设计模式入门

鸭子的设计：

class FlyBehavior //接口类

{

virtual void Fly(){};

};

class QuackBehavior //接口类

{

virtual void Quack();

};

class FlyWithWings:public FlyBehavior //继承于接口类的实现类

{

void Fly(){//用翅膀飞行};

};

class FlyWithWings:public FlyBehavior //继承于接口类的实现类

{

void Fly(){//不会飞行};

};

class QuackGG:public QuackBehavior //继承于接口类的实现类

{

void Quack(){//咕咕叫};

};

class QuackZZ:public QuackBehavior //继承于接口类的实现类

{

void Quack(){//吱吱叫};

};

class Duck //抽象类

{

public:

void PerformFly(){flyBehavior.Fly()}; //运行接口类的对象的成员函数

void PerformQuack(){quackBehavior.Quack()}; //运行接口类的对象的成员函数

void Swim(){//游泳};

virtual void Display(){//外观};

void SetFlyBehavior(FlyBehavior fb){flyBehavior = fb}; //改变鸭子的行为的方法

void SetQuackBehavior(QuackBehavior qb){quackBehavior = qb}; //改变鸭子的行为的方法

private:

FlyBehavior flyBehavior; //接口类对象

QuackBehavior quackBehavior; //接口类对象

};

class MallardDuck:Duck //继承与抽象类的实现类

{

public:

MallardDuck()

{

flyBehavior = new FlyWithWings(); //实现了接口类的对象

quackBehavior = new QuackBehavior(); //实现了接口类的对象

};

void Display(){cout<<”I’m a real Mallard duck!”<<endl;}; //覆盖原有的成员函数

};

设计原则：

* 找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混合在一起。
  + 分开管理
* 针对接口编程，而不是针对实现编程
  + 在类中加入接口，对接口进行操作
* 多用组合，少用继承
  + 只是继承接口，然后将接口和不同的行为实现进行组合

以上为第一种设计模式：

策略模式：定义了算法族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化独立于使用算法的客户。

解释：算法族就是很多算法组成的很多族，每一个族都能具体的实现一个功能，只是族中不同的算法能通过不同的过程实现这个行为。

的

第二种模式：

观察者模式：定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会受到通知并自动更新。

类比为：报纸订阅服务。

设计原则：

* 为了交互对象之间的松耦合设计而努力。
  + 当两个对象之间松耦合，它们依然可以交互，但是不太清楚彼此的细节。

class Subject //主题类接口

{

public:

virtual void RefisterObserver()=0; //注册观察者

virtual void RemoveObserver()=0; //删除观察者

virtual void NotifyObservers()=0; //通知观察者

};

class Observer //观察者更新数据的接口

{

public:

virtual void Updata(float temperature,float humidity,float pressure)=0;

};

class DisplayElement //观察者显示的接口

{

public:

virtual void Display()=0;

};

class WeatherData:public Subject //一个主题的实现类

{

public:

WeatherData(){};

void RefisterObserver(Observer o){observers.push\_back(o);}

void RemoveObserver(Observer o){observers.erase(o);}

void NotifyObservers(){给每一个观察者发布新数据：Observer.Updata(temperature,humidity,pressure);} //将新的数据通知给每一个观察者

void MeasurementsChanged(){NotifyObservers();} //调用通知函数

void SetMeasurements(float temperature,float humidity,float pressure){this.temperature = temperature;…MeasurementsChanged();} //得到新的数据，并进行通知

private:

List<Observer> observers;

float temperature;

float humidity;

float pressure;

};

class CurrentConditionsDisplay:public Observer, DisplayElement //一个观察者的实现类

{

public:

CurrentConditionsDisplay(Subject weatherData){this.weatherData = weatherData; weatherData.registerObserver(this);} //给这个观察者在主题中进行注册

void Updata(float temperature,float humidity,float pressure){//更新数据，并：Display();};

void Display(){//显示布告板信息};

private:

float temperature;

float humidity;

Subject weatherData;

};

第三种模式：

装饰者模式:动态地将责任附加到对象上。若要扩展功能，装饰着提供了比继承更有弹性的替代方案。

设计原则：类应该对扩展开放，对修改关闭。