Beijing Forest Studio

北京理工大学信息系统及安全对抗实验中心

设计模式简介

闫晗　硕士研究生

2019年04月28日

内容提要

- 背景介绍
  - 总体目标
  - 预期收获
- 基础知识
  - 设计模式与架构模式
  - 面向对象的程序设计
- 基本原理
  - 创建型模式
  - 结构型模式
  - 行为型模式
- 应用总结
- 参考文献

设计模式简介

背景介绍

背景介绍

- “每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心，这样，你就能够一次又一次地使用该方案而不必重复劳动。”

——Christopher Alexander

- 《设计模式：可复用面向对象软件的基础》
  - 可复用：设计模式的最终目标
  - 面向对象：设计模式的实现手段

背景介绍

- 总体目标

T	增强程序的可维护性、可复用性及可扩展性
I	某类重复发生的设计问题
P	基于封装、继承、多态等面向对象特性分析问题
O	高内聚、低耦合的设计方案

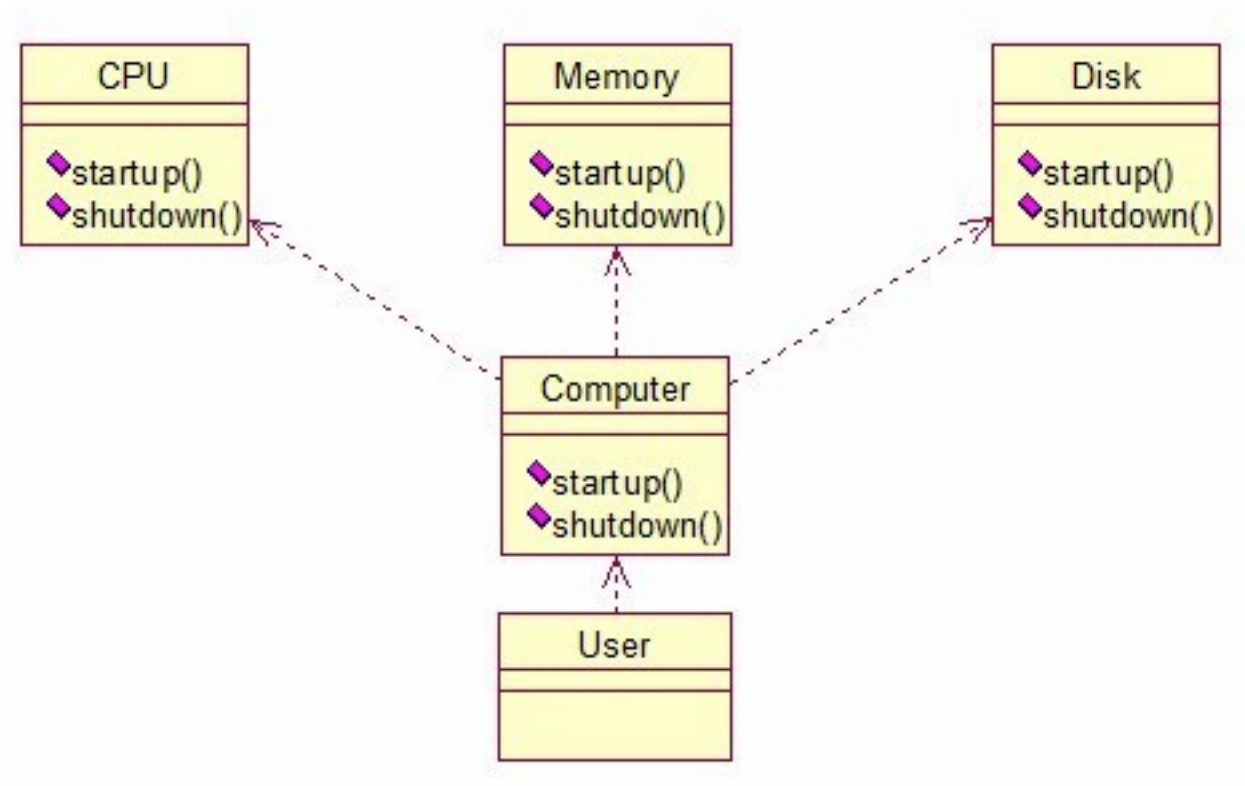
- 预期收获
  - 理解面向对象编程思想
  - 理解软件设计模式的基本原则
  - 了解常见设计模式的背景及设计方法

