Formating Instructions: Please use this provided LATEX template to complete your homework. When including figures such as UMLs, it is highly encouraged to use tools (graphviz, drawio, tikz, etc..). Figures may be hand draw, however, these may not receive credit if the grader cannot read it. For ease of grading, when including code please have it as part of the same pdf as the question while also including correct formatting/indent, preferably syntax highlighting. Latex includes the minted, or lstlisting package as a helpful tool. For this assignment all code must be full code (no pseudo-code) and be written in either Java or C++. However, implements may ignore all logic not relevant to the design pattern with simple print out statements of "[BLANK] logic done here"

Question Instructions: In this homework assignment, you will apply one or more Structural patterns discussed in the lectures.
This is a group assignment that requires two students per group.
For each question:

1. Give the name of the design pattern(s) you are applying to the problem.
2. Present your reasons why this pattern will solve the problem. Please be specific to the problem and do not give general applicability statements. If there is an alternative pattern, explain why you preferred this one..
3. Show you design with a UML class diagram. If the pattern collaborations would be more visible with another diagram (e.g. sequence diagram), give that diagram as well.
   (a) Your diagram should show every participant in the pattern including the pattern related methods.
   (b) In pattern related classes, give the member (method and attribute) names that play a role in the pattern and effected by the pattern. Optionally, include the member names mentioned in the question. You are encouraged to omit the other methods and fields.
   (c) For the non-pattern related classes, you are not expected to give detailed class names etc. You may give a high-level component, like "UserInterface" or "DBManagement"
4. Give Java or C++ code for your design showing how you have implemented the pattern.
   (a) Pattern related methods and attributes should appear in the code
   (b) Client usage of the pattern should appear in the code
   (c) Non-pattern related parts of the methods could be a simple print. (e.g. "System.out.println()", "cout")
5. Evaluate your design with respect to SOLID principles. Each principle should be address, if a principle is not applicable to the current pattern, say so.

1. (14 points) 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 (1-5), rarity (common, rare, scarce), effect (healing, increaseSize), bearer (null or player), and 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. (address all items 1-5)

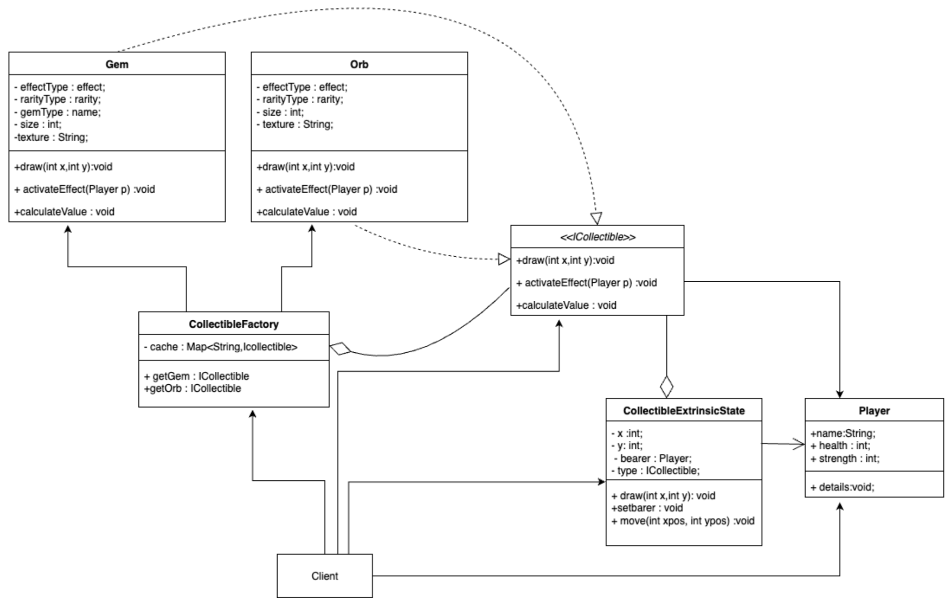
   1) Using the Flyweight design pattern.

