Heaven's Light is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Lab Report

Course No: CSE 3206

Course Title: Software Engineering Sessional

Index :

Submitted By	Submitted To
Group 04 **Consists of** **2003160,2003161,2003162**	**Farjana Parvin** **Lecturer** **Department of CSE, RUET**

Facade Design Pattern

Introduction: The Facade Design Pattern is a structural design pattern that provides a simplified interface to a complex subsystem or set of subsystems. It hides the complexity of the system by providing a higher-level interface that makes it easier for clients to interact with the system.

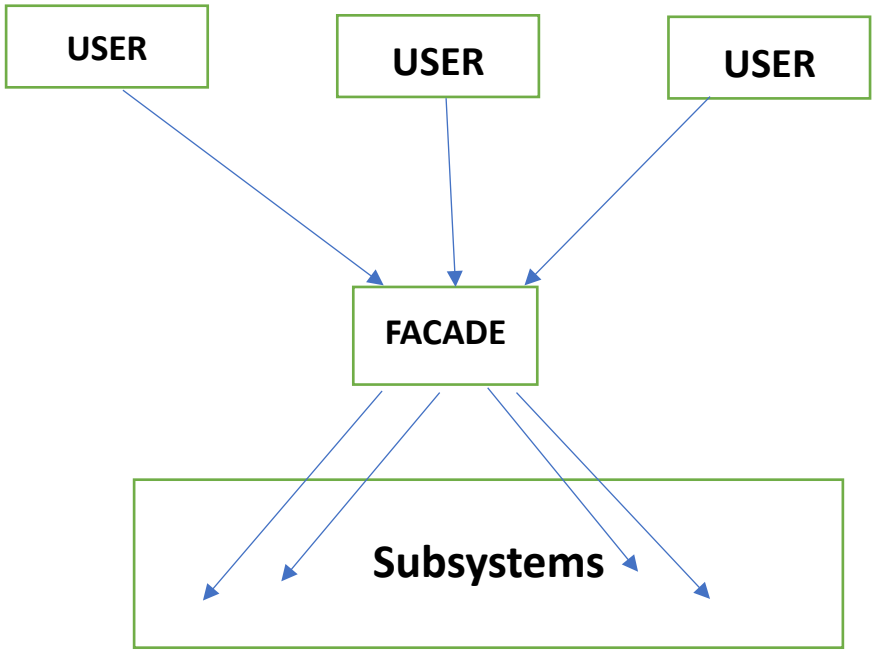
Objective: The primary objective of the Facade Design Pattern is to simplify the interface for interacting with a complex system or set of subsystems. It provides a higher-level abstraction that hides the complexity and makes it easier for the client to interact with the system.

Problem Statement:
Title: Design a Simplified Home Theater Controller Using Facade Pattern.

Scenario:
You are tasked with designing a simplified Home Theater Controller for a user who wants to enjoy movie-watching experience without the hassle of managing multiple devices individually. The system includes the following components:

1. **Room Lights**: Can be dimmed or brightened.
2. **Sound System**: Can be turned on or off.
3. **Television**: Can be turned on or off.

UML DIAGRAM:



Code:

```
// Interface for Television
interface ITelevision {
    void switchOnTv();
    void switchOffTv();
}

// Interface for RoomLights
interface IRoomLights {
    void lightsDim();
    void lightsBright();
}

// Interface for SoundSystem
interface ISoundSystem {
    void switchOnSoundSystem();
    void switchOffSoundSystem();
}
```

```
// Implementation of Television
class Television implements ITelevision {
    @Override
    public void switchOnTv() {
        System.out.println("TV on");
    }

    @Override
    public void switchOffTv() {
        System.out.println("TV off");
    }
}

// Implementation of RoomLights
class RoomLights implements IRoomLights {
    @Override
    public void lightsDim() {
        System.out.println("Lights dimmed");
    }

    @Override
    public void lightsBright() {
        System.out.println("Lights brightened");
    }
}

// Implementation of SoundSystem
class SoundSystem implements ISoundSystem {
    @Override
    public void switchOnSoundSystem() {
        System.out.println("Sound System is on");
    }

    @Override
    public void switchOffSoundSystem() {
        System.out.println("Sound System is off");
    }
}

// HomeTheatreFacade class
class HomeTheatreFacade {
    private IRoomLights roomLights;
    private ISoundSystem soundSystem;
    private ITelevision television;
```

```
    // Constructor
    public HomeTheatreFacade(IRoomLights roomLights, ISoundSystem soundSystem,
ITelevision television) {
        this.roomLights = roomLights;
        this.soundSystem = soundSystem;
        this.television = television;
    }

    // Method to start movie experience
    public void watchMovie() {
        System.out.println("Starting movie experience...");
        soundSystem.switchOnSoundSystem();
        television.switchOnTv();
        roomLights.lightsDim();
        System.out.println("Enjoy your movie!");
    }

    // Method to end movie experience
    public void stopWatchingMovie() {
        System.out.println("Ending movie experience...");
        soundSystem.switchOffSoundSystem();
        television.switchOffTv();
        roomLights.lightsBright();
        System.out.println("Goodbye!");
    }
}

// User class (Main Class)
public class client {
    public static void main(String[] args) {
        // Create subsystem instances
        IRoomLights roomLights = new RoomLights();
        ISoundSystem soundSystem = new SoundSystem();
        ITelevision television = new Television();

        // Create the facade instance
        HomeTheatreFacade facade = new HomeTheatreFacade(roomLights, soundSystem,
television);

        // Use the facade to control the home theater system
        facade.watchMovie();
        facade.stopWatchingMovie();
    }
}
```

Output:

```
PS D:\3206> cd "d:\3206"
PS D:\3206> cd "d:\3206\" ; if ($?) { javac User.java } ; if ($?) { java User }
Starting movie experience...
Sound System is on
TV on
Lights dimmed
Enjoy your movie!
Ending movie experience...
Sound System is off
TV off
Lights brightened
Goodbye!
PS D:\3206>
```

Conclusion

The **Facade Design Pattern** serves the objective of **streamlining interactions** with complex systems and **enhancing maintainability** by abstracting away the complexity. It makes systems easier to use, more flexible, and more maintainable over time.

Flyweight/Cache Design Pattern:

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

Problem:

Imagine you are developing a 2D game where trees are scattered across a large map. Each tree has attributes such as type (oak, pine), texture, color, and position. In a dense forest, there could be hundreds of thousands of trees, leading to significant memory usage if each tree is represented by a separate object.

The problem gets worse as the forest grows. Creating and managing so many objects puts a heavy load on the system, slowing the game down and even making it crash on devices with less memory.

Solution:

Simplifying What Needs to Be Stored:

The Flyweight pattern starts by asking**: "What data is the same for multiple trees, and what data is unique to each tree?"**

Shared Data (Intrinsic Properties): Things like tree type, texture, and color are the same for many trees. These don't need to be stored repeatedly.

Unique Data (Extrinsic Properties): Each tree's position on the map is different. This has to stay unique.

By splitting the data this way, we can focus on storing the shared information in a single place and only dealing with the unique data when it's needed.

Flyweight Factory for Reusability:

A Flyweight Factory manages the creation and reuse of shared (intrinsic) objects. When a new tree type is requested, the factory checks if an object for that type already exists:

- If it exists, the factory reuses it.
- If it doesn't, a new object is created and stored for future use.

This process ensures that shared data is stored only once, regardless of the number of objects in the system.

Keeping Unique Details Flexible:

The Flyweight pattern doesn't store the position (or other unique details) in the shared objects. Instead, this information is passed to the shared object when it's time to draw the tree. For example:

- The shared "Oak" object gets called with positions like (10, 20) or (50, 100) to draw the trees in different places.
- This keeps the system flexible while still saving a huge amount of memory.

