家的引用。通过该引用，处理者可以连成一条链。

ConcreteHandler：具体处理者角色，它是抽象处理者的子类，可以处理用户请求，在具体处理者类中实现了抽象处理者中定义的抽象请求处理方法，在处理请求之前需要进行判断，看是否有相应的处理权限，如果可以处理请求就处理它，否则将请求转发给后继者；在具体处理者中可以访问链中下一个对象，以便请求的转发。

## 14.3模式分析

考虑员工要求加薪。公司的管理者一共有三级，总经理、总监、经理，如果一个员工要求加薪，应该向主管的经理申请，如果加薪的数量在经理的职权内，那么经理可以直接批准，否则将申请上交给总监。总监的处理方式也一样，总经理可以处理所有请求。这就是典型的职责链模式，请求的处理形成了一条链，直到有一个对象处理请求。



//抽象管理者

class Manager

{

public:

Manager(Manager\* manager, string name) :m\_manager(manager), m\_name(name){}

virtual void DealWithRequest(string name,int num){}

protected:

Manager\* m\_manager;//上级

string m\_name;

};

//经理

class CommonManger :public Manager

{

public:

CommonManger(Manager\* manager, string name):Manager(manager,name){}

void DealWithRequest(string name, int num)

{

if (num < 500)

{

cout << "经理" << m\_name << "批准" << name << "加薪" << num << endl;

}

else

{

cout << "经理" << name << "无法处理，交给总监处理" << endl;

m\_manager->DealWithRequest(name, num);

}

}

};

//总监

class Majordomo :public Manager

{

public:

Majordomo(Manager\* manager, string name) :Manager(manager, name) {}

void DealWithRequest(string name, int num)

{

if (num < 1000)

{

cout << "总监" << m\_name << "批准" << name << "加薪" << num << endl;

}

else

{

cout << "总监" << name << "无法处理，交给总经理处理" << endl;

m\_manager->DealWithRequest(name, num);

}

}

};

//总经理

class GeneralManger :public Manager

{

public:

GeneralManger(Manager\* manager, string name) :Manager(manager, name) {}

void DealWithRequest(string name, int num)

{

cout << "总经理" << m\_name << "批准" << name << "加薪" << num << endl;

}

};

int main()

{

Manager\* general = new GeneralManger(nullptr, "A");//总经理没有上级

Manager\* majordomo = new Majordomo(general, "B");

Manager\* common = new CommonManger(majordomo, "C");

common->DealWithRequest("D", 300);

common->DealWithRequest("E", 600);

common->DealWithRequest("F", 1000);

delete general, majordomo, common;

return 0;

}

## 14.4模式优缺点

优点：

* 职责链模式使得一个对象无须知道是其他哪一个对象处理其请求，对象仅需知道该请求会被处理即可，接收者和发送者都没有对方的明确信息，且链中的对象不需要知道链的结构，由客户端负责链的创建，降低了系统的耦合度。
* 请求处理对象仅需维持一个指向其后继者的引用，而不需要维持它对所有的候选处理者的引用，可简化对象的相互连接。
* 在给对象分派职责时，职责链可以给我们更多的灵活性，可以通过在运行时对该链进行动态的增加或修改来增加或改变处理一个请求的职责。
* 在系统中增加一个新的具体请求处理者时无须修改原有系统的代码，只需要在客户端重新建链即可，从这一点来看是符合“开闭原则”的。

缺点：

* 由于一个请求没有明确的接收者，那么就不能保证它一定会被处理，该请求可能一直到链的末端都得不到处理；一个请求也可能因职责链没有被正确配置而得不到处理。
* 对于比较长的职责链，请求的处理可能涉及到多个处理对象，系统性能将受到一定影响，而且在进行代码调试时不太方便。
* 如果建链不当，可能会造成循环调用，将导致系统陷入死循环。

## 14.5模式适用环境

* 有多个对象可以处理同一个请求，具体哪个对象处理该请求待运行时刻再确定，客户端只需将请求提交到链上，而无须关心请求的处理对象是谁以及它是如何处理的。
* 在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
* 可动态指定一组对象处理请求，客户端可以动态创建职责链来处理请求，还可以改变链中处理者之间的先后次序。

## 14.6模式本质

**分离职责，动态组合。**

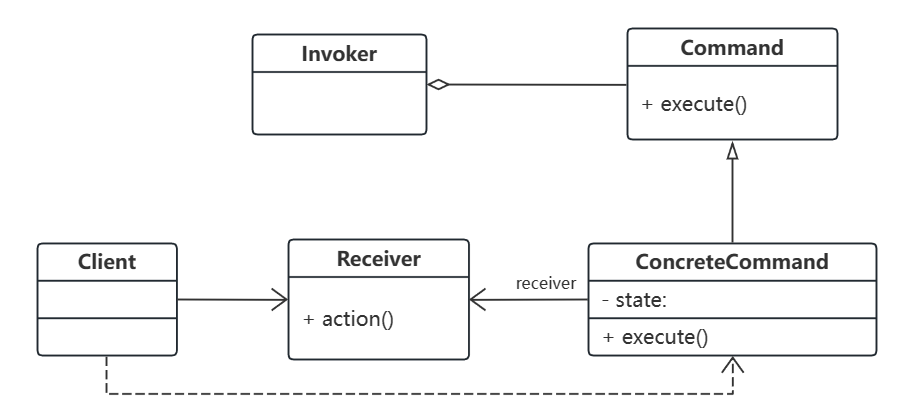
# 15.命令模式

## 15.1模式动机及定义

软件开发中也存在很多与开关和电器类似的请求发送者和接收者对象，例如一个按钮，它可能是一个“关闭窗口”请求的发送者，而按钮点击事件处理类则是该请求的接收者。

命令模式(Command Pattern)：将一个请求封装为一个对象，从而让我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作(Action)模式或事务(Transaction)模式。

## 15.2模式结构



Command：抽象命令类一般是一个抽象类或接口，在其中声明了用于执行请求的execute()等方法，通过这些方法可以调用请求接收者的相关操作。

ConcreteCommand：具体命令类是抽象命令类的子类，实现了在抽象命令类中声明的方法，它对应具体的接收者对象，将接收者对象的动作绑定其中。在实现execute()方法时，将调用接收者对象的相关操作(Action)。

Invoker：调用者，即请求发送者，它通过命令对象来执行请求。一个调用者并不需要在设计时确定其接收者，因此它只与抽象命令类之间存在关联关系。在程序运行时可以将一个具体命令对象注入其中，再调用具体命令对象的execute()方法，从而实现间接调用请求接收者的相关操作。

Receiver：接收者执行与请求相关的操作，它具体实现对请求的业务处理。

## 15.3模式分析

日常生活中，我们经常到饭店去吃饭，当然我们要吃什么会给服务员说，服务员在转告给厨师，点菜可以看做一个请求，看它如何用命令模式实现。

//命令处理者：厨师

class Chef

{

public:

void cook(string name)

{

cout << "厨师做" << name << endl;

}

};

//命令抽象类：点菜

class Commend

{

public:

Commend(Chef\* chef) :c\_chef(chef){}

virtual void excute() = 0;

protected:

Chef\* c\_chef;

};

//具体命令类

class BeefCmd :public Commend

{

public:

BeefCmd(Chef\* chef):Commend(chef){}

void excute()

{

c\_chef->cook("青椒炒牛肉");

}

};

class PotatoCmd :public Commend

{

public:

PotatoCmd(Chef\* chef) :Commend(chef) {}

void excute()

{

c\_chef->cook("青椒炒土豆");

}

};

//命令接收者：服务员

class Waiter

{

public:

void RecieveCmd(Commend\* cmd)

{

this->cmd = cmd;

}

void DealCmd()

{

cmd->excute();

}

private:

Commend\* cmd;

};

int main()

{

Waiter\* waiter = new Waiter;

Chef\* chef = new Chef;

Commend\* beef = new BeefCmd(chef);

Commend\* potato = new PotatoCmd(chef);

waiter->RecieveCmd(beef);

waiter->DealCmd();

waiter->RecieveCmd(potato);

waiter->DealCmd();

return 0;

