**CSCI 598 - SW DESIGN PATTERNS**

**HW3 solutions**

**Question 1. (14 pts)**

We are designing a game where players navigate around a map and collect orbs and gemstones. These collectibles are represented by a Collectible interface, a class with only virtual methods. Each type of orb and gemstone has different effects and can be used for trading within the game. There are vast quantities of orbs and gemstones spread all over the game map.

Each Collectible item is represented by a separate object either as an instance of Orb class or Gem class.

The Orb class has the following attributes: position (2D), size (influences its duration and power of the effect, 1-5), rarity (common, rare, scarce), effect (healing, increaseSize), bearer (null or player), texture (an image). The effect acts on the bearer if there is one.

The Gem class has the following attributes: position (2D), name (ruby, sapphire, emerald), size (1-5), rarity (common, rare, scarce), effect ( strength, health), bearer (null or player), and texture (an image). The effect acts on the bearer if there is one.

Collectible items can *draw* themselves, *activate* their effect, *move*, *calculate value* using rarity, size, and name of the gem or effect of the orb. It also has a setBearer(Player) method.

Everything looks good but we have run into a problem: the game is too resource-intensive and runs out of memory due to the vast number of Collectible objects.

Your task is to propose a design solution that reduces the game’s memory usage while maintaining the game’s visual appeal and functionality.

*You are expected to address items 1-5*

ANSWER

1. Flyweight pattern

2. Reason. The problem is the vast number of collectible objects. This number could be reduced with flyweight pattern when position and bearer are taken out of the orb and gem classes. Then we will have at most 5x3x2 orbs per texture image; and 3x5x3x2 gems per texture image. They will created when requested. If the number of different texture is high, we could take it out as well.

3. UML

A diagram of a computer

AI-generated content may be incorrect.

4. Java Code

public interface **Collectible** {

public void draw();

public void activate();

public void move(Position nextPosition);

public int calculateValue();

public void setBearer(Player player);

}

/\*Context\*/

public class **Orb** implements Collectible{ //acting as CONTEXT

public void draw( ) {flyweight.draw(position);} //these delegations are important

public void activate(){

if(bearer!=null) flyweight.activate(bearer);

}

public void move(Position2D nextPosition){ position=nextPosition;}

public int calculateValue() { return flyweight.value();}

public void setBearer(Player player){ bearer=player;}

public Orb(int size, Rarity rarity, String effect, Image texture, Position2D position){

flyweight= CollectibleFactory.getOrb( size, rarity,effect,texture);

this.position=position;

//some might like this implementation.

//it is hiding the factory from the client code.

//Client thinks different instances with all attributes are created

}

/\* alternatively, the client may call the Factory and

passes the flyweight to the constructor the Orb.

Then client is aware of the factory.

Some may prefer this as it informs the client about the sharing.

\*/

//extrinsic states --important

private Position2D position;

private Player bearer=null;

//flyweight

private OrbFlyweight flyweight;

}

/\*Concrete Flyweight\*/

public class **OrbFlyweight**{ //attributes are important, the methods are not

private int size; //1-5

private Rarity rarity;

private String effect; // healing, increaseSize

private Image texture;

public OrbFlyweight(int size, Rarity rarity, String effect, Image texture){

this.size=size; this.rarity=rarity; this.effect=effect; this.texture=texture;

}

public void draw(Position2D position) { //important. Position must be passed as arg.

System.out.println(“drawing ”+toString()+ “at position” + position.getX()+ “,”+position.getY());

}

public void activate(Player bearer){ //important. Bearer must be passed as arg.

//this is not a pattern related code, so I made the simplest implementation

if(effect.equals(“healing”)) {

System.out.println(“healing Player ”+ bearer);

//bearer.health=100; //alternative

} else if(effect.equals(“increaseSize”)) {

System.out.println(“increasing the size of the bearer ”+bearer);

}

//alternative: make Effect a class and then effect.activate(Player);

}

public int value() {

//any calculation is ok. This is an arbitrary value calculation

int result=size;

if(effect.equals(“healing”)) result\*2; //having effect as class would be so much better

if(effect.equals(“increaseSize”)) result\*3; //but that is not our focus now.

//this is not a good implementation.

