**（1）观察者模式**

**特点：**

①主题接口和发布者接口互相独立，同时主题接口一般需要组合订阅者在对象内部。

②一对多的关系 ，表示一个被观察者对象对应多个观察者对象的关系。

③观察者模式是一种 行为型模式 ，因为他涉及到观察的行为和发布的行为，发布的行为是抽象的，而且订阅的行为也是抽象的。

**举例：**

#include <bits/stdc++.h>

//观察者模式

class Observer;

//抽象被观察者

class Subject {

public:

Subject() : m\_nState(0) {}

virtual ~Subject() = default;

virtual void Attach(const std::shared\_ptr<Observer> pObserver) = 0;

virtual void Detach(const std::shared\_ptr<Observer> pObserver) = 0;

virtual void Notify() = 0;

virtual int GetState() { return m\_nState; }

void SetState(int state) {

std::cout << "Subject updated !" << std::endl;

m\_nState = state;

}

protected:

std::list<std::shared\_ptr<Observer>> m\_pObserver\_list;

int m\_nState;

};

//抽象观察者

class Observer {

public:

virtual ~Observer() = default;

Observer(const std::shared\_ptr<Subject> pSubject, const std::string &name = "unknown")

: m\_pSubject(pSubject), m\_strName(name) {}

virtual void Update() = 0;

virtual const std::string &name() { return m\_strName; }

protected:

std::shared\_ptr<Subject> m\_pSubject;

std::string m\_strName;

};

//具体被观察者

class ConcreteSubject : public Subject {

public:

void Attach(const std::shared\_ptr<Observer> pObserver) override {

auto iter = std::find(m\_pObserver\_list.begin(), m\_pObserver\_list.end(), pObserver);

if (iter == m\_pObserver\_list.end()) {

std::cout << "Attach observer" << pObserver->name() << std::endl;

m\_pObserver\_list.emplace\_back(pObserver);

}

}

void Detach(const std::shared\_ptr<Observer> pObserver) override {

std::cout << "Detach observer" << pObserver->name() << std::endl;

m\_pObserver\_list.remove(pObserver);

}

//循环通知所有观察者

void Notify() override {

auto it = m\_pObserver\_list.begin();

while (it != m\_pObserver\_list.end()) {

(\*it++)->Update();

}

}

};

//具体观察者1

class Observer1 : public Observer {

public:

Observer1(const std::shared\_ptr<Subject> pSubject, const std::string &name = "unknown")

: Observer(pSubject, name) {}

void Update() override {

std::cout << "Observer1\_" << m\_strName << " get the update.New state is: "

<< m\_pSubject->GetState() << std::endl;

}

};

//具体观察者2

class Observer2 : public Observer {

public:

Observer2(const std::shared\_ptr<Subject> pSubject, const std::string &name = "unknown")

: Observer(pSubject, name) {}

void Update() override {

std::cout << "Observer2\_" << m\_strName << " get the update.New state is: "

<< m\_pSubject->GetState() << std::endl;

}

};

int main() {

std::shared\_ptr<Subject> pSubject = std::make\_shared<ConcreteSubject>();// 创建被观察者

// 创建观察者

std::shared\_ptr<Observer> pObserver1\_1 = std::make\_shared<Observer1>(pSubject, "1");

std::shared\_ptr<Observer> pObserver1\_2 = std::make\_shared<Observer1>(pSubject, "2");

std::shared\_ptr<Observer> pObserver1\_3 = std::make\_shared<Observer1>(pSubject, "3");

std::shared\_ptr<Observer> pObserver2\_4 = std::make\_shared<Observer2>(pSubject, "4");

std::shared\_ptr<Observer> pObserver2\_5 = std::make\_shared<Observer2>(pSubject, "5");

std::shared\_ptr<Observer> pObserver2\_6 = std::make\_shared<Observer2>(pSubject, "6");

// 注册观察者

pSubject->Attach(pObserver1\_1);

pSubject->Attach(pObserver1\_2);

pSubject->Attach(pObserver1\_3);

pSubject->Attach(pObserver2\_4);

pSubject->Attach(pObserver2\_5);

pSubject->Attach(pObserver2\_6);

pSubject->SetState(2);// 改变状态

pSubject->Notify();

std::cout << std::string(50, '-') << std::endl;

// 注销观察者

pSubject->Detach(pObserver1\_1);

pSubject->Detach(pObserver1\_2);

pSubject->SetState(3);

pSubject->Notify();

return 0;

//运行结果如下：

//Attach observer1

//Attach observer2

//Attach observer3

//Attach observer4

//Attach observer5

//Attach observer6

//Subject updated !