}

## 15.4命令扩展

### 命令队列

其中最常用、灵活性最好的一种方式是增加一个CommandQueue类，由该类来负责存储多个命令对象，而不同的命令对象可以对应不同的请求接收者

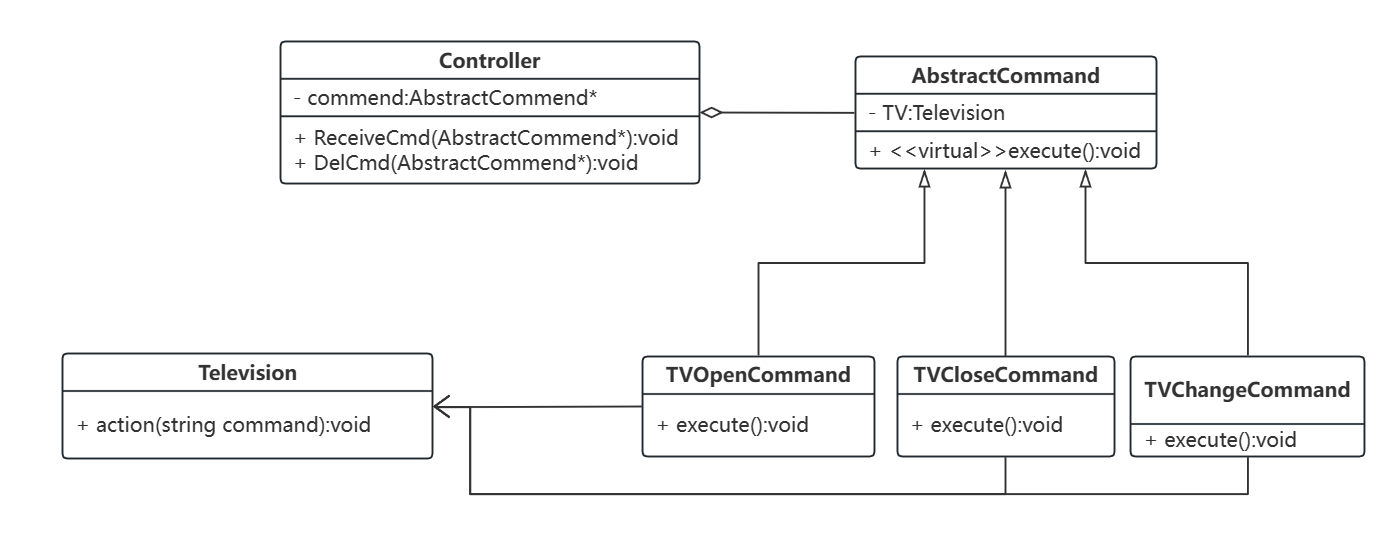
### 请求日志

请求日志就是将请求的历史记录保存下来，通常以日志文件(Log File)的形式永久存储在计算机中。

## 15.5模式实例

电视机遥控器

打开电视机、关闭电视机和切换频道。



class Television

{

public:

void action(string name)

{

cout << "电视执行" << name << endl;;

}

};

class AbstractCommand

{

public:

AbstractCommand(Television\* tv) :tv(tv) {}

virtual void execute() = 0;

protected:

Television\* tv;

};

class TVOpenCommand :public AbstractCommand

{

public:

TVOpenCommand(Television\* tv) :AbstractCommand(tv){}

void execute()

{

tv->action("开机");

}

};

class TVCloseCommand :public AbstractCommand

{

public:

TVCloseCommand(Television\* tv) :AbstractCommand(tv) {}

void execute()

{

tv->action("关机");

}

};

class TVChangeCommand :public AbstractCommand

{

public:

TVChangeCommand(Television\* tv) :AbstractCommand(tv) {}

void execute()

{

tv->action("换频道");

}

};

class Controller

{

public:

void ReceiveCmd(AbstractCommand\* cmd)

{

command = cmd;

}

void DelCmd()

{

command->execute();

}

private:

AbstractCommand\* command;

};

int main()

{

Controller\* control = new Controller;

Television\* tv = new Television;

AbstractCommand\* open = new TVOpenCommand(tv);

AbstractCommand\* close = new TVCloseCommand(tv);

AbstractCommand\* change = new TVChangeCommand(tv);

control->ReceiveCmd(open);

control->DelCmd();

control->ReceiveCmd(change);

control->DelCmd();

control->ReceiveCmd(close);

control->DelCmd();

}

## 15.6模式优缺点

优点：

* 更松散的耦合，请求者与接收者实现完全解耦，相同的请求者可以对应不同的接收者，相同的接收者也可以供不同的请求者使用，两者都具有良好的独立性。
* 更动态的控制，把请求封装起来，可以动态对它进行参数化、队列化和日志化等操作，使系统更灵活。
* 更容易的组合命令，能够很容易的设计一个命令队列或宏命令，使系统操作更简单，功能更强大。
* 更好的扩展性，新命令可以很容易地加入到系统中，无须修改原有代码，甚至客户类代码，满足“开闭原则”的要求。

缺点：

使用命令模式可能会导致某些系统有过多的具体命令类。因为针对每一个对请求接收者的调用操作都需要设计一个具体命令类，因此在某些系统中可能需要提供大量的具体命令类，这将影响命令模式的使用。

## 15.7模式优缺点

* 如果需要抽象出需要执行的动作，并参数化这些对象，可以选用命令模式，把这些需要执行的动作抽象成为命令，然后实现命令的参数化配置。
* 如果需要在不同的时刻指定、排列和执行请求，可以选用命令模式，把这些请求封装成为命令对象，然后实现把请求队列化。
* 如果需要支持取消操作，可以选用命令模式，通过管理命令对象，能很容易的实现命令的恢复和重做的功能。

# 16.迭代器模式

## 16.1模式动机及定义

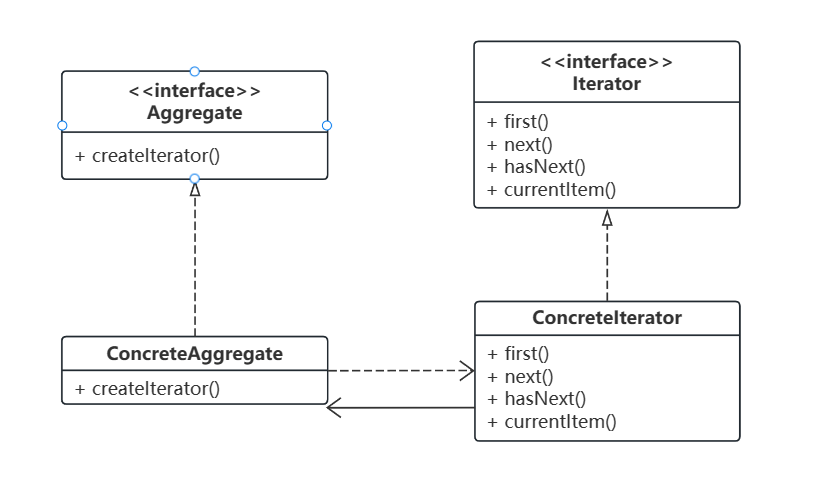
在软件开发中，也存在大量类似电视机一样的类，它们可以存储多个成员对象（元素），这些类通常称为聚合类(Aggregate Classes)，对应的对象称为聚合对象。

聚合对象拥有两个职责：一是存储数据；二是遍历数据。从依赖性来看，前者是聚合对象的基本职责；而后者既是可变化的，又是可分离的。

迭代器模式(Iterator Pattern) ：提供一种方法来访问聚合对象，而不用暴露这个对象的内部表示，其别名为游标(Cursor)。迭代器模式是一种对象行为型模式。

在迭代器模式结构中包含聚合和迭代器两个层次结构，考虑到系统的灵活性和可扩展性，在迭代器模式中应用了工厂方法模式。

## 16.2模式结构



Iterator：抽象迭代器，定义了访问和遍历元素的接口，声明了相应方法，例如：用于获取第一个元素的first()方法，用于访问下一个元素的next()方法，用于判断是否还有下一个元素的hasNext()方法，用于获取当前元素的currentItem()方法等，在具体迭代器中将实现这些方法。