设计模式简介

基础知识

基础知识

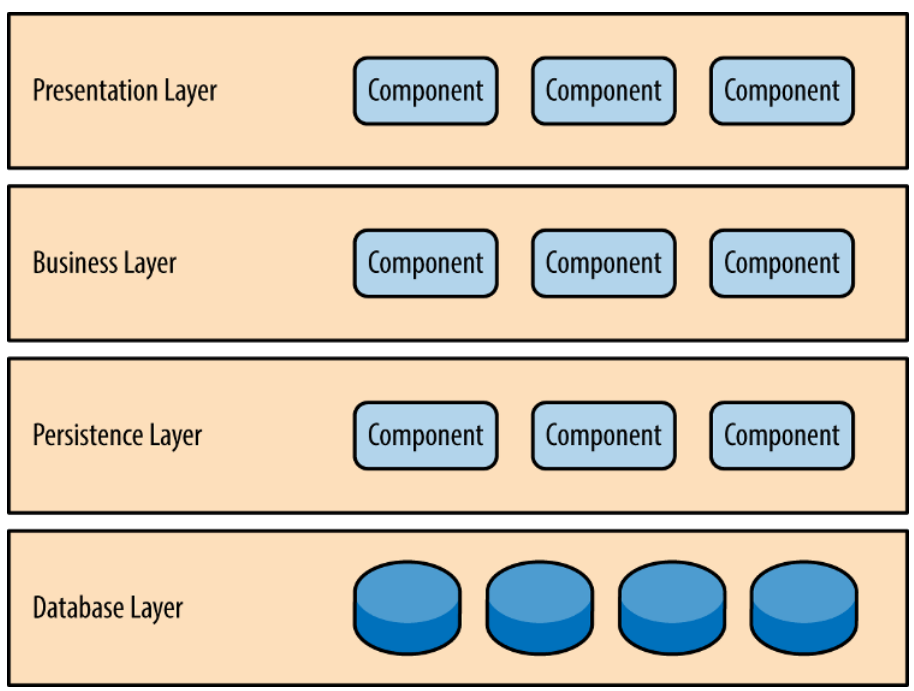
- 设计模式与架构模式
  - 设计模式
    - 定义：设计模式描述了一组**类和接口**的关系，用于解决特定上下文内的某个常见的设计问题
    - 举例：工厂方法模式、外观模式、适配器模式等



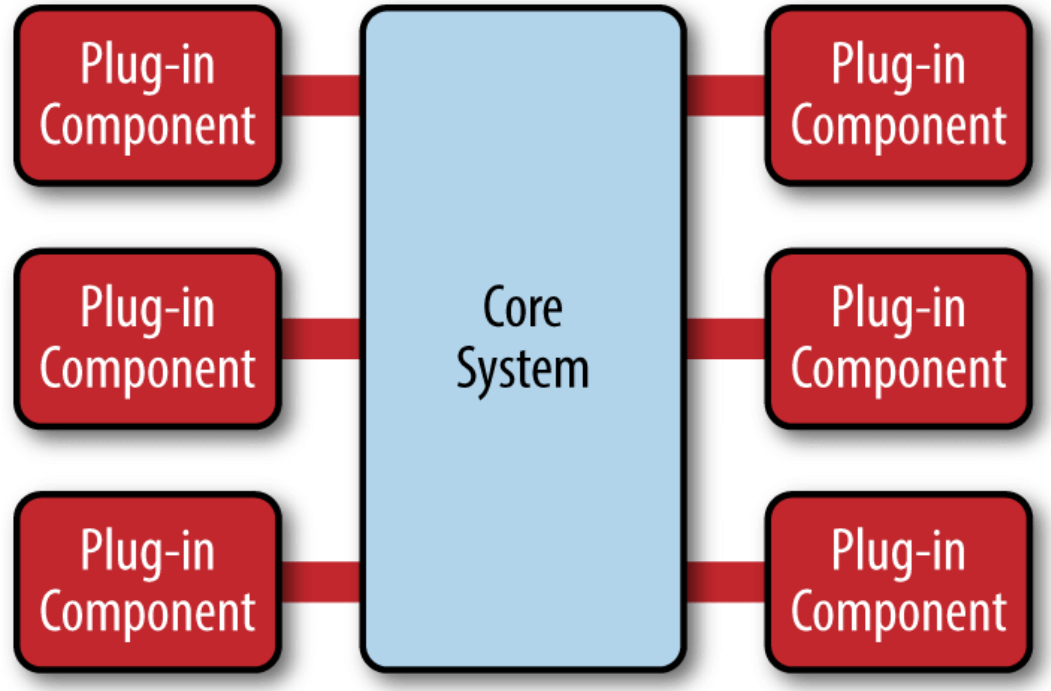
外观模式类图

基础知识

- 设计模式与架构模式
  - 架构模式
    - 定义：架构模式描述了一组**组件**之间的关系，用于解决特定上下文内的某个常见的架构问题
    - 举例：MVC模式、分层模式、微核模式等



