装饰者模式：Buff加成类道具模块

Buff加成类道具有护盾、加速鞋两种。在吃掉Buff加成类道具后，该玩家会获得相应的Buff加成效果。

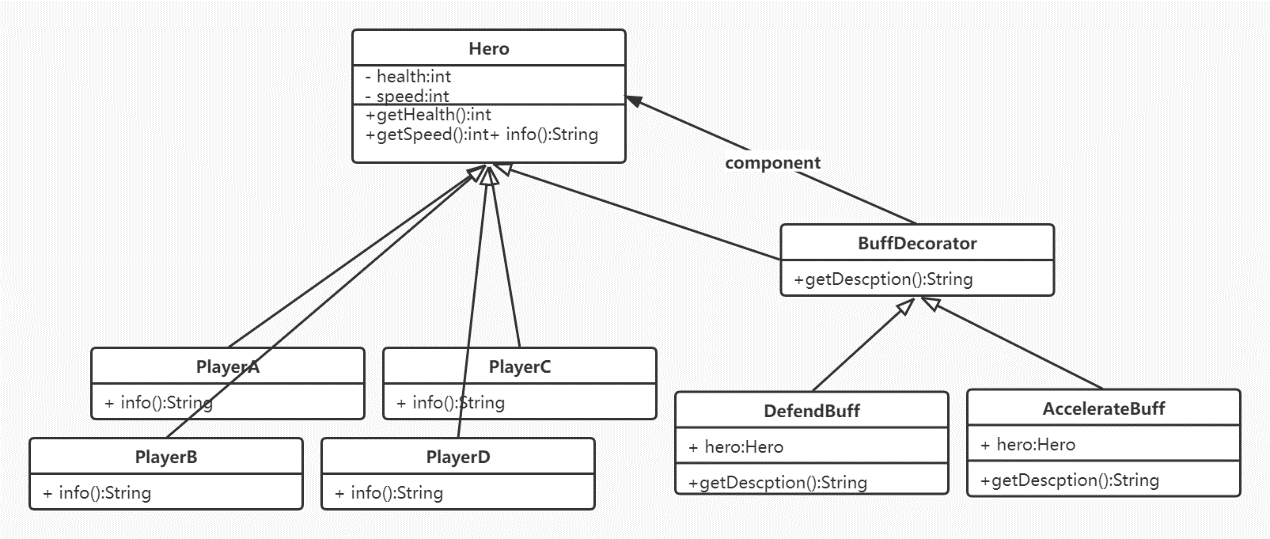


护盾道具如上图所示。在玩家吃掉道具后，会获得生命值加一点的加成。



加速鞋道具如上图所示。在玩家吃掉道具后，会获得移动速度加一点的加成。

从游戏玩法上，这两种加成道具是可以动态的、可叠加的附加到玩家身上，因此可以使用装饰者模式，将玩家作为被装饰对象，加成道具作为装饰者，二者拥有相同的超类型，在玩家获取到加成道具时可以用装饰过的玩家替代当前玩家，达到获取加成的目的。



该模块使用装饰者模式的类图如上图所示。虚基类英雄Hero类作为被装饰对象的超类，拥有health生命值和speed移动速度等属性；PlayerA等子类是动态的可叠加的加上Buff的装饰对象。

public abstract class Hero **{**

//生命值

private int health=3**;**

//移动速度

private int speed=1**;**

public int getHealth **()** **{** return **this.**health**;** **}**

public int getSpeed **()** **{** return **this.**speed**;** **}**

public abstract String info **(){}**

**}**

public class PlayerA **extends** Hero {

public String info **(){**

return “I am player A!”;

**}**

}

装饰者BuffDecorator类也继承自Hero类，从而用装饰过的玩家替代当前玩家：

public abstract class BuffDecorator **extends** Hero **{**

public abstract String getDescption();

**}**

护盾装饰者DefendBuff类是一个具体装饰者，因此拓展自BuffDecorator类，同时它拥有一个被装饰者Hero类的引用，以便对其进行装饰并替代被装饰者。

public class DefendBuff **extends** BuffDecorator **{**

//被装饰者

Hero hero;

public DefendBuff **(**Hero hero**)** **{**

**this.**hero **=** hero**;**

**}**

Public String getDescption(){

return hero.info()+”buff:护盾+1”;

}

//进行buff加成

Public int getHealth(){

return hero.getHealth()+1;

}

**}**

加速鞋装饰者AccelerateBuff类也是一个具体装饰者，因此也拓展自BuffDecorator类，同时它也拥有一个被装饰者Hero类的引用，以便对其进行装饰并替代被装饰者。

public class AccelerateBuff **extends** BuffDecorator **{**

//被装饰者

Hero hero;

public AccelerateBuff **(**Hero hero**)** **{**

**this.**hero **=** hero**;**

**}**

Public String getDescption(){

return hero.info()+”buff:移动速度+1”;

