1. 隔离变化
2. 各司其职

多态调用，（接口一样实现不一样）

对象就是封装了代码和数据的公共接口

一

依赖倒置原则 DIP

高层模块（稳定）不应该依赖于低层模块（变化），两者都应该依赖于抽象（稳定）

抽象（稳定）不应该依赖与实现细节（变化），实现细节应该依赖于抽象（稳定）

子类要依赖抽象，因为子类变化不会影响抽象，相对于更稳定，

二

开放封闭原则（开闭原则OCP）

对扩展开放，对更改封闭，类模块应该可扩展不可修改

遇到变更需求的时候，不要想着修改重做，要想着增加一些东西，不去修改源代码

三

单一职责原则（SRP）

一个类应该仅有一个引起它变化的原因

变化的方向隐含着类的责任

一个类里面的功能应该是一个变化类型的 （运算类，工具类）如果一个类有多个引起他变化就会很乱

四

Liskov替换原则（LSP）

子类必须能够替换他们的基类

继承表达类型抽象

子类当基类用 父类有的方法，子类必须要有

五

接口隔离原则（ISP）

不应该强迫客户程序依赖他们不用的方法

接口应该小而完备（

接口里的方法，不需要全部public 如果是本类用的话可以私有，子类用可以保护，如果是公共的那就public

）

如果实现接口的对象（客户程序）调用了不相关的方法就会产生依赖，这样很不稳定

六

优先使用对象组合，而不是类继承

对象组合是一个对象里面有另一个对象作为数据类型

继承会破坏封装性，会暴露不必要的东西，而对象组合只需要被组合的对象有良好的外部接口就行，耦合度低

七

封装变化点

利用封装来封装变化点（对象之间的分界层），一侧变化，一侧稳定

八

针对接口编程，而不是针对实现编程（映射业务类型）

不将变量类型声明为某个特定的具体类，而是声明为某个接口

客户程序（调用方法的类）无需知道对象的具体类型，只需要知道对象所具有的接口

这样会减少系统各部分的依赖关系，实现高内聚松耦合

面向接口编程

定义统一的接口化标准，而所有的东西都去实现这个接口标准

设计模式

23种设计模式根据目的分类：

创建型设计模式：主要解决对象创建

结构型设计模式：根据需求变化对对象结构进行冲击

行为型设计模式：多个类之间交互过程中责任的划分

根据范围分类：

类模式：处理类和子类的静态关系（继承关系，父-子）

对象模式：处理对象之间的动态关系（组合关系 类包含类）

从封装角度分类：

组件协作：解决协作问题，通过晚绑定来实现框架与应用程序之间的松耦合

表现明显的有：模板模式、策略模式、事件模式

单一职责：类与类责任划分，如果职责划分的不清晰，会出现很多重复的代码

表现明显的有：装饰者模式

对象创建：解决对象创建的依赖关系

对象性能：解决性能

接口隔离：

状态变化：

数据结构：

行为变化：

领域问题：

设计模式要在在变化点中应用，业务的频繁变化，我们需要设计模式

运用设计模式不适合一下子就用，要以重构的方式一步一步进行修改

重构技法：

静态 ----= 动态 静态绑定到动态绑定

早绑定---= 晚绑定

继承----= 组合

编译时依赖 ----= 运行时依赖

紧耦合 ----= 松耦合

模板设计模式 （看红字，就用模板模式）

动机：一个任务，它整体是稳定的，但是他的各个子模块需要频繁更改业务或者由于一些原因（比如框架和应用）我没有办法和任务的整体结构同时实现

需求：我怎么保证任务结构整体稳定的情况下，而灵活地去应对各个子模块或子步骤的变化和晚期需求

结构化编程思想

方法5

方法4

方法3

方法2

方法1

编写main方法同时调用后台接口把方法按照顺序实现

编写方法2，方法4

编写方法1，方法3，方法5

前台程序

后台程序

面向对象编程思想

方法5

方法4

方法3

方法2

方法1

后台程序

实现父类方法2，方法4抽象方法，然后写main方法调用后台接口

编写方法1，方法3，方法5

编写抽象类2和抽象类4

不变的（稳定的）

把原本main方法的步骤封装起来利用多态调用抽象类2和4

前台程序

变化的（业务）

策略模式 （含有大量if-else 且会发生变化会增加的代码 就用策略模式）

注：只用于算法方面

动机：在项目构建中，某些对象使用的算法可能会多种多样，经常改变，但是如果把算法都放到对象里就会非常的臃肿，而且有时候支持不需要的算法也会是一个负担，

如何在运行时根据需要透明地更改对象的算法？将算法和对象本身解耦，避免上述问题？

模式定义：定义一系列算法，然后把他们一个一个封装起来，并且使他们可以互相替换（变化），该模式可以使得算法独立于使用他的客户程序（稳定）而变化（扩展，子类化）

大量算法编程（以前）

引入枚举 然后通过if-else来判断如果是加法怎么样，减法怎么样等

或者没有枚举类直接if判断

枚举类 加法减法乘法

缺点，如果想改就要枚举，方法都要改，如果算法多的话代码会很臃肿

大量算法编程（现在）

应用程序调用算法接口，根据多态，new出不同子类来实现算法

多个子算法同时实现算法接口并且实现父类接口方法

乘法类

减法类

加法类

算法的接口，提供一个方法用于编写子类算法

不变的（稳定的）

变化的（业务）

观察者模式（一对多） 核心是抽象的依赖关系

动机：在软件构建过程中，我们需要为某些对象建立一种“通知依赖关系”，即一个对象（目标对象）的状态发生变化，所有依赖对象（观察者对象）都将得到通知（一只猫多只老鼠），如果这样的依赖过于紧密，会使软件不能很好的抵御变化