//Observer1\_1 get the update.New state is: 2

//Observer1\_2 get the update.New state is: 2

//Observer1\_3 get the update.New state is: 2

//Observer2\_4 get the update.New state is: 2

//Observer2\_5 get the update.New state is: 2

//Observer2\_6 get the update.New state is: 2

//--------------------------------------------------

//Detach observer1

//Detach observer2

//Subject updated !

//Observer1\_3 get the update.New state is: 3

//Observer2\_4 get the update.New state is: 3

//Observer2\_5 get the update.New state is: 3

//Observer2\_6 get the update.New state is: 3

}

**（2）策略模式**

**特点：**

①多重条件语句不易维护，而使用策略模式可以避免使用多重条件语句。

②策略模式提供了一系列的可供重用的算法族，恰当使用继承可以把算法族的公共代码转移到父类里面，从而避免重复的代码。

③策略模式可以提供相同行为的不同实现，客户可以根据不同时间或空间要求选择不同的。

④策略模式提供了对开闭原则的完美支持，可以在不修改原代码的情况下，灵活增加新算法。

⑤策略模式把算法的使用放到环境类中，而算法的实现移到具体策略类中，实现了二者的分离。

**举例：**

//发奖策略

//AwardPrizeStrategy接口，其中有发奖方法，需要实现类实现。

package com.ismartgo.uqcode.strategy.awardPrize;

import com.ismartgo.uqcode.model.UqcUserJoinRecord;

public interface AwardPrizeStrategy {

boolean award(String takePrizeParams , UqcUserJoinRecord record);

}

//免费资源发奖实现类

package com.ismartgo.uqcode.strategy.awardPrize.impl;

import java.util.HashMap;

import java.util.Map;

import com.ismartgo.uqcode.common.utils.DateUtil;

import com.ismartgo.uqcode.common.utils.HttpClientUtil;

import com.ismartgo.uqcode.model.UqcUserJoinRecord;

import com.ismartgo.uqcode.strategy.awardPrize.AwardPrizeStrategy;

public class AwardFreeResourcePrizeStrategy implements AwardPrizeStrategy {

@Override

public boolean award(String takePrizeParams, UqcUserJoinRecord record) {

//发免费资源逻辑

return true;

}

}

//红包发奖实现类

package com.ismartgo.uqcode.strategy.awardPrize.impl;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Date;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import org.apache.commons.lang.StringUtils;

import com.ismartgo.uqcode.common.constants.QRSVConstants;

import com.ismartgo.uqcode.common.utils.DateUtil;

import com.ismartgo.uqcode.common.utils.HttpClientUtil;

import com.ismartgo.uqcode.common.utils.MD5Util;

import com.ismartgo.uqcode.common.utils.Wx3HttpUtils;

import com.ismartgo.uqcode.model.UqcUserJoinRecord;

import com.ismartgo.uqcode.strategy.awardPrize.AwardPrizeStrategy;

import net.sf.json.JSONObject;

public class AwardMoneyPrizeStrategy implements AwardPrizeStrategy{

@Override

public boolean award(String takePrizeParams, UqcUserJoinRecord record) {

Map<String, Object> paramsMap = new HashMap<String, Object>();

//发奖参数屏蔽

JSONObject result = null;

try {

result = Wx3HttpUtils.post(URL,paramsMap);

} catch (Exception e) {

e.printStackTrace();

}

if(result!=null&&result.getInt("error")==0) {

return true;

}

return false;

}

}

//Context类

package com.ismartgo.uqcode.strategy.awardPrize.context;

import com.ismartgo.uqcode.model.UqcUserJoinRecord;

import com.ismartgo.uqcode.strategy.awardPrize.AwardPrizeStrategy;

public class AwardPrizeContext {

private AwardPrizeStrategy awardPrizeStrategy;

public AwardPrizeContext(AwardPrizeStrategy awardPrizeStrategy) {

super();

this.awardPrizeStrategy = awardPrizeStrategy;

}

public boolean award(String prizeParams,UqcUserJoinRecord record) {

return awardPrizeStrategy.award(prizeParams, record);

}

}

**（3）单例模式**

**特点：**

①单例类只能有一个实例

②单例类必须自己创建自己的唯一实例

③单例类必须给所有其他对象提供这一实例的访问

**举例：**

//一个人只能同时和一个人结婚，已婚状态下则不能再次登记结婚

//MarriageException.java

public class MarriageException extends Exception{

private static final long serialVersionUID = 1L;

public MarriageException(String message) {

super(message);

}

}

//MarriageSingleton.java

public class MarriageSingleton {

private static MarriageSingleton instance=null;

private MarriageSingleton(){}

public static MarriageSingleton getInstance() throws MarriageException{

if(instance==null)