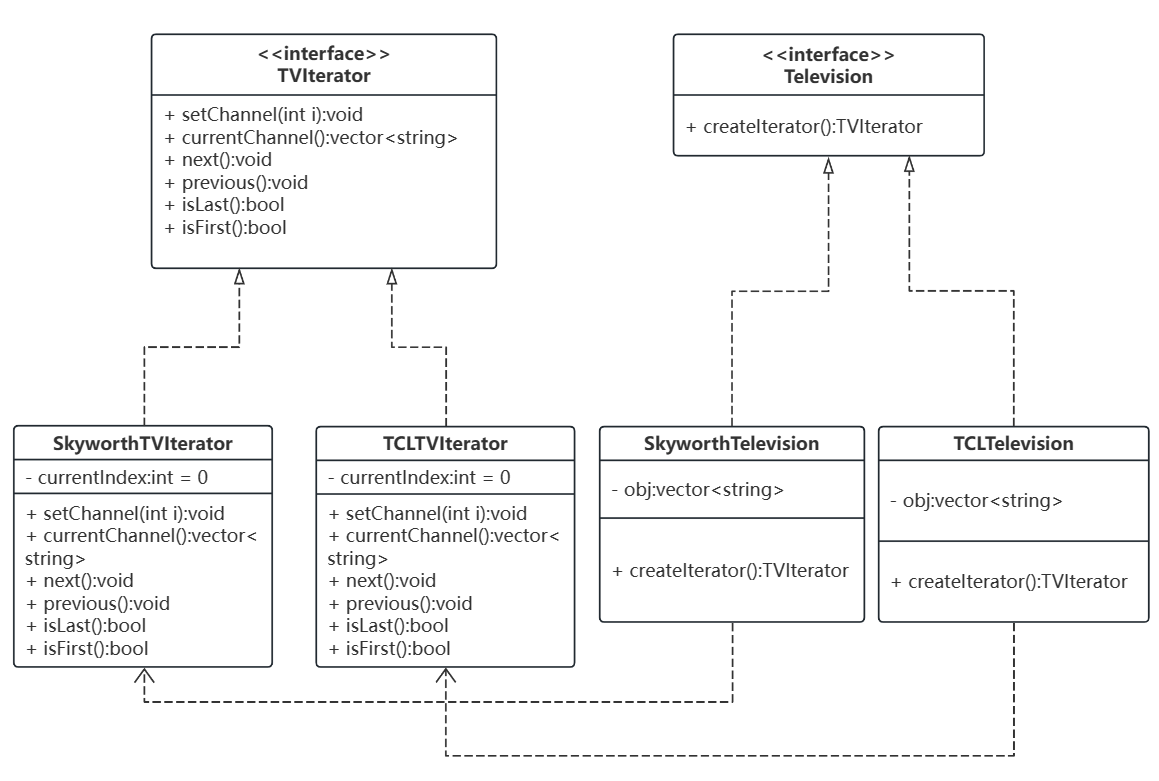
ConcreteIterator：具体迭代器，实现了抽象迭代器接口，完成对聚合对象的遍历，同时在具体迭代器中通过游标来记录在聚合对象中所处的当前位置，在具体实现时，游标通常是一个表示位置的非负整数。

Aggregate：抽象聚合类，用于存储和管理元素对象，声明一个createIterator()方法用于创建一个迭代器对象，充当抽象迭代器工厂角色。

ConcreteAggregate：具体聚合类，实现了在抽象聚合类中声明的createIterator()方法，该方法返回一个与该具体聚合类对应的具体迭代器ConcreteIterator实例。

## 16.3模式实例

电视机遥控器



class TVIterator

{

public:

virtual void setChannel(int i) = 0;

virtual string currentChannel() = 0;

virtual void next() = 0;

virtual void previous() = 0;

virtual bool isLast() = 0;

virtual bool isFirst() = 0;

};

class Television

{

public:

virtual TVIterator\* createIterator() = 0;

};

class SkyworthIterator :public TVIterator

{

public:

void setChannel(int i)

{

current = i;

}

string currentChannel()

{

return channels[current];

}

void next()

{

if (current < channels.size())

{

current++;

}

}

void previous()

{

if (current > 0)

{

current--;

}

}

bool isLast()

{

return current == channels.size() - 1;

}

bool isFirst()

{

return current == 0;

}

private:

int current = 0;

vector<string> channels = { "CCTV-1","CCTV-2","CCTV-3","CCTV-4","CCTV-5","CCTV-6","CCTV-7","CCTV-8",

"CCTV-9","CCTV-10","CCTV-11","CCTV-12","CCTV-13","CCTV-14","CCTV-15" };

};

class SkyworthTelevision :public Television

{

public:

TVIterator\* createIterator()

{

return new SkyworthIterator();

}

};

void display(TVIterator\* it)

{

it->setChannel(0);

cout<<"电视机频道："<<endl;

while (!it->isLast())

{

cout << it->currentChannel() << " ";

it->next();

}

cout << endl;

}

int main()

{

Television\* tv = new SkyworthTelevision();

TVIterator\* it = tv->createIterator();

display(it);

}

## 16.4模式优缺点

优点：

* 支持以不同的方式遍历一个聚合对象，在同一个聚合对象上可以定义多种遍历方式。在迭代器模式中只需要用一个不同的迭代器来替换原有迭代器即可改变遍历算法，我们也可以自己定义迭代器的子类以支持新的遍历方式。
* 迭代器简化了聚合类。由于引入了迭代器，在原有的聚合对象中不需要再自行提供数据遍历等方法，这样可以简化聚合类的设计。
* 在迭代器模式中，由于引入了抽象层，增加新的聚合类和迭代器类都很方便，无须修改原有代码，满足“开闭原则”的要求。

缺点：

* 由于迭代器模式将存储数据和遍历数据的职责分离，增加新的聚合类需要对应增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性。
* 抽象迭代器的设计难度较大，需要充分考虑到系统将来的扩展，例如JDK内置迭代器Iterator就无法实现逆向遍历，如果需要实现逆向遍历，只能通过子类ListIterator等来实现，而ListIterator迭代器无法用于操作Set类型的聚合对象。在自定义迭代器时，创建一个考虑全面的抽象迭代器并不是件很容易的事情。

## 16.5模式适用环境

* 访问一个聚合对象的内容而无须暴露它的内部表示。将聚合对象的访问与内部数据的存储分离，使得访问聚合对象时无须了解其内部实现细节。
* 需要为一个聚合对象提供多种遍历方式。
* 为遍历不同的聚合结构提供一个统一的接口，在该接口的实现类中为不同的聚合结构提供不同的遍历方式，而客户端可以一致性地操作该接口。

## 16.6模式本质

控制访问聚合对象中的元素。

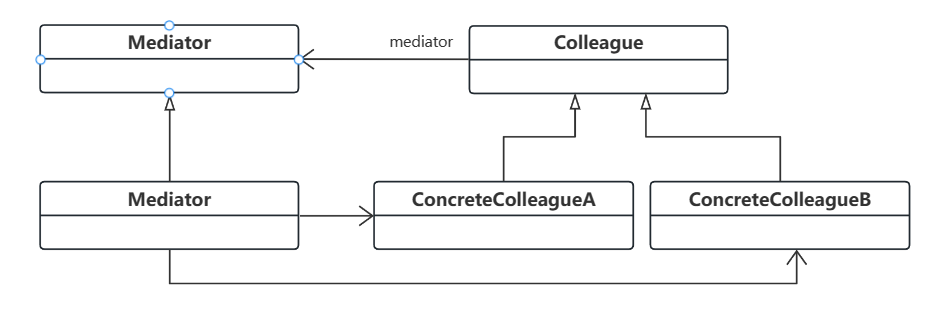
# 17.中介者模式

## 17.1模式动机及定义

通过引入群的机制，将极大减少系统中用户之间的两两通信，用户与多个用户之间的联系可以通过群来实现。

**中介者模式**(Mediator Pattern)定义：用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。中介者模式又称为调停者模式，它是一种对象行为型模式。

## 17.2模式结构



Mediator: 抽象中介者，它定义一个接口，该接口用于与各同事对象之间进行通信。

ConcreteMediator: 具体中介者，它是抽象中介者的子类，通过协调各个同事对象来实现协作行为，它维持了对各个同事对象的引用。

Colleague: 抽象同事类，它定义各个同事类公有的方法，并声明了一些抽象方法来供子类实现，同时它维持了一个对抽象中介者类的引用，其子类可以通过该引用来与中介者通信。

ConcreteColleague: 具体同事类，它是抽象同事类的子类；每一个同事对象在需要和其他同事对象通信时，先与中介者通信，通过中介者来间接完成与其他同事类的通信；在具体同事类中实现了在抽象同事类中声明的抽象方法。