使用面向对象技术，可以弱化这种依赖关系，并形成稳定的依赖关系，实现软件体系结构的松耦合

不变的（稳定的）

Subject

Add(IProgress) （Attach(Observer)）

Remove(IProgress) （Detach(Observer)）

Notify() （onProgress）

Observer

IProgress（相当于Observer）

--------------------------------------------

DoProgress()（相当于update()）

For all o in Observer{

o.updata()

}对代码的List数据结构进行遍历

ConcreteSubject

GetState()------------- return subjectState

SetState()

subjectState

ConcreteObserver

Update()--------observerState = subject.GetState

observerState

subject

界面

Class MainForm 继承 Form 实现IProgress {

String txtfilePath; 路径 读取到的

Int txtNumber; 个数

// 需求1

Progressbar Progressbar；

Public Void Button1\_Click(){

String filePath= txtfilePath; 获取到的

Int Number= txtNumber;

ConsoleNotifer cn

FileSplitter splitter（filePath，Number）

//添加观察者

Splitter.add（this）

Splitter.add（cn）

Splitter. split

}

Void DoProgress(float value){

Progressbar.setValue(value)

} 实现方法

}

}

Class ConsoleNotifer 实现 IProgress {

DoProgress(float value){

………..

}

}

接口 Class IProgress{

抽象类 DoProgress(float value);

}

Class FileSplitter{

String m\_filePath;

Int m\_fileNumber;

// Progressbar m\_Progressbar； 通知控件

List<IProgress> m\_iProgressList; 抽象通知机制

上 集合

构造函数(String filePath, Int fileNumber){

m\_filePath= filePath

m\_fileNumber= fileNumber

}

Add方法 参数IProgress； 添加

Remove方法 参数IProgress 删除

Void split(){

1 读文件

2 分批向小文件写入

For(int I = 0 ; i< m\_fileNumber;i++){

……

//更新进度

If(m\_iProgress!= null)

onProgress(); // 发送通知

}

}

}

Protected void onProgress(){

List<IProgress> itor=m\_iProgressList.begin() 取迭代器循环

针对每一个值来进行DoProgress

itor. DoProgress ((i+1)/ m\_fileNumber)

}

变化的（业务）

代码如上

需求1：加个进度条

总结：

1. 使用面向对象的抽象，观察者模式可以使我们独立的改变目标和观察者，
2. 目标（被观察者）发送通知时，无需制定观察者，通知会自动传播（携带通知信息作为参数）
3. 观察者自己决定是否需要订阅通知，目标对象不需要知道
4. 观察者模式是基于事件的ui框架中非常常用的设计模式，MVC重要组成部分

装饰者模式

动机：过度的通过集成来扩展对象的功能会出现问题，由于集成为类型引入的静态特质，是这样的扩展缺乏灵活性，并且随着子类增多，会导致更多子类膨胀

如何使对象功能的扩展动态的实现同时避免扩展功能的增多带来的子类的膨胀

模式定义：动态（组合）地给一个对象增加一些额外的职责。就增加功能而言，装饰者模式比生成子类（集成）更为灵活（消除重复代码&减少子类个数）

通过组合的方式来new 不同的子类（多态）可以消除重复的代码。

红色表示稳定

橙色表示变化

Component

Operation()

例：操作类 如：读文件，定位文件等等

Component.Operation()

Concrete Decorator

Operation()

AddedBehavior()

Concrete Decorator

Operation()

addedState

add

Component.Operation()

组合形式

Component

Decorator

Operation()

ConcreteComponent

Operation()

主体操作

例：加密等额外功能

如：文件流的操作，内存流的操作等等

扩展功能

装饰者模式特点：有继承有组合，继承为了接口统一规范，组合是为了扩展功能

装饰者模式主要是解决**主体类在多方面的扩展功能**

主体操作和扩展操作分开来

桥接模式（桥模式）

动机：由于某些类型的固有的实现逻辑使得它们具有两种或多个维度的变化

将抽象部分（业务功能）与实现部分（平台实现）分离，使他们可以独立的变化

Abstraction

Operation()

Imp变量指向Imp

Implementor

OperationImp()

Imp

抽象部分

实现部分

Imp. OperationImp()

业务功能需要平台的实现

所以需要一个Imp的引用

ConcreteImplementorB

OperationImp()

ConcreteImplementorA

OperationImp()

RefinedAbstraction

版本子类集成版本实现该方法然后通过指向imp把平台实现方法给版本如：精简版的PC，手机

例：RefinedAbstraction相当于版本变化 精简版，豪华版，普通版

ConcreteImplementor相当于不同平台 PC平台 手机平台

他们两个部分的抽象基类则是定义这两种需要的实现方法

分离变化就是桥模式