分层架构图



微核架构图

基础知识

- 面向对象的程序设计
  - 具象层面：理解面向对象的基本机制
    - 封装：隐藏内部实现
    - 继承：复用已有代码
    - 多态：改写对象行为
  - 抽象层面：理解面向对象的具体内涵
    - 基于面向对象语言的程序设计≠面向对象的程序设计
    - 面向对象设计的优势：**应对变化**

基础知识

- 面向对象的程序设计
  - 例：根据员工的类型计算工资并打印（结构化实现）

```
//枚举
enum EmployeeType{
   Saler,
   Manager
   ...};

//计算工资
int salary(EmployeeType type){
   if(type==EmployeeType.Saler){
   ...
   }
   else if(...){
   ...}
}
```

```
//打印工资
void main(){
   if(type==EmployeeType.Saler){
   ...
   }
   else if(...){
   ...
   }
}
```

基础知识

- 面向对象的程序设计
  - 例：根据员工的类型计算工资并打印（面向对象实现）

```
//抽象类
abstract class Employee{
   private int salary;
   public abstract int getSalary();
}

//实现类
class Saler extends Employee{
   private int salary;
   public int getSalary(){
   ...
   }
}
```

```
//打印工资
public static void main(String[] args){
   Employee e =
      EmFactory.getEmployee(id);
   System.out.println(e.getSalary());
}
```

基础知识

- 面向对象的程序设计
  - 例：新增员工类型（结构化实现）

```
//枚举
enum EmployeeType{
   Saler,
   Manager,
   Engineer
};

//计算工资
int salary(EmployeeType type){
   if(type==EmployeeType.Engineer){
   ...
   }
   ...
}
```

```
//打印
void main(){
   if(type==EmployeeType.Engineer){
   ...
   }
   ...
}
```

基础知识

- 面向对象的程序设计
  - 例：新增员工类型（面向对象实现）

```
//抽象类
abstract class Employee{
   private int salary;
   public abstract int getSalary();
}

//实现类
class Engineer extends Employee{
   private int salary;
   public int getSalary(){
   ...
   }
}
```

```
//打印
public static void main(String[] args){
   Employee e =
      EmFactory.getEmployee(id);
System.out.println(e.getSalary());}
```

基础知识

- 面向对象的程序设计
  - 面向对象的具体内涵
    - 宏观角度：隔离变化
      - 例中，新增操作的变化被隔离在实现类中
    - 微观角度：各司其职
      - 新增的类型不应该对原有的代码造成影响
  - 面向对象中的“对象”
    - 语言层面：对象封装了属性和方法
    - 规格层面：对象是可被其它对象使用的公共接口
    - 概念层面：对象是拥有某种责任的抽象

基础知识

- 面向对象的程序设计
  - 面向对象的设计原则
    - 依赖倒置原则：
      - 高层模块不依赖于底层模块，二者都依赖于抽象
      - 抽象不依赖于实现细节，实现细节依赖于抽象

```
//抽象类
abstract class Employee{
   private int salary;
   public abstract int getSalary();}

//实现类
class Engineer extends Employee{
   public int getSalary(){
   ...}}
```

```
//打印
public static void main(String[] args){
   Employee e =
      EmFactory.getEmployee(id);
System.out.println(e.getSalary());}
```

基础知识

- 面向对象的程序设计
  - 面向对象的设计原则
    - 依赖倒置原则
      - 例：编写测试任意代码执行时间的程序

```
//测试代码执行时间
public long testTime(){ //问题：高层模块不稳定
   long start = 0;
   long end =  0;
   start = System.currentTimeMillis();
   //执行待测试代码
   end  = System.currentTimeMillis();
   return end-start;
}
```