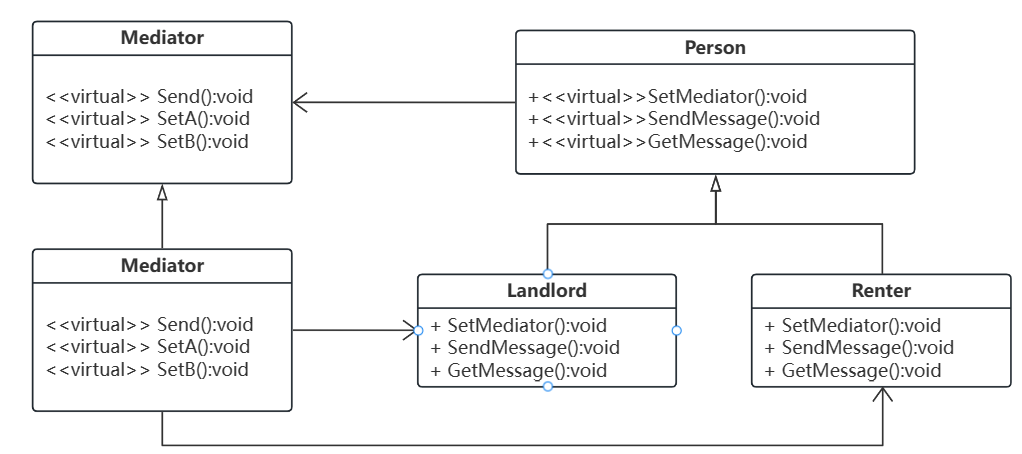
## 17.3模式分析

模式的核心在于中介者类的引入，它承担两方面职责：

中转作用（结构性）

协调作用（行为性）

本文就以租房为例子，如果没有房屋中介，那么房客要自己找房东，而房东也要自己找房客，非常不方便。有了房屋中介机构就方便了，房东可以把要出租的房屋信息放到中介机构，而房客可以去中介机构咨询。在软件中，就是多个对象之间需要通信，如果没有中介，对象就需要知道其他对象，最坏情况下，可能需要知道所有其他对象，而有了中介对象就方便多了，对象只需与中介对象通信，而不用知道其他的对象。



class Mediator;

//抽象人

class Person

{

public:

virtual void SetMediator(Mediator\* mediator){} //设置中介

virtual void SendMessage(string message){} //向中介发送信息

virtual void GetMessage(string message){} //从中介获取信息

protected:

Mediator\* m\_mediator;

};

//抽象中介机构

class Mediator

{

public:

virtual void Send(string message, Person\* person) {}

virtual void SetA(Person\* A) {} //设置其中一方

virtual void SetB(Person\* B) {}

};

//租房者

class Renter :public Person

{

public:

void SetMediator(Mediator\* mediator) { m\_mediator = mediator; }

void SendMessage(string message) { m\_mediator->Send(message, this); }

void GetMessage(string message) { cout << "租房者收到信息" << message; }

};

//房东

class Landlord :public Person

{

public:

void SetMediator(Mediator\* mediator) { m\_mediator = mediator; }

void SendMessage(string message) { m\_mediator->Send(message, this); }

void GetMessage(string message) { cout << "房东收到信息" << message; }

};

//房屋中介

class HouseMediator :public Mediator

{

public:

HouseMediator():m\_A(nullptr),m\_B(nullptr){}

void SetA(Person\* A) { m\_A = A; } //设置其中一方

void SetB(Person\* B) { m\_B = B; }

void Send(string message, Person\* person)

{

if (person == m\_A)//租房者给房东发信息

m\_B->GetMessage(message);

else

m\_A->GetMessage(message);

}

private:

Person\* m\_A;//租房者

Person\* m\_B;//房东

};

int main()

{

Mediator\* mediator = new HouseMediator();

Person\* person1 = new Renter();//租房者

Person\* person2 = new Landlord();//房东

mediator->SetA(person1);

mediator->SetB(person2);

person1->SetMediator(mediator);

person2->SetMediator(mediator);

person1->SendMessage("我想在南京路附件租房子，价格800一月\n");

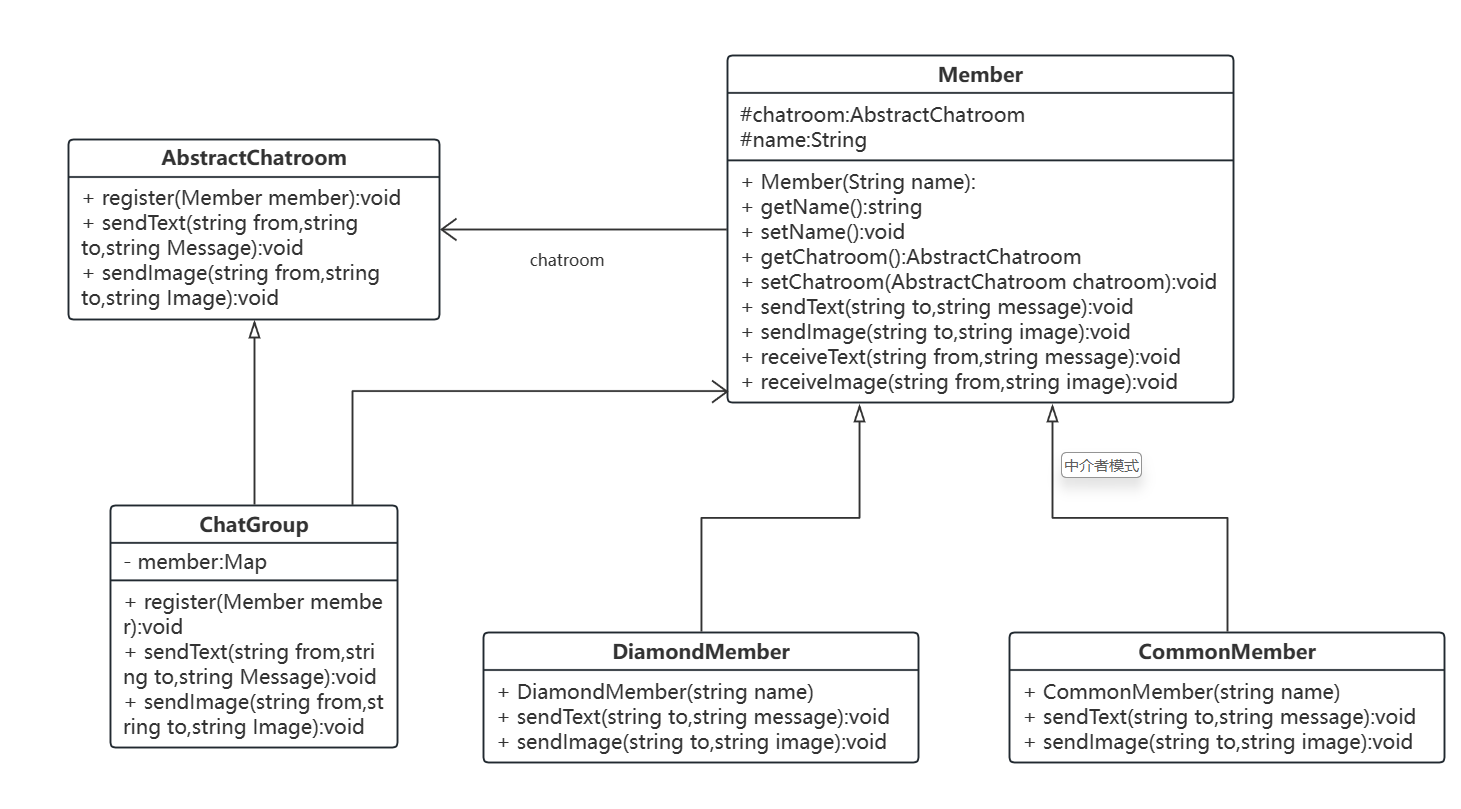
person2->SendMessage("出租房子：南京路100号，70平米，1000元一个月\n");

delete person1, person2, mediator;

return 0;

}

## 17.4模式实例



class AbstractChatRoom;

class Member

{

public:

Member(string name) :m\_name(name){}

string getname() { return m\_name; }

void setChatRoom(AbstractChatRoom\* chatRoom) { m\_chatRoom = chatRoom; }

virtual void SendText(string to, string message) = 0;

virtual void SendImage(string to, string image) = 0;

virtual void ReceiveText(string from, string message) = 0;

virtual void ReceiveImage(string from, string image) = 0;

protected:

AbstractChatRoom\* m\_chatRoom;

string m\_name;

};

class AbstractChatRoom

{

public:

virtual void Register(Member\* member) = 0;

virtual void SendText(string from, string to, string message) = 0;

virtual void SendImage(string from, string to, string image) = 0;

};;

class DiamondMember :public Member

{

public:

DiamondMember(string name) :Member(name){}

void SendText(string to, string message)

{

cout << "钻石会员"<<m\_name<<"发送信息" << endl;

m\_chatRoom->SendText(m\_name, to, message);

}

void SendImage(string to, string image)

{

cout << "钻石会员"<<m\_name<<"发送图片" << endl;

m\_chatRoom->SendImage(m\_name, to, image);

}

void ReceiveText(string from, string message)

{

cout << "钻石会员"<<m\_name<<"接收信息" << endl;

cout << from << "发送信息给" << m\_name << "内容为" << message << endl;