   2) The flyweight pattern is used to minimise the usage of memory by sharing the objects instead of duplicating them. In this game scenario:

   - Gems and Orbs have many features like effect, name, rarity, size, and texture.
   - Without Flyweight, each collectible object will store all these attributes repeatedly, consuming a lot of memory.
   - By applying the Flyweight Pattern:
     - The intrinsic state is created with features name,effect, rarity, size, texture into a single Gem or Orb flyweight object.
     - The extrinsic state is created with unique properties like position,bearer is stored separately in a lightweight wrapper.

     This way, the client can reuse the same Gem or Orb flyweight object to represent many collectibles at different positions on the map without recreating full objects.

   3) Diagram inserted



4)

```
import java.util.*;

class Player
{
 String name;
 int health=100,strength=100;

Player(String name)
{
    this.name = name;
}
public void details()
{
    System.out.println("Player name:" +name + " health: " +health+
    " strength: " +strength);
}
}
//Flyweight
interface ICollectible
{
    void draw(int x,int y);
    void activateEffect(Player p);
    void calculateValue();

}

//Concrete Flyweight
```

```
class Orb implements ICollectible
{

  enum rarity { common, rare, scarce }
  enum effect { healing , increasesize}

  private effect effectType;
  private rarity rarityType;
  private int size;
  private String texture;
  Orb (effect effectType,rarity rarityType,int size,String texture )
  {
    this.effectType =effectType;
    this.rarityType =rarityType;
    this.size =size;
    this.texture=texture;

  }
  @Override
public void draw(int x,int y)
  {
    System.out.println("Position of the orb is:" +x+ "," +y);
  }



@Override
public void calculateValue()
{

int orbValue =0;
//rarity,size,effect
switch(rarityType)
{
    case common:
        orbValue +=5;
        break;
    case rare:
        orbValue +=15;
        break;
    case scarce:
        orbValue +=25;
        break;
}
switch(effectType)
{
    case healing:
        orbValue +=5;
        break;
    case increasesize:
        orbValue +=15;
        break;
}
orbValue += size*3;
System.out.println("Orb Value : " +orbValue);
}
@Override
public void activateEffect(Player p)
{
    if(p==null)
    {
        return;
    }
    else
    {
switch(effectType)
{
    case healing:
        p.health += 5;
        break;
    case increasesize:
        p.strength +=10;
        break;
}
System.out.println("Effect " +effectType+ " activated.");
    }

}
}


//Concrete Flyweight
class Gem implements  ICollectible
{
enum name { ruby, sapphire, emerald}
enum rarity { common, rare, scarce }
enum effect { strength, health}
private int size;
private String texture;
  private effect effectType;
  private rarity rarityType;
  private name gemType;
Gem(name gemType,rarity rarityType,effect effectType,int size,String texture)
{
    this.gemType=gemType;
    this.rarityType =rarityType;
    this.effectType=effectType;
    this.size=size;
    this.texture=texture;
}
@Override
public void draw(int x,int y)
  {
     System.out.println("Position of the gem is:" +x+ "," +y);
  }
  @Override
  public void calculateValue()
{

int gemValue =0;
//rarity,size,name
switch(rarityType)
{
    case common:
        gemValue +=5;
        break;
    case rare:
        gemValue +=15;
        break;
    case scarce:
        gemValue +=25;
        break;
}
switch(gemType)
{
    case ruby:
        gemValue +=5;
        break;
    case sapphire:
        gemValue +=15;
        break;
    case emerald:
        gemValue +=25;
        break;
}
gemValue += size*3;
System.out.println("Gem value is: " +gemValue);
}
@Override
public void activateEffect(Player p)
{
    if(p==null)
    {
        return;
    }
    else
    {
switch(effectType)
{
    case strength:
        p.strength += 5;
        break;
    case health:
        p.health +=10;
        break;
}
System.out.println("Effect " +effectType+ " activated.");
    }

}

}


class CollectibleExtrinsicState
{

    private int x,y;
    private Player bearer;
    private ICollectible type;

    public CollectibleExtrinsicState(ICollectible type, int x, int y) {

        this.type=type;
        this.x=x;
        this.y=y;
    }
    public void draw()
    {
        type.draw(x,y);
    }
    public void setbarer(Player p)
    {
        this.bearer=p;
        type.activateEffect(bearer);
    }

    public void move(int xPosition, int yPosition)
    {
         x += xPosition;
         y += yPosition;
         System.out.print("Item moved,New ");
        type.draw(x,y);

    }

}

//Flyweight Factory
class CollectibleFactory
{
private static Map<String,ICollectible> cache = new HashMap<>();
public static ICollectible getOrb(Orb.effect effectType,
Orb.rarity rarityType,int size,String texture)
{
String key = "orb-" + effectType + "-" + size + "-" +
rarityType + "-" + texture;

    ICollectible orb = cache.get(key);
        if(orb == null)
        {
            orb = new Orb(effectType,rarityType,size,texture);
            cache.put(key,orb);
        }

    return orb;
}
public static ICollectible getGem(Gem.name gemType,Gem.rarity rarityType,
Gem.effect effectType,int size,String texture)
{
String key = "gem-" + gemType + "-" + size + "-" +
rarityType + "-" + texture + "-" + effectType;

    ICollectible gem = cache.get(key);
        if(gem== null)
        {
           gem = new Gem(gemType,rarityType,effectType,size,texture);
           cache.put(key,gem);

        }

    return gem;
}

}
```

```
//Client
class GameDemo
{
    public static void main(String[] args) {

        Player player1 = new Player("Denise");
        ICollectible orb1 =CollectibleFactory.getOrb(Orb.effect.healing,
        Orb.rarity.rare,5,"orb.png");
        ICollectible gem1 = CollectibleFactory.getGem(Gem.name.emerald,
        Gem.rarity.common, Gem.effect.strength, 5,"gem.png");


        CollectibleExtrinsicState c1 = new CollectibleExtrinsicState(orb1,2,3);
        System.out.println("-----------------Orb details----------------- ");
        orb1.calculateValue();
        c1.draw();
        player1.details();
        c1.setbarer(player1);
        c1.move(3,5);
        player1.details();
        CollectibleExtrinsicState c2 = new CollectibleExtrinsicState(orb1,6,7);
        c2.draw();
        player1.details();
        c2.setbarer(player1);
        player1.details();
        CollectibleExtrinsicState g1 = new CollectibleExtrinsicState(gem1,9,10);
        System.out.println("-----------------Gem details----------------- ");
        gem1.calculateValue();
        g1.draw();
        player1.details();
        g1.setbarer(player1);
        player1.details();


    }
}
```