基础知识

- 面向对象的程序设计
  - 面向对象的设计原则
    - 依赖倒置原则
      - 例：编写测试任意代码执行时间的程序

```
//抽象接口
Interface Runnable{
   public void run();//抽象方法
}
//实现类
class T implements Runnable{
   public void run(){
      //Code...
  }
}
```

```
//测试代码执行时间
public long testTime(Runnable r){
   long start = 0;
   long end =  0;
   start = System.currentTimeMillis();
   r.run();
   end  = System.currentTimeMillis();
   return end-start;
}
```

基础知识

- 面向对象的程序设计
  - 面向对象的设计原则
    - 依赖倒置原则
      - 接口回调

//Java多线程

```
public static void main(String[] args){
   Thread t = new Thread(
      new Runnable(){
         public void run(){...}
   })
}
```

//Android监听器

```
protected void onCreate(Bundle b){
   btn.setOnClickListener(
      new OnClickListener(){
         public void onClick(View v){...}
   })
}
```

基础知识

- 面向对象的程序设计
  - 面向对象的设计原则
    - 开闭原则：
      - 对扩展开放，对更改封闭
      - 类模块应为可扩展但不可修改的

```
//枚举
enum EmployeeType{
   Saler,
   Engineer};

//计算工资
int salary(EmployeeType type){
   if(type==
      EmployeeType.Engineer){
   ...}...}
```

```
//抽象类
abstract class Employee{
   private int salary;
   public abstract int getSalary();}

//实现类
class Engineer extends Employee{
   private int salary;
   public int getSalary(){
   ...}}
```

基础知识

- 面向对象的程序设计
  - 面向对象的设计原则
    - 单一职责原则：
      - 一个类应该仅有一个引起它变化的原因
      - 类变化的方向与类的责任有关
    - 里氏替换原则：
      - 子类必须能够替换它们的基类
    - 接口隔离原则
      - 不应该强迫客户端实现不必要的方法
    - 优先使用对象组合，而不是类继承
      - 继承在某种程度上破坏了封装性

设计模式简介

基本原理

基本原理

- GOF-23设计模式分类
  - 创建型模式：将对象的部分创建工作延迟到子类或其他对象，从而应对需求变化为**对象创建**带来的冲击
  - 结构型模式：通过类继承或者对象组合来获得更灵活的结构，从而应对需求变化为**对象结构**带来的冲击
  - 行为型模式：通过类继承或对象组合来划分类与对象间的职责，从而应对需求变化为**对象交互**带来的冲击

基本原理

- 创建型模式
  - 分类
    - 对象创建：通过该模式绕开new操作，避免使用new创建对象过程中所导致的紧耦合，从而支持对象创建的稳定性。典型的对象创建模式包括**工厂方法模式**、**建造者模式**等
    - 对象性能：面向对象较好地解决了“抽象”的问题，但是不可避免地要付出一定的代价。通常情况下，面向对象的成本可以忽略不计，但是在某些情况下则需要谨慎处理。典型的对象性能模式包括**单例模式**、**享元模式**等

基本原理

- 创建型模式
  - 工厂方法模式
    - 分类：属于对象创建模式
    - 背景：在软件系统中，经常面临着创建对象的工作。由于需求的变化，需要创建的对象的具体类型经常变化。如何提供一种“封装机制”避免客户程序和创建工作**紧耦合**?
    - 实例：编写“发送消息”服务并提供给客户端使用

```
//提供发送邮件服务
class MailSender{
   public void send(){
      ...//发送邮件
   }
}
```

```
//客户端（调用者）
public void Notice(){
   MailSender sender = new MailSender();
   sender.send();
}
```

未定义接口，服务不具备扩展性

基本原理

- 创建型模式
  - 工厂方法模式
    - 实例：编写“发送消息”服务并提供给客户端使用

```
//抽象成接口
Interface ISender{
   public void send();
}
//提供发送邮件服务
class MailSender
 implements ISender{
   public void send(){
      ...//发送邮件
   }
}
```

```
//客户端
public void Notice(){
   ISender sender = new MailSender();
   sender.send();
}
```

未实现**解耦**，需要绕开new（在不使用反射的前提下）

基本原理

- 创建型模式
  - 工厂方法模式
    - 实例：编写“发送消息”服务并提供给客户端使用

```
//抽象成接口
Interface ISender{
   public void send();
}
//提供发送邮件服务
class MailSender
 implements ISender{
   public void send(){
      ...//发送邮件
   }
}
```

```
//创造Sender对象
class Factory{
   public ISender getSender(){
      return new MailSender();}}
//客户端
public void Notice(){
   Factory f = new Factory();
   ISender sender = f.getSender();
   sender.send();
}
```