}

void ReceiveImage(string from, string image)

{

cout << "钻石会员"<<m\_name<<"接收图片" << endl;

cout << from << "发送图片给" << m\_name << "内容为" << image << endl;

}

};

class CommonMember :public Member

{

public:

CommonMember(string name) :Member(name) {}

void SendText(string to, string message)

{

cout << "普通会员" << m\_name << "发送信息" << endl;

m\_chatRoom->SendText(m\_name, to, message);

}

void SendImage(string to, string image)

{

cout << "普通会员" << m\_name << "不能发送图片" << endl;

}

void ReceiveText(string from, string message)

{

cout << "普通会员" << m\_name << "接收信息" << endl;

cout << from << "发送信息给" << m\_name << "内容为" << message << endl;

}

void ReceiveImage(string from, string image)

{

cout<<"普通会员" << m\_name << "接收图片" << endl;

cout<<from << "发送图片给" << m\_name << "内容为" << image << endl;

}

};

class ChatRoom :public AbstractChatRoom

{

public:

void Register(Member\* member)

{

m\_members.insert(make\_pair(member->getname(), member));

member->setChatRoom(this);

}

void SendText(string from, string to, string message)

{

m\_members[to]->ReceiveText(from, message);

}

void SendImage(string from, string to, string image)

{

m\_members[to]->ReceiveImage(from, image);

}

private:

map<string, Member\*> m\_members;

};

int main()

{

AbstractChatRoom\* chatRoom = new ChatRoom();

Member\* member1 = new DiamondMember("张三");

Member\* member2 = new CommonMember("李四");

chatRoom->Register(member1);

chatRoom->Register(member2);

member1->SendText("李四", "你好");

member2->SendText("张三", "你好");

member1->SendImage("李四", "图片");

member2->SendImage("张三", "图片");

return 0;

}

## 17.5模式优缺点

优点：多对多变成一对多，没有使用中介者模式的时候，同事对象之间的关系通常是多对多的，引入中介者对象后，中介者对象和同事对象的关系变成了一对多，这让对象间的关系更容易理解和实现。

缺点：过度集中化，中介者模式中，如果同事对象的交互非常多，而且比较复杂，当这些交互全部集中到中介者的时候，会导致中介者对象变得十分的复杂，而且难于管理和维护。

## 17.6模式本质

封装交互。

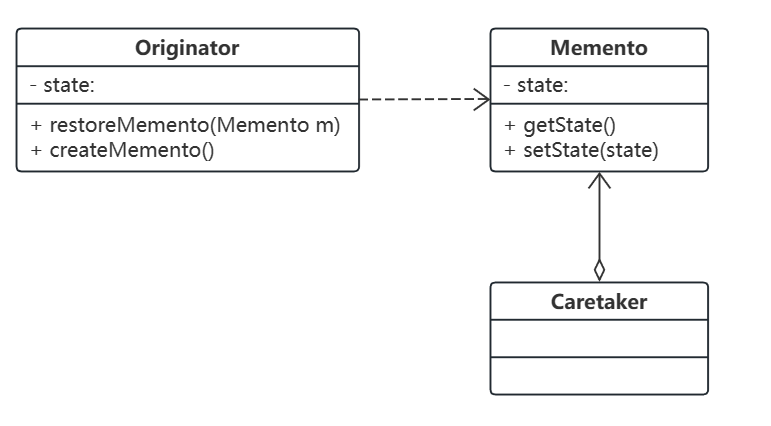
# 18.备忘录模式

## 18.1模式动机及定义

备忘录模式是一种给我们的软件提供后悔药的机制，通过它可以使系统恢复到某一特定的历史状态。

**备忘录模式(Memento Pattern)：**在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以在以后将对象恢复到原先保存的状态。它是一种对象行为型模式，其别名又叫做Token模式或快照模式(Snapshot Pattern) 。

备忘录模式的核心是备忘录类以及用于管理备忘录的负责人类的设计，其结构如下图所示。



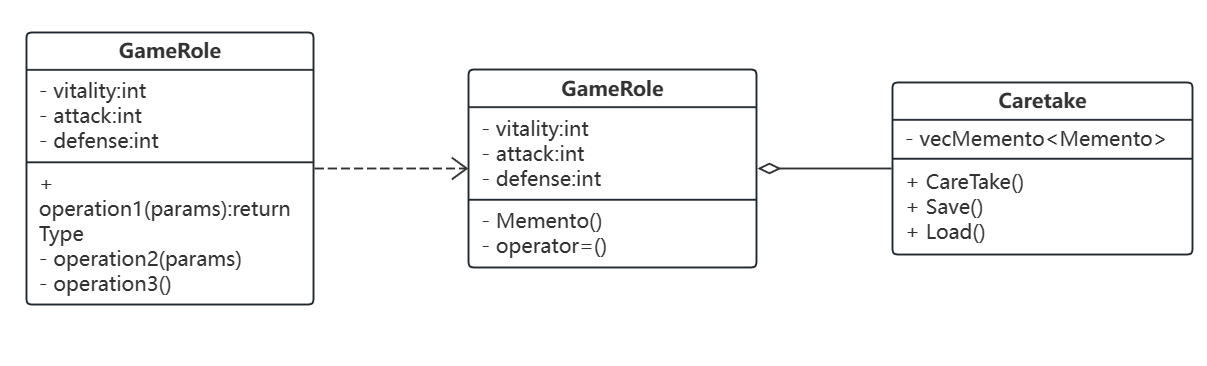
Caretaker：负责人又称为管理者，它负责保存备忘录，但是不能对备忘录的内容进行操作或检查。在负责人类中可以存储一个或多个备忘录对象，它只负责存储对象，而不能修改对象，也无须知道对象的实现细节。

Originator：原发器，它是一个普通类，可以创建一个备忘录，并存储它的当前内部状态，也可以使用备忘录来恢复其内部状态，一般将需要保存内部状态的类设计为原发器。

Memento：备忘录，存储原发器的内部状态，根据原发器来决定保存哪些内部状态。备忘录的设计一般可以参考原发器的设计，根据实际需要确定备忘录类中的属性。需要注意的是，除了原发器本身与负责人类之外，备忘录对象不能直接供其他类使用，原发器的设计在不同的编程语言中实现机制会有所不同。

## 18.2模式分析

保存游戏的进度为例



class Memento

{

friend class GameRole;

private:

Memento(int vitality, int attack, int defense) :vitality(vitality), attack(attack), defense(defense){}

Memento& operator=(const Memento& memento)

{

vitality = memento.vitality;

attack = memento.attack;

defense = memento.defense;

return \*this;

}

private:

int vitality;//生命力

int attack;//攻击力

int defense;//防御力

};

class GameRole

{

public:

GameRole():vitality(100),attack(100),defense(100){}

Memento Save()

{

Memento memento(vitality, attack, defense);

return memento;

}

void Load(Memento memento)

{

vitality = memento.vitality;

attack = memento.attack;

defense = memento.defense;

}

void Show()

{

cout << "生命力：" << vitality << endl;

cout << "攻击力：" << attack << endl;

cout << "防御力：" << defense << endl;

}

void Attack()

{

vitality -= 10;

attack -= 10;

defense -=10;

}

private:

int vitality;//生命力

int attack;//攻击力

int defense;//防御力

};

class Caretaker

{

public:

void Save(Memento memento)

{

m\_mementos.push\_back(memento);

}

Memento Load(int index)

{

return m\_mementos[index];

}

private:

vector<Memento> m\_mementos;

};

int main()

{

Caretaker caretaker;

GameRole gameRole;

gameRole.Show();

caretaker.Save(gameRole.Save());

gameRole.Attack();

gameRole.Show();

//恢复

gameRole.Load(caretaker.Load(0));