{

System.out.println("为此人创建婚姻！");

instance = new MarriageSingleton();

}

else

{

throw new MarriageException("此人已经拥有了婚姻，无法再次创建婚姻！");

}

return instance;

}

public void marriage(){

System.out.println("正在享受幸福美满的婚姻中！");

}

}

//Client.java

public class Client {

public static void main(String[] args) {

MarriageSingleton m1,m2;

try {

m1 = MarriageSingleton.getInstance();

m1.marriage();

} catch (MarriageException e) {

System.out.println(e.getMessage());

}

System.out.println("-----------------------------");

try{

m2 = MarriageSingleton.getInstance();

m2.marriage();

} catch(MarriageException e){

System.out.println(e.getMessage());

}

}

}

**（4）备忘录模式**

**特点：**

①提供了一种可以恢复状态的机制。当用户需要时能够比较方便地将数据恢复到某个历史的状态。

②实现了内部状态的封装。除了创建它的发起人之外，其他对象都不能够访问这些状态信息。

③简化了发起人类。发起人不需要管理和保存其内部状态的各个备份，所有状态信息都保存在备忘录中，并由管理者进行管理，这符合单一职责原则。

**举例：**

//Editor

public class Editor {

private String title;

private String content;

private String imgs;

public Editor(String title, String content, String imgs) {

this.title = title;

this.content = content;

this.imgs = imgs;

}

public String getTitle() {

return title;

}

public String getContent() {

return content;

}

public String getImgs() {

return imgs;

}

public void setTitle(String title) {

this.title = title;

}

public void setContent(String content) {

this.content = content;

}

public void setImgs(String imgs) {

this.imgs = imgs;

}

public ArticleMemento saveToMemento() {

ArticleMemento articleMemento = new ArticleMemento(this.title, this.content, this.imgs);

return articleMemento;

}

public void undoFromMemento(ArticleMemento articleMemento) {

this.title = articleMemento.getTitle();

this.content = articleMemento.getContent();

this.imgs = articleMemento.getImgs();

}

@Override

public String toString() {

return "Editor{" +

"title='" + title + '\'' +

", content='" + content + '\'' +

", imgs='" + imgs + '\'' +

'}';

}

}

//ArticleMemento

public class ArticleMemento {

private String title;

private String content;

private String imgs;

public ArticleMemento(String title, String content, String imgs) {

this.title = title;

this.content = content;

this.imgs = imgs;

}

public String getTitle() {

return title;

}

public String getContent() {

return content;

}

public String getImgs() {

return imgs;

}

@Override

public String toString() {

return "ArticleMemento{" +

"title='" + title + '\'' +

", content='" + content + '\'' +

", imgs='" + imgs + '\'' +

'}';

}

}

//DraftsBox

public class DraftsBox {

private final Stack<ArticleMemento> STACK = new Stack<ArticleMemento>();

public ArticleMemento getMemento(){

ArticleMemento articleMemento = STACK.pop();

return articleMemento;

}

public void addMemento(ArticleMemento articleMemento){

STACK.push(articleMemento);

}

}

//测试类

public class Main {

public static void main(String[] args) {

DraftsBox draftsBox = new DraftsBox();

Editor editor = new Editor("标题", "内容","图片");

ArticleMemento articleMemento = editor.saveToMemento();

draftsBox.addMemento(articleMemento);

System.out.println("标题：" + editor.getTitle() + "\n" +

"内容：" + editor.getContent() + "\n" +

"插图：" + editor.getImgs() + "\n暂存成功");

System.out.println("完整的信息" + editor);

System.out.println("==========首次修改文章===========");

editor.setTitle("首次修改标题");

editor.setContent("首次修改内容");

System.out.println("==========首次修改文章完成===========");

System.out.println("完整的信息" + editor);

articleMemento = editor.saveToMemento();

draftsBox.addMemento(articleMemento);

System.out.println("==========保存到草稿箱===========");

System.out.println("==========第2次修改文章===========");

editor.setTitle("二次修改标题");

editor.setContent("二次修改内容");

System.out.println("完整的信息" + editor);

System.out.println("==========第2次修改文章完成===========");

System.out.println("==========第1次撤销===========");

articleMemento = draftsBox.getMemento();

editor.undoFromMemento(articleMemento);

System.out.println("完整的信息" + editor);

System.out.println("==========第1次撤销完成===========");

System.out.println("==========第2次撤销===========");

articleMemento = draftsBox.getMemento();

editor.undoFromMemento(articleMemento);

System.out.println("完整的信息" + editor);

System.out.println("==========第2次撤销完成===========");

}

}