定义工厂，但仍未实现解耦

基本原理

- 创建型模式
  - 工厂方法模式
    - 实例：编写“发送消息”服务并提供给客户端使用

```
//将发送抽象成接口
Interface ISender{
   public void send();
}

//将工厂抽象成接口
Interface Factory{
   public ISender getSender();
}
```

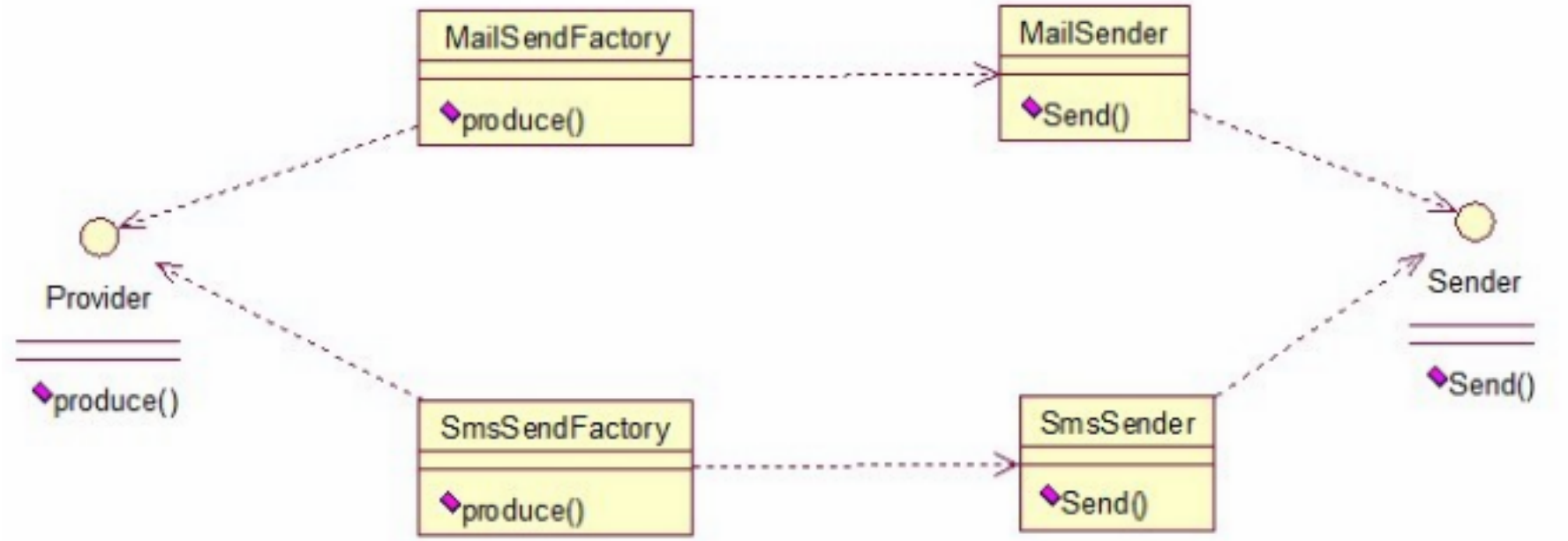
```
//定义MailSender的工厂
class MFactory implements Factory{
   public ISender getSender(){
      return new MailSender();}}

//客户端
public void Notice(Factory f){
   ISender sender = f.getSender();
   sender.send();}
```

已实现解耦

基本原理

- 创建型模式
  - 工厂方法模式
    - UML类图



工厂方法模式类图

基本原理

- 创建型模式
  - 单例模式
    - 分类：属于对象性能模式
    - 背景：在软件系统中，经常有这样一些特殊的类，必须保证它们在系统中只存在一个实例，才能确保它们的逻辑正确性及良好的效率，如何提供一种机制保证单例?
    - 思路：私有构造函数，创建静态对象，提供静态访问接口
    - 实现：懒汉式，饿汉式