5) SOLID PRINCIPLES:

- Single Responsibility Principle :

  Each class in the design has one responsibility: Each class established in the flyweight pattern has only one responsibility to carry on.

- Open/Closed Principle :

  The design is open for extension but closed for modification: The system can be extented by adding a new Collectible item and the factory method can be extended without any modification to the code

- Liskov Substitution Principle :

Both the Gem and Orb can be used where ever the ICollectible is expected.

- Interface Segregation Principle:

  The interface ICollectible only has essential methods related to the Collectibles.

- Dependency Inversion Principle :

  The CollectibleExtrinsicState and CollectibleFactory depends on the ICollectible not on the concerete classes Gem or Orb.

2. (14 points) 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 (not the video) 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. (address all items 1-5)

   1) (1 points) Bridge

   2) (1 points)

   The Bridge pattern separates the media type (Book and Movie) from the formatter type (HTML or paper). Adaptor pattern is another possible design pattern because its intent is to make incompatible things work together, but you still have to create multiple adaptors classes in order to work. With bridge pattern, the total number of classes is minimal.

   3) (5 points)

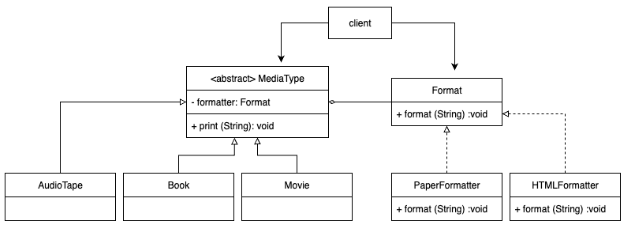


Figure 1: Enter Caption

   4)

```java
// Implementer Interface
interface Format {
    void format(String itemType);
}

// Concrete Implementer 1 (Web Platform)
class HTMLFormatter implements Format {
    @Override
    public void format(String itemType) {
        // Platform-specific logic
        System.out.println("HTMLFormatter");
        System.out.println(itemType);
    }
}

// Concrete Implementer 2 (Paper Platform)
```

```java
class PaperFormatter implements Format {
    @Override
    public void format(String itemType) {
        // Platform-specific logic
        System.out.println("PaperFormatter");
        System.out.println(itemType);

    }
}

// Abstraction
abstract class MediaType {
    protected Format formatter;

    protected MediaType(Format formatter) {
        this.formatter = formatter;
    }

    // delegates to the Implementer
    public void print(String type) {
        formatter.format(type);
    }
}

// Refined Abstraction 1
class Book extends MediaType {

    public Book(Format formatter) {
        super(formatter);
    }
}

// Refined Abstraction 2
class Movie extends MediaType {
    public Movie(Format formatter) {
        super(formatter);
    }
}

// Refined Abstraction 3
class AudioTape extends MediaType {
    public AudioTape(Format formatter) {
        super(formatter);
    }
}
```

```java
class Client {
    public static void main(String[] args) {
        Format webFormat = new HTMLFormatter();
        Format paperFormat = new PaperFormatter();

        MediaType book = new Book(webFormat);
        // print book in web format
        book.print("Book");

        System.out.println();

        book = new Book(paperFormat);
        // print book in paper format
        book.print("book");

        System.out.println();

        MediaType movie = new Movie(paperFormat);
        // print movie info in paper format
        movie.print("movie");

        System.out.println();

        movie = new Movie(webFormat);
        // print movie info in web format
        movie.print("movie");

        System.out.println();

        MediaType audioTape = new AudioTape(paperFormat);
        // print audio tape info in paper format
        audioTape.print("audoio tape");

        System.out.println();

        audioTape = new Movie(webFormat);
        // print audio tape info in web format
        audioTape.print("movie");
    }
}
```

5) (1 points)

Single Responsibility Principle

The design separates responsibilities across the two hierarchies: Media Type classes Book, Movie are only responsible for managing item data. Format classes HTMLFormatter, PaperFormatter are only responsible for platform specific formatting.

Open/Closed Principle

The design is open for extension but closed for modification: To add a new item type e.g., AudioTape, you extend MediaType without changing any existing Format classes. To add a new platform, you implement Format without changing any existing MediaType classes.

Liskov Substitution Principle

Subclasses can be substituted for their base types without errors: Any MediaType object Book or Movie can be treated as a generic MediaType, and its print method will function correctly.

Interface Segregation Principle

The Format interface format is minimal and cohesive, containing only the method necessary for printing/formatting.

Dependency Inversion Principle

High level modules depend on abstractions, not concretions: The high level abstraction MediaType depends on the Format interface abstraction, not the concrete classes like HTMLFormatter. This injection of the Format interface into MediaType inverts the dependency, allowing the two hierarchies to vary independently.

Table 1: Grading Rubric for **14** points questions

1 (1 point)	0	missing or incorrect
	+1	correct pattern
2 (1 point)	0	missing
	+1	the reason provided correctly describes an advantage of the pattern and is specifically beneficial to this scenario
3 (5 points)	0	missing
	+2	includes all participants (including client) that play a role in the pattern
	+2	all class relations are correct
	+1	includes all class members that are related to the pattern
4 (5 points)	0	missing
	+1	includes all pattern related methods and attributes
	+2	includes client usage
	+2	correctly implements and uses all pattern related methods
5 (2 points)	0	missing
	+2	correctly lists multiple ways the pattern benefits a user