设计模式

可以复用的面向对象软件设计方法





## 设计模式的6大原则

复用，稳定，改动最小，

1. √依赖倒置原则（依靠抽象）

高层模块（稳定）不应该依赖于低层模块（变化），二者都应该依赖于抽象（稳定）

抽象（稳定）不应该依赖于实现细节（变化），实现细节应该依赖于抽象（稳定）

依赖于抽象而不依赖于具体。

simulatordevice中的传感器成员是sensor抽象类而不是具体类。

当底层传感器替换的时候，不影响到上层代码。

1. 开闭原则（抽象化，可拓展，不修改）

对拓展开放，对修改关闭。使用接口和抽象类在不修改元代码基础上，实现程序的拓展性号，易于维护和升级。

1. 单一职责
2. 里氏代换原则 （抽象实现规范）

对继承实现的步骤具体规范。任何基类可以出现的地方子类一定可以出现。

子类有父类所有功能的实现。而不是继承了很多东西都用不着。

比如Qsensor是个抽象类，继承它的都是传感器的具体类。

而不会让模拟器去继承它。也就是要弄清楚组合和继承。

1. 接口隔离原则（减低耦合）

单继承父类，多继承接口。降低耦合。

降低类之间耦合度。多个接口比单个接口好

接口小而完备，不应该强迫客户程序依赖他们不用的方法

1. √ 合成复用原则 （减少使用继承）

Class A里面放class B的对象或者指针。ClassA就是classB的组合。

继承在某种成单独上破化了封装性，子类父类耦合度高。

Simulatordevice里面是各种东西的组合而不是使用继承。

尽可能使用合成和聚合，减少使用继承。

合成复用原则是通过将已有的对象纳入新对象中，作为新对象的成员对象来实现的，新对象可以调用已有对象的功能，从而达到复用

1. 封装变化原则（知道变和不变的分界点）

使用封装来创建对象之间的分界层，让设计者可以在分界层的一侧进行修改，而不会对另一侧产生不良影响，从而实现层次松耦合

1. 最少知道原则（针对接口编程原则）

客户程序无需获知对象的具体类型，而是知道对象所具有的接口。

工厂模式

构造对象使用new，这种做法让调用者多知道了一种类，违背松耦合。构造过程可以封装。

简单工厂模式

生产很多水果，但是每次增加水果都要将该水果生产方式添加到工厂，违背了开闭原则。

工厂方法模式

为每个水果建立工厂，解决了复杂过程封装；

抽象工程模式

每个工厂都具有相似的接口。抽象出来。

用通用接口实现

设计思路：

如何实现

是否违背设计原则

进行改进

# 观察者模式 （行为模式-依赖倒置）

QT的信号和槽机制就是观察者订阅模式。

Button组件被按下，connect的所有类都会收到消息去处理。

所有的信号与槽机制基类都是QOBJECT

Button类中，有个vector<QOBJECT> 队列，添加相关类（就是connet添加进来的）

Button中事件发生就在button中对队列中的对象调用对应slot的对应函数。从而实现了观察者订阅者模式。（依赖倒转原则实现的，订阅者都继承了抽象类，在被观察者中被调用回调函数实现）

是一种多态的体现

# 单例模式 （创建模式）

提供一个全局访问点，使得系统中只有一个实例，避免重复创建和信息不对等

应用场景：管理资源，如日志、线程池

饿汉模式，类不调用也会有个实例占用内存。浪费资源

Class singleton

{

Private:

Static singleton\* m\_single=new singleton();

Singleton(){};

Public:

Static Singleton\* getInstance(){return m\_single;};//提供一个访问点  
}

懒汉模式 在调用的时候才会分配内存

保证一个系统只有一个实例

多进程: 双重锁；volatile防止reorder（编译器优化重排导致的内存内容未初始化）；防止拷贝构造函数；禁止赋值

**public** **class** **Singleton** {  
   **private** **static** **volatile** Singleton instance = **null**; //有的用final修饰(final就是C++里的const)  
   **private** **Singleton**() {}  
   
   **public** **static** Singleton **getInstance**() {  
   **if** (instance == **null**){  
     **synchronized**(Singleton.class){  
       **if** (instance == **null**)  
         instance = **new** Singleton();  
     }  
   }  
   **return** instance;  
   }  
}

C++版本

Class Singleton

{

Private：

Static Singleton\* handle=nullptr;

Singleton();

Public：

Static Singleton\* getInstnce()

{

If(handle==nullptr)

{

Std::mutex m\_mtx;

M\_mtx.lock();

If(instance==null)

{

Instance=new Singleton();

}

M\_mtx.unlock();

Return instance;

}  
}

**volatile**关键字的一个作用是禁止指令重排，把instance声明为volatile之后，对它的写操作就会有一个内存屏障，这样，在它的赋值完成之前，就不用会调用读操作。（写的时候不能读）

**为什么要声明类静态函数？**类加载的时候就会分配内存，类的静态成员函数不会访问或者修改任何对象（非static）数据成员。类的非静态成员（变量和方法）属于类的对象，所以只有在类的对象产生（创建类的实例）时才会分配内存，然后通过类的对象（实例）去访问。

类的创建需要调用getinsstance，所以没有在没有对象前都能调用的函数必须是类静态函数。

**如何禁止拷贝构造函数？如何禁止赋值拷贝？**

编译器将自动产生4个public inline的默认函数。形式为：

A();

A(const A&);

A& operator=(const A& a)

~A();

如果类的成员中有指针的话，浅拷贝方式的结果是两个不同对象的指针指向同一块内存区域，容易出现访问冲突，多次delete等错误，不是我们所希望的。深拷贝是调用构造函数

将拷贝构造函数和重载赋值运算符设为private来禁止拷贝

private:

 Widget(const Widget&);

 Widget& operator=(const Widget&);

## 静态内部类(使用时候加载)

加载确保线程安全带来的便利，又能延迟加载的方式，就是静态内部类。Java静态内部类的特性是，加载的时候不会加载内部**静态类**，使用的时候才会进行加载。而**使用**到的时候类加载又是线程安全

public class Singleton {

private static class SingletonHolder{

private static Singleton instance = new Singleton();

}

private Singleton(){}

public static Singleton getInstance(){

return SingletonHolder.instance;

}

}

# 工厂模式 （创建模式）

## 普通工厂模式

解决new带来的紧耦合问题

把new这个具体部分放到具体中

场景：从外部传进一个指针（抽象工厂），放着用户要用的具体工厂，

然后交给工厂进程具体的new

这样的化，每个对象都需要一个工厂。

## 抽象工厂模式

有关联的放一起，降低编程出错的可能。

比如：数据库有connect对象，cmd对象，reader对象

在创建一个数据库的时候，要创建对应的connect，cmd，reader类。因此需要把他们封装到一起

原型模式

普通工厂模式中，A类，需要Afactory工厂类去实现new的松耦合

A类可以和Afactory类合并在一起。提供自我克隆接口。那么这个类模板就叫原型。

怎么自我克隆？通过拷贝构造函数。具体类BinarySplitter类，ISplitter是抽象类

Virtual ISplitter\* clone()

{

Return new Binarysplitter(\*this);  
}

拷贝构造函数实现

This是一个已经存在的实例

然后通过拷贝构造函数进行了深拷贝



Super是父类不是个实例，所有这里实现的是浅拷贝



Person类里面有个computor实例，实例拷贝自己是深拷贝