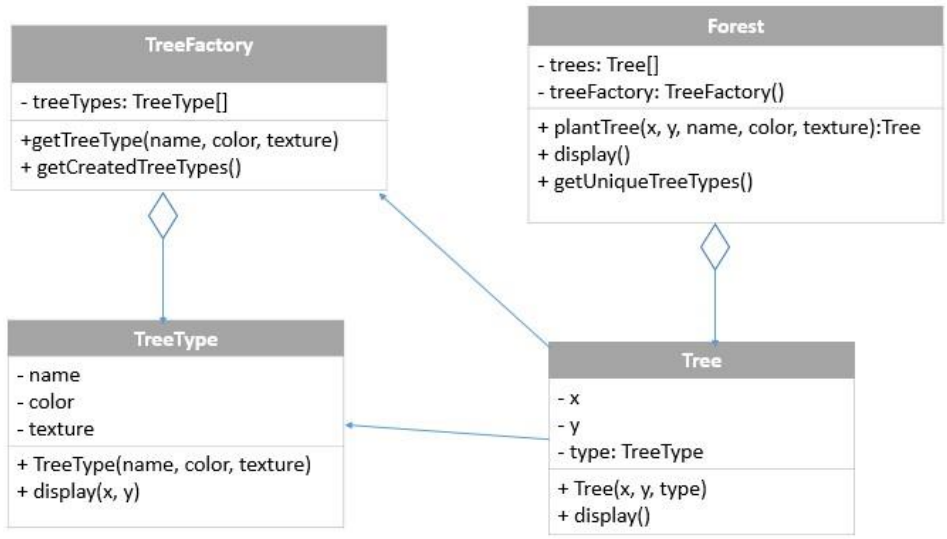
Memory and Performance Benefits:

With Flyweight:

- The number of objects drops dramatically (e.g., 2 shared objects for 100,000 trees).
- Memory usage shrinks because shared data isn't duplicated.
- The game runs faster with less strain on the system.

- In short, Flyweight reuses shared data and processes unique details dynamically, making the solution lightweight and scalable.

UML Diagram:



Code:

```
import java.util.ArrayList;
import java.util.HashMap; import
java.util.List; import
java.util.Map; // Flyweight
class class TreeType {
private final String name;
private final String color;
private final String texture;
     public TreeType(String name, String color, String texture)
{         this.name = name;         this.color = color;
this.texture = texture;
    }     public void display(int x, int y) { // Extrinsic state provided
here
```

```
        System.out.println("Tree of type: " + name + ", Color: " + color + ",
Texture: " + texture + " at (" + x + ", " + y + ")");
    }
}

// Flyweight Factory class TreeFactory {     private final Map<String,
TreeType> treeTypes = new HashMap<>();

    public TreeType getTreeType(String name, String color, String texture) {
String key = name + "-" + color + "-" + texture;
treeTypes.putIfAbsent(key, new TreeType(name, color, texture));         return
treeTypes.get(key);
    }      public int
getCreatedTreeTypes() {         return
treeTypes.size();
    }
}

// Context Class class Tree
{     private final int x;
private final int y;
    private final TreeType type;
     public Tree(int x, int y, TreeType type) {
this.x = x;         this.y = y;         this.type
= type;
    }      public void
display() {
type.display(x, y);
    }
}

// Client class Forest {     private final List<Tree> trees = new
ArrayList<>();     private final TreeFactory treeFactory = new
TreeFactory();
     public void plantTree(int x, int y, String name, String color, String
texture) {
        TreeType type = treeFactory.getTreeType(name, color, texture);
```

```
        trees.add(new Tree(x, y, type));
    }      public void display()
{         for (Tree tree :
trees) {
tree.display();
        }
    }      public int getUniqueTreeTypes() {
return treeFactory.getCreatedTreeTypes();
    }
    public static void main(String[] args) {
        Forest forest = new Forest();
        // Plant Oak trees         for (int i = 0; i < 100; i++) {
forest.plantTree(i % 10, i / 10, "Oak", "Green", "GreenLeaves");         }

        // Plant Pine trees         for (int i = 0; i < 100; i++) {
forest.plantTree(i % 10 + 50, i / 10 + 50, "Pine", "DarkGreen",
"DarkGreenLeaves");
        }

        // Display the forest
forest.display();

        // Display unique tree types created
        System.out.println("Unique tree types created: " +
forest.getUniqueTreeTypes());
    }
}
                                                                                \
```