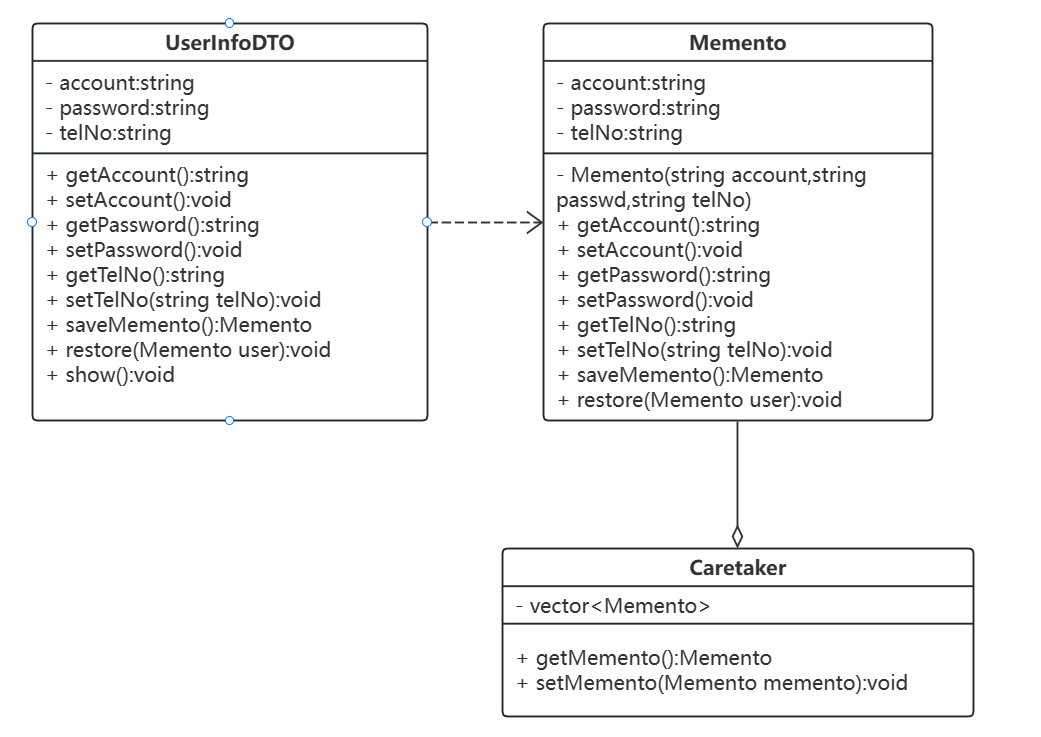
gameRole.Show();

return 0;

}

## 18.3模式实例

用户信息操作撤销



class Memento

{

friend class userInfo;

private:

Memento(string account, string password, string telNo):account(account), password(password), telNo(telNo){}

void setAccount(string account) { this->account = account; }

void setPassword(string password) { this->password = password; }

void setTelNo(string telNo) { this->telNo = telNo; }

string getAccount() { return account; }

string getPassword() { return password; }

string getTelNo() { return telNo; }

private:

string account;

string password;

string telNo;

};

class userInfo

{

public:

Memento\* createMemento()

{

return new Memento(account, password, telNo);

}

string getAccout() { return account; }

string getPassword() { return password; }

string getTelNo() { return telNo; }

void setAccout(string accout) { this->account = account; }

void setPassword(string password) { this->password = password; }

void setTelNo(string telNo) { this->telNo = telNo; }

void restoreMemento(Memento\* memento)

{

this->account = memento->getAccount();

this->password = memento->getPassword();

this->telNo = memento->getTelNo();

}

void show()

{

cout<<"account:"<<account<<endl;

cout<<"password:"<<password<<endl;

cout<<"telNo:"<<telNo<<endl;

}

private:

string account;

string password;

string telNo;

};

class Caretaker

{

public:

void addMemento(Memento\* memento)

{

mementos.push\_back(memento);

}

Memento\* getMemento(int index)

{

return mementos[index];

}

private:

vector<Memento\*> mementos;

};

int main()

{

userInfo user;

Caretaker caretaker;

user.setAccout("zhangsan");

user.setPassword("123456");

user.setTelNo("123456789");

cout<<"初始状态"<<endl;

user.show();

//保存状态

caretaker.addMemento(user.createMemento());

cout << "-----------------------------------------" << endl;

user.setPassword("654321");

user.setTelNo("987654321");

cout<<"修改后状态"<<endl;

user.show();

cout << "-----------------------------------------" << endl;

//恢复状态

user.restoreMemento(caretaker.getMemento(0));

cout<<"恢复后状态"<<endl;

user.show();

return 0;

}

## 18.4模式优缺点

优点：它提供了一种状态恢复的实现机制，使得用户可以方便地回到一个特定的历史步骤，当新的状态无效或者存在问题时，可以使用暂时存储起来的备忘录将状态复原。

备忘录实现了对信息的封装，一个备忘录对象是一种原发器对象状态的表示，不会被其他代码所改动。备忘录保存了原发器的状态，采用列表、堆栈等集合来存储备忘录对象可以实现多次撤销操作。

缺点：资源消耗过大，如果需要保存的原发器类的成员变量太多，就不可避免需要占用大量的存储空间，每保存一次对象的状态都需要消耗一定的系统资源。

## 18.5模式本质

备忘录模式的本质：保存和恢复内部状态。

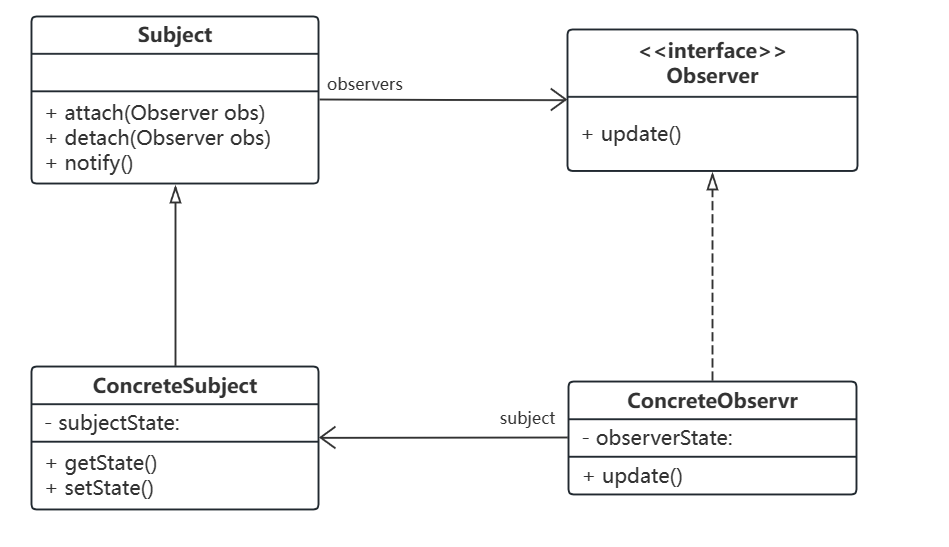
保存是手段，恢复才是目的。

# 19.观察者模式

## 19.1模式动机及定义

观察者模式(Observer Pattern)：定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。

观察者模式是使用频率最高的设计模式之一，它用于建立一种对象与对象之间的依赖关系，一个对象发生改变时将自动通知其他对象，其他对象将相应作出反应。在观察者模式中，发生改变的对象称为**观察目标**，而被通知的对象称为**观察者**，一个观察目标可以对应多个观察者，而且这些观察者之间可以没有任何相互联系，可以根据需要增加和删除观察者，使得系统更易于扩展。



Observer：观察者将对观察目标的改变做出反应，观察者一般定义为接口，该接口声明了更新数据的方法update()，因此又称为抽象观察者。

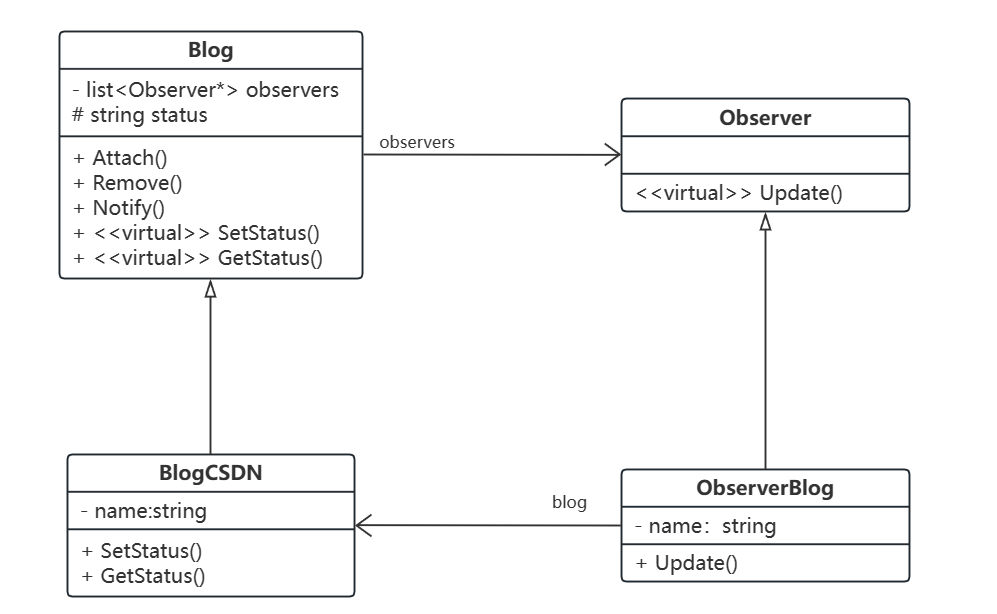
ConcreteObserver：具体观察者中维护一个指向具体目标对象的引用，它存储具体观察者的有关状态，这些状态需要和具体目标的状态保持一致；它实现了在抽象观察者中定义的update()方法。通常在实现时，可以调用具体目标类的attach()方法将自己添加到目标类的集合中或通过detach()方法将自己从目标类的集合中删除。