// It is not related to the flyweight patten, so nothing good is expected.

switch(rarity){

case COMMON: return result+1;

case RARE: return result+2;

case SCARCE: return result+3;

}

return result;

}

}

public enum Rarity { COMMON, RARE, SCARCE}

public enum GemName {Ruby, Saphire, Emerald}

public class Position2D{private int x; private int y;

public Position2D (int x, int y){this.x=x; this.y=y;}

public int getX(){return x;}

public int getY(){return y;}

}

/\* do a similar design for the Gem.\*/

class Gem implements Collectible {// act as Context

public Gem(GemName name, int size, Rarity rarity, String effect, Image texture, Position2D position){

flyweight= CollectibleFactory.getGem(name, size, rarity,effect,texture);

this.position=position;

}

public void draw(){ flyweight.draw(position);}

public void activate(){flyweight.activate(bearer);}

public void move(Position nextPosition){position=nextPosition;}

public int calculateValue(){return flyweight.value();}

public void setBearer(Player player){bearer=player;}

private Position2D position;

private Player bearer;

private GemFlyweight flyweight;

}

class GemFlyweight {

private GemName name;

private int size; //1-5

private Rarity rarity;

private String effect;

private Image texture;

public GemFlyweight(GemName name, int size, Rarity rarity, String effect, Image texture){

this,name=name; this.size=size; this.rarity=rarity;

this.effect=effect; this.texture=texture;

}

public void draw(Position2D position) {….}

public void activate(Bearer bearer){….}

public void value(){ /\*do some calculations\*/ return 110;}

}

/\*Flyweight Factory\*/ //this code is important

public class **CollectibleFactory**{

public static OrbFlyweight getOrb( int size, Rarity rarity, String effect, Image texture){

**//find the orb in the pool and return it. If not found create it, add to pool, then return**

for(OrbFlyweight f:orbpool){

if(f.size()==size && f.rarity()==rarity &&

f.effect().equals(effect) && f.texture().equals(texture) )

return f;

}

**OrbFlyweight orb=new OrbFlyweight(size, rarity, effect, texture);**

**orbpool.add(orb);**

**return orb;**

}

public static GemFlyweight getGem( GemName name,int size, Rarity rarity, String effect, Image texture){

**//find the gem in the pool and return it. If not found create it, add to pool, then return**

for(GemFlyweight f:gempool){

if(f.name()==name && f.size()==size && f.rarity()==rarity &&

f.effect().equals(effect) && f.texture().equals(texture) )

return f;

}

GemFlyweight gem=new GemFlyweight(name,size, rarity, effect, texture);

orbpool.add(gem);

return gem;

}

//private Set<Collectible> pool=new HashSet<Collectible>(); //any collection is ok

//if I use HashSet, I should implement hashCode() and equals() method of Orb and Gem

private List<OrbFlyweight> orbpool=new ArrayList<OrbFlyweight>();

private List<GemFlyweight> orbpool=new ArrayList<GemFlyweight>();

}

Client code

public class Game(){

public init(){

//client code allocates flyweights

Image texture=new Image(“orb.jpg”);

for(int i=0;i<100000;i++){

//spawn Collectible items at random locations

Position2D position=new Position2D (Math.random(100), Math.random(100));

Orb orb=new Orb (3, COMMON, “healing”, texture, position);

//Client thinks 100K orbs with all features are created

terrain.add(orb);

}

//similarly create orbs and gems with different sizes, effects and textures.

//ALTERNATIVE. Use factory here. Then set the Orb context with the flyweight

for(Object o: terrain){

if(o instanceof Collectible) ((Collectible) o).draw(); //client code use

}

}

private List<Object> terrain; //simulating terrain

}

class Player(){

private List<Collectible> items;

public void someMethod(){

items.get(0).activate(); //client code use of orb

}

}

//alternative Flyweight implementation.

// Not as much as good as the above.

// I did not use Collectible but have another FCollectible. It is possible to integrate it

//this code is much complex than the previous

/\*Context\*/

public class GemContext{

private List<Position2D> positions=new ArrayList<Position2D>();

private List<Player> bearers=new ArrayList<Bearer>();

public void addGem(Position2D p, Bearer b){

positions.add(p); bearers.add(b); //parallel lists –not a good idea

}

public void changePosition(Position2D position, int index){

positions.at(index)=position;

}

public void setBearer (Player bearer, int index){

bearers.at(index)=bearer;

}

public List<Position2D> getPositions(){ return positions;}

}

/\*Flyweight interface\*/

public interface FCollectible{

public void draw(Position2D position);

public void activate(Player bearer);

public int value();