}

//进行buff加成

Public int getSpeed(){

return hero.getSpeed()+1;

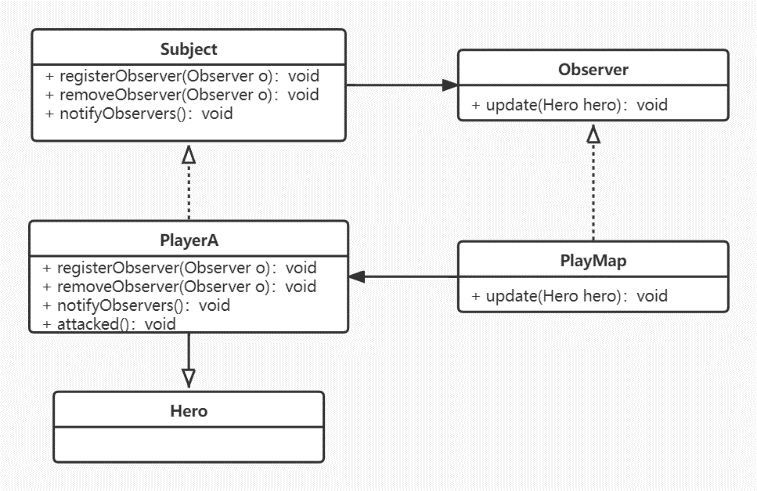
}

**}**

这样就完成了采用装饰者模式对Buff加成道具进行了实现，两种加成道具可以随时动态的、可叠加的加成到玩家身上。后续如果还有更多的加成道具，都可以继承自BuffDecorator装饰者基类，非常灵活且易于扩展。

观察者模式

在游戏中当玩家生命值降到0时即会死亡，这时玩家的在地图上的所有状态包括身影都将移除出地图，因此当玩家改变状态为死亡时，地图都会收到通知并将该玩家移除出当前场景。为了让玩家和地图之间进行交互但同时维持松耦合，可以采用观察者模式，玩家作为被观察的对象，当玩家死亡后通知作为观察者的地图，地图再做出相应的去除玩家操作，观察者模式的类图如下所示：



首先需要创建被观察者接口Subject和观察者接口Observer，其中Subject接口中含有观察者的注册和移除功能，还有变更发生后通知观察者的功能。而Observer接口中主要含有的是接收到变更后系统应该作出相应的更新的操作。

public interfaceSubject **{**

Public void registerObserver(Observer o);

Public void removeObserver(Observer o);

Public void notifyObservers();

**}**

public interfaceObserver **{**

Public void update(Hero hero);

**}**

因为玩家是被观察的对象，因此玩家PlayerA类要实现Subject接口，并且在受到攻击死亡时会调用notifyObservers()通知观察者。

public class PlayerA extends Hero **implements** Subject {

private ArrayList observers;

public PlayerA(){

observers = new ArrayList();

}

//实现Subject接口

Public void registerObserver(Observer o){

observers.add(o);

}

Public void removeObserver(Observer o){

int i = observers.indexOf(o);

if(i>=0) observers.remove(i);

}

Public void notifyObservers(){

For(int i=0;i<observers.size();i++){

Observer observer=(Observer)observer.get(i);

Observer.update(this);

}

}

//当玩家受到攻击后生命值等于0时死亡，将通知观察者

public void attacked()**{**

this.health-=1;

if(this.health<=0)

notifyObservers();

}

}

因为地图是需要在接收到玩家死亡的通知后进行更新的，因此它需要实现Observer接口以从被观察者对象中获得改变并作出相应的更新。并且地图需要被观察者的引用作为注册观察者之用。

public class PlayMap **implements** Observer{

private Subject PlayerA;

public PlayMap(Subject playerA){

this.PlayerA=playerA;

playerA.registerObserver(this);

}

Public void update(Hero hero){

changeMap(hero);

}

//做出相应的更新

Public void changeMap(Hero hero){

//将该玩家移除出当前地图

deleteHeroFromMap(hero);

}

}

这样就完成了采用观察者模式对玩家死亡后地图及时更新的实现，具体而言，初始时玩家会注册地图成为自己的观察者，之后当玩家受到攻击并死亡时会通知其观察者即地图，，地图随之做出更新将该玩家移除出当前地图。这种设计使得被观察者（玩家）不需要知道具体的观察者（地图）是谁、做了什么或其他任何细节，两者之间是松耦合的，并且任何时候都可以增加新的观察者或者移除观察者，而被观察者不会受到任何影响，从而建立能应对变化的OO系统。