Subject：目标又称为主题，它是指被观察的对象。在目标中定义了一个观察者集合，一个观察目标可以接受任意数量的观察者来观察，它提供一系列方法来增加和删除观察者对象，同时它定义了通知方法notify()。目标类可以是接口，也可以是抽象类或具体类。

ConcreteSubject：具体目标是目标类的子类，通常它包含有经常发生改变的数据，当它的状态发生改变时，向它的各个观察者发出通知；同时它还实现了在目标类中定义的抽象业务逻辑方法（如果有的话）。如果无须扩展目标类，则具体目标类可以省略。

## 19.2模式分析

可以举个博客订阅的例子，当博主发表新文章的时候，即博主状态发生了改变，那些订阅的读者就会收到通知，然后进行相应的动作，比如去看文章，或者收藏起来。



class Observer

{

public:

Observer(){}

virtual ~Observer() {};

virtual void Update()=0;

};

class Blog

{

public:

Blog(){}

virtual ~Blog() {}

void Attach(Observer\* observer) { observers.push\_back(observer); }

void Remove(Observer\* observer) { observers.remove(observer); }

void Notify() //通知观察者

{

list<Observer\*>::iterator iter = observers.begin();

for (iter; iter != observers.end(); iter++)

(\*iter)->Update();

}

virtual void SetStatus(string s) { status = s; }//设置状态

virtual string GetStatus() { return status; }

private:

list<Observer\*> observers;

protected:

string status;

};

class BlogCSDN :public Blog

{

public:

BlogCSDN(string name) :name(name) {};

virtual void SetStatus(string s) { status = "CSDN通知：" + name + s; }//设置状态

string GetStatus() { return status; }

private:

string name;//博主名称

};

class ObserverBlog :public Observer

{

public:

ObserverBlog(string name,Blog\*blog):name(name),blog(blog){}

void Update()//获取更新状态

{

string status = blog->GetStatus();

cout << name << "-----------------" << status << endl;

}

private:

string name;//观察者名称

Blog\* blog;//观察的博客

};

int main()

{

Blog\* blog = new BlogCSDN("zgr0629");

Observer\* observer1 = new ObserverBlog("tlm", blog);

Observer\* observer2 = new ObserverBlog("qqqq", blog);

blog->Attach(observer1);

blog->Attach(observer2);

blog->SetStatus("发表xxxxx");

blog->Notify();

delete blog, observer1, observer2;

return 0;

}

## 19.3模式实例

猫、狗与老鼠

假设猫是观察目标，老鼠和狗是观察者，猫叫，则老鼠跑，狗也跟着叫，使用观察者模式描述该过程。

class Observer

{

public:

virtual void response() = 0;

};

class Subject

{

public:

void attach(Observer\*observer) { observers.push\_back(observer); }

void remove(Observer\* observer) { observers.remove(observer); }

virtual void cry() = 0;

protected:

list<Observer\*> observers;

};

class Cat :public Subject

{

void cry()

{

cout << "猫叫！" << endl;

cout << "-------------------" << endl;

for (auto iter = observers.begin(); iter != observers.end(); iter++)

(\*iter)->response();

}

};

class Dog :public Observer

{

void response() { cout << "狗跟着叫" << endl; }

};

class Mouse :public Observer

{

void response() { cout << "老鼠努力逃跑" << endl; }

};

int main()

{

Subject\* cat = new Cat();

Observer\* obs1 = new Dog();

Observer\* obs2 = new Mouse();

cat->attach(obs1);

cat->attach(obs2);

cat->cry();

return 0;

}

## 19.4 模式优缺点

优点：

* 实现了观察者和目标之间的抽象耦合，原本目标对象在状态改变时，需要直接调用所有的观察者对象，但是抽象出观察者接口后，目标和观察者就只是在抽象层面上耦合了，即目标只是知道观察者接口，并不知道具体的观察者的类，从而实现目标类和具体观察者类之间解耦。
* 实现了动态联动，所谓联动，就是做一个操作会引起其它相关的操作。由于观察者模式对观察者注册实行管理，那就可以在运行期间，通过动态的控制注册的观察者，来控制某个动作的联动范围，从而实现动态联动。
* 支持广播通信，由于目标发送通知给观察者是面向所有注册的观察者，所以每次目标通知的信息就要对所有注册的观察者进行广播。当然，也可以通过在目标上添加新的功能来限制广播的范围。
* 观察者模式满足“开闭原则”的要求，增加新的具体观察者无须修改原有系统代码，在具体观察者与观察目标之间不存在关联关系的情况下，增加新的观察目标也很方便。

缺点：

* 由于观察者模式每次都是广播通信，不管观察者需不需要，每个观察者都会被调用update方法，如果观察者不需要执行相应处理，要及时解注册。
* 若两个对象互为观察者和目标对象，要注意避免相互广播造成死循环、有可能导致系统崩溃的问题。

## 19.5模式本质

观察者模式的本质：触发联动。

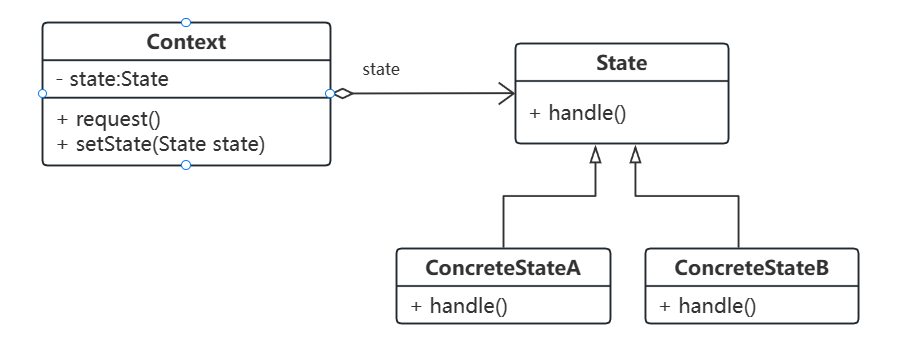
# 20.状态模式

20.1模式动机及定义

状态模式(State Pattern)：允许一个对象在其内部状态改变时改变它的行为，对象看起来似乎修改了它的类。其别名为状态对象(Objects for States)，状态模式是一种对象行为型模式。

状态模式将一个对象的状态从该对象中分离出来，封装到专门的状态类中，使得对象状态可以灵活变化，对于客户端而言，无须关心对象状态的转换以及对象所处的当前状态，无论对于何种状态的对象，客户端都可以一致处理。

## 20.2模式结构



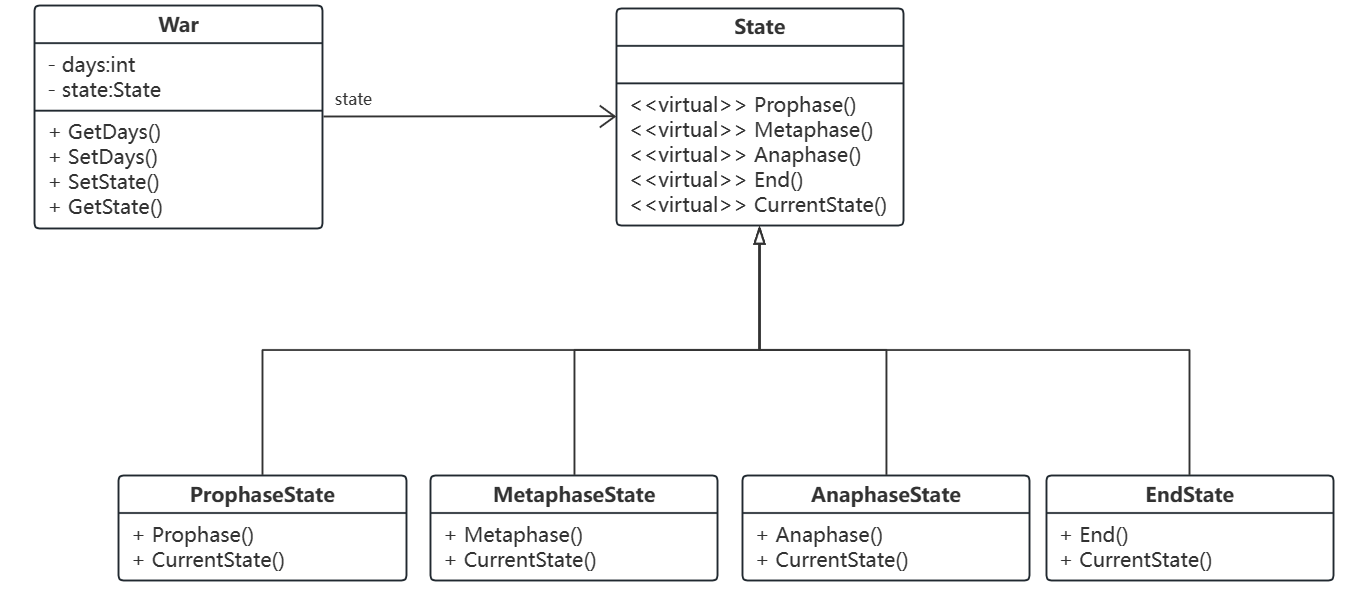
Context：环境类，又称为上下文类，它是拥有多种状态的对象。由于环境类的状态存在多样性且在不同状态下对象的行为有所不同，因此将状态独立出去形成单独的状态类。在环境类中维护一个抽象状态类State的实例，这个实例定义当前状态，在具体实现时，它是一个State子类的对象。