}

/\*Concrete Flyweight\*/

public class Gem implements FCollectible{ //acting as Flyweight

private GemName name; //(ruby, sapphire, emerald),

private int size; //1-5

private Rarity rarity;

private String effect; // health, strength

private Image texture;

public Gem(GemName name, int size, Rarity rarity, String effect, Image texture){

this.name=name;

this.size=size; this.rarity=rarity; this.effect=effect; this.texture=texture;

}

public void draw(Position2D position) {

System.out.println(“drawing ”+toString()+ “at position” + position.getX()+ “,”+position.getY());

}

public void activate(Player bearer){

if(effect.equals(“health”)) {

System.out.println(“adding 5 to health of Player ”+ bearer);

//bearer.inchealth(5); //alternative

} else if(effect.equals(“strength”)) {

System.out.println(“increasing the strength of the bearer ”+bearer);

}

}

public int value() {

//the calculation is arbitrary. Any calculation is ok.

int gemvalue=size;

switch(rarity){

case COMMON: gemvalue+1; break;

case RARE: gemvalue+2; break;

case SCARCE: gemvalue+3; break;

}

switch(name){

case Ruby: gemvalue\*2; break;

case Saphire:gemvalue\*3; break;

case Emerald:gemvalue\*4;break;

}

return gemvalue;

}

}

/\* Factory is same as before, but the name of the class is different\*/

Client code

public class Game(){

public init(){

Image texture=new Image(“gem.jpg”);

Gem gem1=new CollectibleFactory.getOrb (Ruby, 3, COMMON, “strength”, texture);

GemContext context1=new GemContext();

for(int i=0;i<100000;i++){

Position2D position=new Position2D (Math.random(100), Math.random(100));

context1.addGem(position, null);

}

terrain.put(gem1, context1);

Gem2=new CollectibleFactory.getOrb (Saphire, COMMON, “strength”, texture);

GemContext context2=new GemContext();

for(int i=0;i<100000;i++){

Position2D position=new Position2D (Math.random(100), Math.random(100));

context2.addGem(position, null);

}

terrain.put(gem2, context2);

//similarly create orbs and gems with different sizes, effects and textures.

}

private Map<FCollectible,GemContext> terrain=new

HashMap<FCollectible,GemContext>();

public void display(){ //client code use of the flyweight

terrain.forEach((item, context) -> {

List<Position2D> positions=context.getPositions();

for (Position2D pos: positions) item.draw(pos);

});

}

// there is no gem.move(position) but GemContext.changePosition(index,position)

5. Evaluate

**SRP**: Flyweights are responsible for shared state: size, effect etc. Context object is responsible for position and bearer. The factory is responsible for managing the flyweights. (OPTIONAL)

**OCP**: Allows adding new collectible flyweights without modifying existing code. This enables extending the system with new shared states as needed.

**LSP**: Flyweights can replace their base types without altering program correctness. flyweight objects can be used interchangeably with their concrete counterparts.

**ISP**: (not expected) Uses focused interfaces, ensuring clients depend only on what they use.

**DIP**: Relies on abstractions, promoting decoupling and flexibility. depending on abstractions (interfaces) rather than concrete implementations, promoting flexibility and decoupling.

**Question 2 (14 pts)**

We are developing a software component for printing different types of library items on different platforms. The library has books, movies, audio tapes, maps, and newspapers. In the first iteration, I want to print books both on the web and on paper (platforms). Printing a book on web requires different formatting than printing a book on paper. In the next iteration, I also need to print movie information (its metadata) in a slightly different format on both the web and on paper. Then, I need to print similar information for other item types in the library, such as audio tapes.

Currently, these requirements have resulted in an explosion of media and formatting combinations: BookHTMLFormatter, TapeHtmlFormatter, BookPaperFormatter, and so on. Any new combined derivative (for example, MovieHtmlFormatter) will necessarily need to introduce redundancies with the existing implementation. Suggest a **structural** design pattern to avoid this explosion of media and formatting combinations.

*You are expected to address items 1-5 on the first page in your answer.*

ANSWER

1. Bridge pattern –Strategy is an alternative, but it is not a structural pattern

2. Reason. There are four media types and two ways of printing them. The explosion in the number of classes (5x2) happens because the Abstraction (media) and Implementation (printing) are coupled. The Bridge pattern will separate them. Media type abstraction forwarding the print request to Formatter implementation will result in 5+2 classes and 2 interfaces.

3. UML

A diagram of a computer

AI-generated content may be incorrect.

4.Java Code

/\*Abstraction interface\*/

public abstract class Media{ //this could also be a Java interface

**protected** Formatter formatter; // cannot have like this in interface.

//formatter **should NOT be public**, it defies the purpose of hiding underlying impl.