基本原理

- 创建型模式
  - 单例模式
    - 懒汉式：类加载时不创建实例

```
class LazySingleton{

    private static LazySingleton instance = null;

    private LazySingleton(){}

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

基本原理

- 创建型模式
  - 单例模式
    - 饿汉式：类加载时创建实例（**推荐使用**）

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

   private EagerSingleton(){}

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

基本原理

- 结构型模式
  - 分类
    - 单一职责：在软件组件的设计中，如果责任划分不清晰，使用继承得到的结果往往随着需求而变化，导致子类膨胀。解决该问题的关键在于划清职责。典型的单一职责模式包括**装饰器模式、桥接模式**等

    - 接口隔离：在组件构建过程中，某些接口之间直接的依赖常常会带来问题，甚至无法实现。采用添加一层间接的接口来隔离本来互相紧密关联的接口是一种常见解决方案。典型的接口隔离模式包括**外观模式、代理模式**等

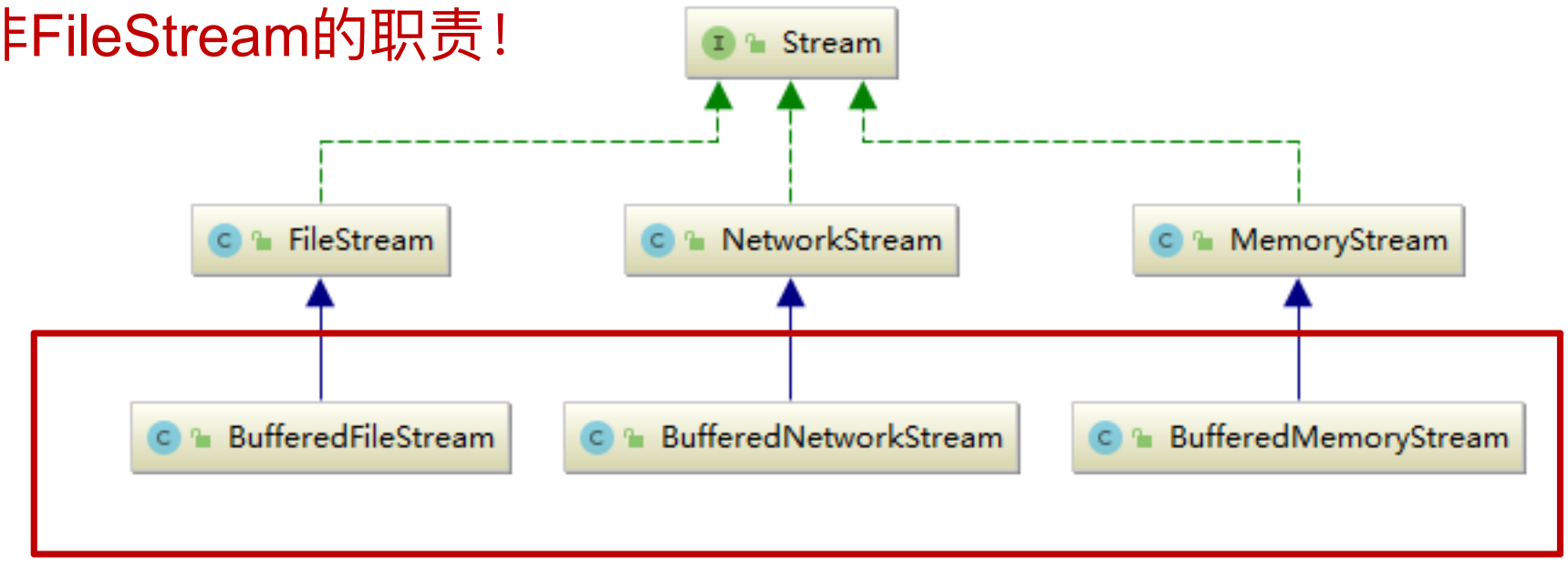
基本原理

- 结构型模式
  - 装饰器模式
    - 分类：属于单一职责模式
    - 背景：在某些情况下，通过继承来扩展对象功能的方法可能被过度地使用。由于继承为类型引入了静态特质，使得这种扩展方式缺乏灵活性。另外，随着子类的增多，各种子类的组合会导致子类的膨胀。何种机制可使对象功能的扩展能够根据需要来动态地实现，同时避免子类膨胀?