State：抽象状态类，它用于定义一个接口以封装与环境类的一个特定状态相关的行为，在抽象状态类中声明了各种不同状态对应的方法，而在其子类中实现类这些方法，由于不同状态下对象的行为可能不同，因此在不同子类中方法的实现可能存在不同，相同的方法可以写在抽象状态类中。

ConcreteState：具体状态类，它是抽象状态类的子类，每一个子类实现一个与环境类的一个状态相关的行为，每一个具体状态类对应环境的一个具体状态，不同的具体状态类其行为有所不同。

## 20.3状态模式

状态模式：允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。它有两种使用情况：（1）一个对象的行为取决于它的状态, 并且它必须在运行时刻根据状态改变它的行为。（2）一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。本文的例子为第一种情况，以战争为例，假设一场战争需经历四个阶段：前期、中期、后期、结束。当战争处于不同的阶段，战争的行为是不一样的，也就说战争的行为取决于所处的阶段，而且随着时间的推进是动态变化的。



class War;

class State

{

public:

virtual void CurrentState(War\* war) = 0;

};

class War

{

public:

War(State\*state):state(state),day(0){}

int GetDay() { return day; }

void SetDay(int day) { this->day = day; }

void SetState(State\* state) { this->state = state; }

void getState() { state->CurrentState(this); }

private:

State\* state;

int day;

};

class EndState :public State

{

public:

void End(War\* war)

{

cout << "第" << war->GetDay() << "天，结束" << endl;

}

void CurrentState(War\* war) { End(war); }

};

class AnaphaseState :public State

{

public:

void Anaphase(War\* war) //后期具体行为

{

if (war->GetDay() < 30)

cout << "第" << war->GetDay() << "天，后期" << endl;

else

{

war->SetState(new EndState());

war->getState();

}

}

void CurrentState(War\* war) { Anaphase(war); }

};

class MetaphaseState :public State

{

public:

void Metaphase(War\* war) //中期具体行为

{

if (war->GetDay() < 20)

cout << "第" << war->GetDay() << "天，中期" << endl;

else

{

war->SetState(new AnaphaseState());

war->getState();

}

}

void CurrentState(War\* war) { Metaphase(war); }

};

class ProphaseState :public State

{

public:

void Prophase(War\* war) //前期具体行为

{

if(war->GetDay()<10)

cout<<"第"<<war->GetDay()<<"天，准备"<<endl;

else

{

war->SetState(new MetaphaseState());

war->getState();

}

}

void CurrentState(War\* war) { Prophase(war); }

};

int main()

{

War\* war = new War(new ProphaseState());

for (int i = 0; i < 40; i++)

{

war->SetDay(i);

war->getState();

}

return 0;

}

## 20.4模式实例

论坛用户等级。在某论坛系统中，用户可以发表留言，发表留言将增加积分；用户也可以回复留言，回复留言也将增加积分；用户还可以下载文件，下载文件将扣除积分。该系统用户分为三个等级，分别是新手、高手和专家，这三个等级对应三种不同的状态，这三种状态分别定义如下：(1) 如果积分小于100分，则为新手状态，用户可以发表留言、回复留言，但是不能下载文件。 (2) 如果积分大于等于100分但小于1000分，则为高手状态，用户可以发表留言、回复留言，还可以下载文件，而且用户在发表留言时可以获取双倍积分。如果下载文件后积分小于0，则不能下载该文件。(3) 如果积分大于等于1000分，则为专家状态，用户可以发表留言、回复留言和下载文件，用户除了在发表留言时可以获取双倍积分外，下载文件只扣除所需积分的一半。如果下载文件后积分小于0，则不能下载该文件。

## 20.5模式优缺点

更好的扩展性，引入了状态处理的公共接口后，使得扩展新的状态变得非常容易，只需要新增加一个实现状态处理的公共接口的实现类，然后在进行状态维护的地方，设置状态变化到这个新的状态即可。

显式化进行状态转换，状态模式为不同的状态引入独立的对象，使得状态的转换变得更加明确。而且状态对象可以保证上下文不会发生内部状态不一致的情况，因为上下文中只有一个变量来记录状态对象，只要为这一个变量赋值就可以了。

状态模式的使用必然会增加系统中类和对象的个数，导致系统运行开销增大。

状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱，增加系统设计的难度。

状态模式对“开闭原则”的支持并不太好，增加新的状态类需要修改那些负责状态转换的源代码，否则无法转换到新增状态；而且修改某个状态类的行为也需修改对应类的源代码。

## 20.6模式本质

状态模式的本质：根据状态来分离和选择行为。

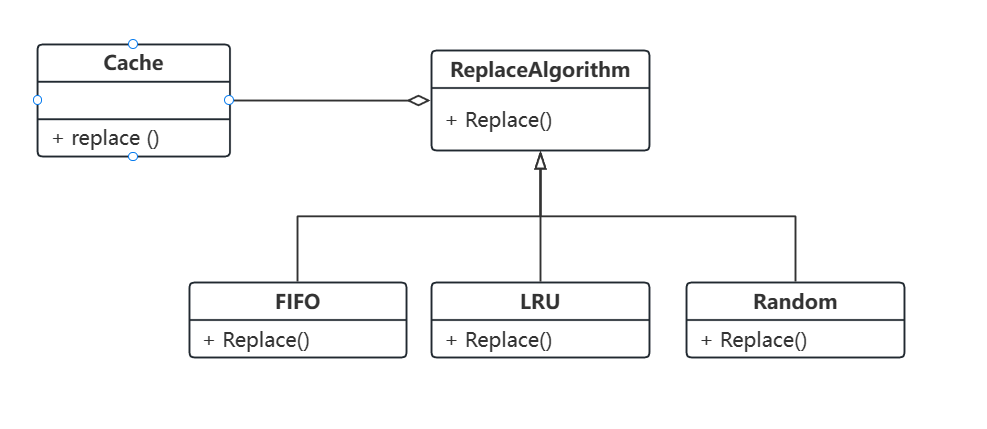
# 21.策略模式

## 21.1模式动机及定义

完成一项任务，往往可以有多种不同的方式，每一种方式称为一个策略，我们可以根据环境或者条件的不同选择不同的策略来完成该项任务。

策略模式(Strategy Pattern)：定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法独立于使用它的客户而变化，也称为政策模式(Policy)。策略模式是一种对象行为型模式。

## 21.2模式分析



class ReplaceAlgorithm

{

public:

virtual void Replace() = 0;

};

class LRU :public ReplaceAlgorithm

{

public:

virtual void Replace() { cout << "Least Recently Used replace algorithm" << endl; }

};

class FIFO :public ReplaceAlgorithm

{

public:

virtual void Replace() { cout << "First in First out replace algorithm" << endl; }

};

class Random :public ReplaceAlgorithm

{

public:

virtual void Replace() { cout << "Random replace algorithm" << endl; }

};

class Cache

{

public:

Cache(ReplaceAlgorithm\* ra):ra(ra){}

~Cache() { delete ra; }

void Replace() { ra->Replace(); }

private:

ReplaceAlgorithm\* ra;

};

int main()

{

ReplaceAlgorithm\* ra = new FIFO();

Cache cache(ra);

cache.Replace();

}