/\*with this constructor, subclasses cannot have default constructor.

Its purpose is to make sure that there is always a formatter\*/

public Media(Formatter formatter){

this.formatter=formatter; //good to have null checks

}

public void setFormatter(Formatter implementor) {

formatter = implementor;

} //only setter method. No getter for Formatter. It defies the purpose of hiding impl.

public **abstract** void print(); //operation to be delegated to implementor

/\*other methods are not the focus of the pattern\*/

}

/\*implementor interface\*/

public interface Formatter{

/\* could be as simple as public void String format(String text); \*/

public void String format (Map<String,String> metadata, String text);

/\* could be a series of methods, so that

\* the Media will call them in its print method in a specific order.

The following is just an example for how the primitive methods look like

\*/

public void setTitle(String title);

public void addPageBreak();

public void addText(String text);

public void addTable(String text);

}

/\* Concrete Abstraction classes\*/

public class Book extends Media{

private Map<String, String> metadata=new HashMap<String,String>(); //OPTIONAL

private String text; /\* this attribute could be Document document.\*/

/\*I don’t want to implement a whole Document class and then provide a toString() method. If you choose to implement a Document class, make it a Composite with Chapters having Sections or Text as children.\*/

public Book(Formatter formatter, String text){

//good to have null checks to make sure there is a formatter

if(formatter!=null) then super(formatter)

else { /\*do something, your choice. ask the customer\*/}

this.text=text; //then, fill in the metadata like author etc.

}

public void print(){

formatter.format(metadata, text);

/\* if the formatter interface has fine grain formatting methods

\* then, this print() method would call them in a specific order

\* to print the book contents and information.

\*/

}

/\*other book related methods,\*/

}

/\*another concrete or refined abstraction\*/

public class Movie extends Media{

public Movie(Formatter formatter, String title, String director, int year, String plot){

super(formatter); //good to have null checks to make sure there is a formatter

this.title=title; this.director=director; this.year=year; this.plot=plot;

}

private String title, director, plot; private int year;

private Object video;

public void print(){

Map<String, String> metadata=new HashMap<String, String>();

metadata.add(“title”, title);

metadata.add(“director”,director);

//if formatter has many methods, then I would call them instead of creating this map

String formattedText= formatter.format(metadata, text);

}

/\*other movie related methods\*/

}

/\*similar implementations for audio tapes, maps, and newspapers \*/

/\*Concrete Implementation classes \*/

public class PaperFormatter implements Formatter{

public void String format (Map<String,String> metadata, String text){

System.out.println(“formatting for a paper print”);

return text;

}

/\* could be as simple as

\* public void String format(String text){

\* System.out.println(“formatting for a paper print”);

\* return text; }

\*/

}

public class HTMLFormatter implements Formatter{

//a very simplified html format. this is not the main issue to realize the pattern.

//any implementation is ok. see the PaperFormatter

public void String format (Map<String,String> metadata, String text){

StringBuffer str=new StringBuffer(header);

metadata.forEach((k, v) -> str.append(("<p>"+k + ":" + v)));

str.append("<p>"+text);

return str.toString();

}

private String header=”<!DOCTYPE html><html><body>”

private String end=”</body></html>”

}

Client code:

/\*client code\*/

public class Library{

//any one of the use is enough

public void printPaper(Book book){

book.setFormatter(paperformatter);

String txt=book.print();

System.out.println(“printing “+txt+” on paper”);

}

private Formatter paperformat=new paperFormatter();

//another example use

public void printWeb(Media item){

item.setFormatter(new HTMLFormatter());

String txt=(book.print());

System.out.println(“printing “+txt+” on web”);

}

//another use

public void someMethod(){

Movie m=new Movie(paperformat);

System.out.println(“printing “+m.print()+” on paper”);

}

//other methods and some private attributes related to library

}

5.

**SRP**: Separates media abstraction from print formatting implementation. This relieves the media from formatting responsibility. Implementation has the single responsibility of formatting.

**OCP**: Allows extending both media abstractions and formatting implementations independently. (EXPECTED)

**LSP**: Subclasses of Media Abstractions can be substituted without affecting correctness. Similar for implementation classes. (OPTIONAL, least expected)

**ISP**: Promotes focused interfaces for both media abstractions and formatting implementations. (OPTIONAL)

**DIP**: Decouples high-level library item abstractions from low-level formatting implementations, relying on interfaces.