    - 实例：增强IO流

基本原理

- 结构型模式
  - 装饰器模式
    - 实例：增强IO流（继承方式导致子类爆炸）

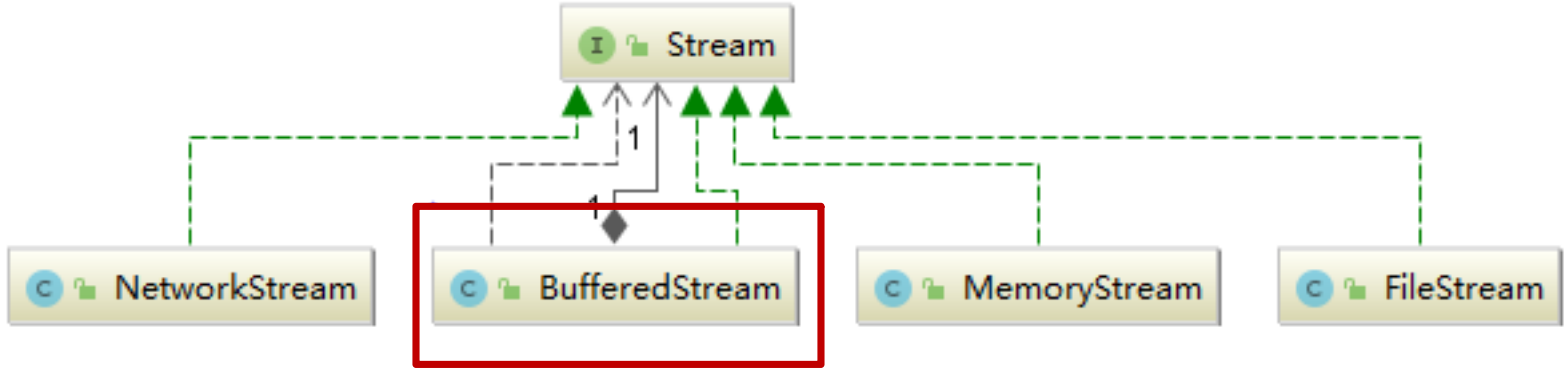
缓冲并非FileStream的职责！



IO流继承树

基本原理

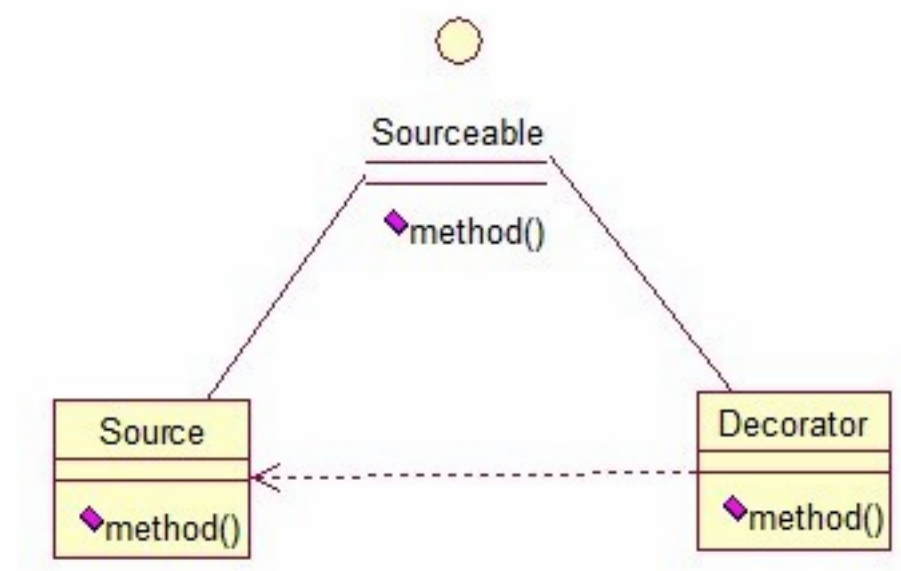
- 结构型模式
  - 装饰器模式
    - 实例：增强IO流（装饰器减少了代码冗余）



IO流继承树

基本原理

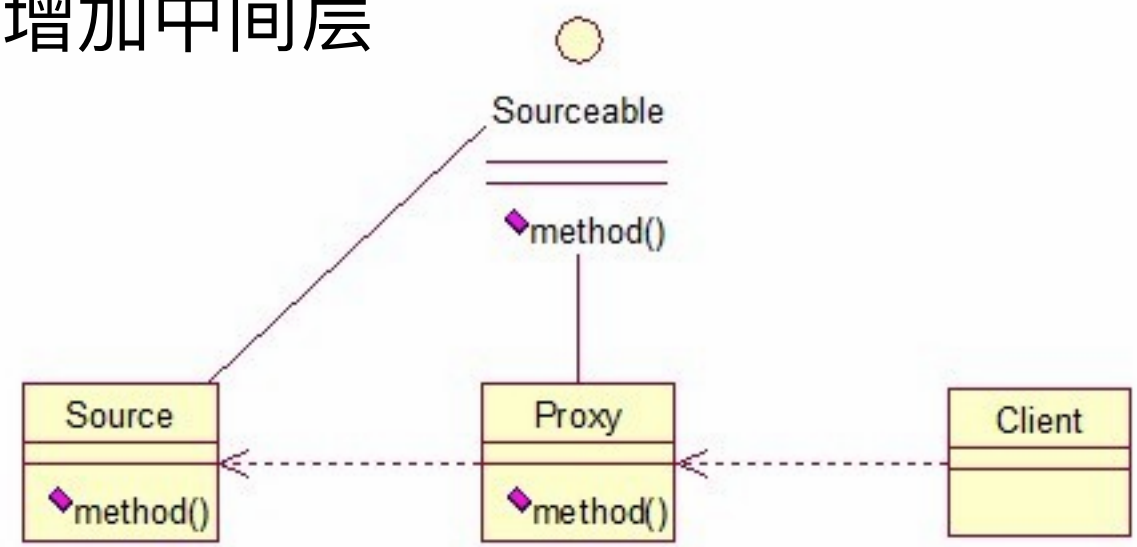
- 结构型模式
  - 装饰器模式
    - 结论：通过**继承**（白盒）和**组合**（黑盒）都可以增强功能，相比之下，**组合**的优先级要高于**继承**。
    - UML类图



装饰器模式类图

基本原理

- 结构型模式
  - 代理模式
    - 分类：属于接口隔离模式
    - 背景：在面向对象系统中，有些对象由于某种原因（例如某些操作需要安全控制等），无法由调用者直接访问。如何在不破坏透明性的前提下，管理这些对象特有的复杂性?
    - 思路：增加中间层



代理模式类图

基本原理

- 行为型模式
  - 分类
    - 组件协作：通过晚期绑定实现框架与程序的松耦合
    - 状态变化：对变化进行管理，维持高层模块的稳定
    - 行为变化：组件行为的变化导致组件本身剧烈变化
    - 领域问题：特定领域变化频繁但可抽象为某种规则

基本原理

- 行为型模式
  - 策略模式
    - 分类：属于组件协作模式
    - 背景：在软件构建过程中，某些对象使用的算法经常会发生改变，如果将这些算法硬编码到对象中，会造成冗余。如何在运行时根据需要透明地更改对象的算法?
- 举例：税额计算（违反了**开-闭原则**的设计）

```
//枚举税种
enum TaxBase{
   CN_Tax,
   US_Tax,
   ...};
```

```
//根据类型计算税额
int getTax(TaxBase t){
   if(CN_Tax == t){...}
   else if(US_Tax==){...}
}
```

基本原理

- 行为型模式
  - 策略模式
    - 举例：税额计算（接口去耦）

```
//算法抽象
interface TaxStrategy{
   int calculate(Context context);
}

//算法实现
class CHTax implements TaxStrategy{
   int calculate(Context context){}
}
...
```

```
//调用算法
public void getTax(SFactory sf){
   TaxStrategy t = sf.getInstance();
    t.calculate();
}
```

设计模式简介

应用总结

应用总结

- 实际应用
  - 工厂设计模式
    - Spring框架中的BeanFactory
  - 单例设计模式
    - Spring框架中的单例bean创建
  - 装饰器模式
    - Java SE中的缓冲流（BufferedReader等）
  - 代理模式
    - Spring AoP

应用总结

- 总结分析
  - 面向对象设计模式是“好的面向对象设计”，“好的设计”指可以满足“**应对变化，提高复用**”的设计
  - 软件设计的基本特征是“**需求的频繁变化**”，理解设计模式的关键之处在于“**寻找变化点**”，在变化点应用设计模式
  - 软件设计应“**面向问题**”而非面向“**设计模式**”，没有一步到位的设计模式，应提倡“**重构获得模式**”

参考文献

参考文献

- [1]Gamma E, Helm R, Johnson R, et al. Design Patterns: Elements of Reusable Object-Oriented Software[J]. 1995.
- [2]菜鸟教程. 设计模式[EB/OL]. http://www.runoob.com/design-pattern/design-pattern-tutorial.html , 2019-04-25.
- [3]CodeTao. 设计模式解析[EB/OL]. https://www.cnblogs.com/geek6/p/3951677.html , 2019-04-25.

大成若缺，其用不弊。
大盈若冲，其用不穷。
大直若屈。大巧若拙。
大辩若讷。静胜躁，寒
胜热。清静为天下正。

谢谢！