Output:

```
D:\3-2\3206>cd "d:\3-2\3206\" && javac Forest.java && java Forest
Tree of type: Oak, Color: Green, Texture: GreenLeaves at (0, 0)
Tree of type: Oak, Color: Green, Texture: GreenLeaves at (1, 0)
Tree of type: Oak, Color: Green, Texture: GreenLeaves at (2, 0)
Tree of type: Oak, Color: Green, Texture: GreenLeaves at (3, 0)
Tree of type: Oak, Color: Green, Texture: GreenLeaves at (4, 0)
Tree of type: Oak, Color: Green, Texture: GreenLeaves at (5, 0)
Tree of type: Oak, Color: Green, Texture: GreenLeaves at (6, 0)
Tree of type: Oak, Color: Green, Texture: GreenLeaves at (7, 0)
Tree of type: Oak, Color: Green, Texture: GreenLeaves at (8, 0)
Tree of type: Oak, Color: Green, Texture: GreenLeaves at (9, 0)
Tree of type: Oak, Color: Green, Texture: GreenLeaves at (0, 1)
```

.....

```
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (51, 51)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (52, 51)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (53, 51)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (54, 51)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (55, 51)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (56, 51)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (57, 51)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (58, 51)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (59, 51)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (50, 52)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (51, 52)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (52, 52)
```

.....

```
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (52, 59)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (53, 59)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (54, 59)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (55, 59)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (56, 59)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (57, 59)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (58, 59)
Tree of type: Pine, Color: DarkGreen, Texture: DarkGreenLeaves at (59, 59)
Unique tree types created: 2
```

Design Pattern Used: Structural Design Pattern

Definition of Facade:

- Facade provides a simplified interface to a complex system.
- Helps in reducing dependencies and improving maintainability.

Problem Statement:

1. **Handling Multiple Objects** ○ When working with a **large library or framework**, developers need to deal with multiple classes and objects. ○ Each object might have different methods, dependencies, and initialization requirements.
2. **Managing Complex Dependencies**
   - Clients (the application using the library) must initialize several components in the correct order. ○ Dependencies between different components must be managed carefully.
   - Any small mistake can cause errors or unexpected behavior.
3. **Tightly Coupled Code** ○ If the client directly interacts with multiple subsystem classes, it becomes **highly dependent** on them. ○ Changes in the library may **break existing code**, requiring modifications across multiple parts of the application. ○ Maintenance becomes difficult, and the codebase becomes **harder to scale**.

Why This is a Problem?

- Increases **development time** because of unnecessary complexity.

- Reduces **code reusability**, as the logic is spread across many dependent components.
- Makes the system **less flexible**, as changing the underlying subsystem requires updating all dependent parts.

Solution - Facade Pattern

2. **What is a Facade Class?**
    - A **Facade** is a single class that provides a **simplified, unified interface** to a complex system. ◦ Instead of working directly with multiple classes in a subsystem, the client interacts with the facade, which manages the complexity internally.
3. **Encapsulating Complexity** ◦ Large frameworks or libraries often contain **many interdependent classes**, making them difficult to use.
    - The **Facade Pattern encapsulates** this complexity by grouping commonly used functionalities into a single, easyto-use interface. ◦ This allows developers to focus on what they need without worrying about the underlying implementation.
4. **Providing a High-Level API** ◦ The facade offers **a simple set of methods** that abstract away the intricate details of the subsystem.
    - It serves as a **high-level API**, making interactions straightforward and reducing the learning curve for developers.
5. **How Clients Benefit?**
    - Clients (applications using the system) no longer need to handle multiple objects or maintain complex dependencies.

- They simply call the **facade's methods**, and it internally manages communication with various subsystem components.
- This leads to **cleaner, more maintainable, and loosely coupled** code.

Example:

- Suppose an application needs to **convert a video format** using a professional video processing library.
- Instead of calling multiple classes (CodecFactory, BitrateReader, AudioMixer), the client can use a **single VideoConverter class (Facade)** with an easy-to-use method:

Code Implementation:

```
// These are some of the classes of a complex 3rd-party video
conversion framework.
// We don't control this code, so we can't simplify it.

class VideoFile {
private String filename;
private String format;

    public VideoFile(String filename) {
this.filename = filename;
        this.format = filename.substring(filename.lastIndexOf(".") +
1);
    }
    public String getFilename() {
return filename;
    }
    public String getFormat() {
return format;
    }
}
interface Codec {
String getType();
}
```

```
class OggCompressionCodec implements Codec {
public String getType() {         return
"ogg";
    }
}
class MPEG4CompressionCodec implements Codec {
public String getType() {         return
"mp4";
    }
}
class CodecFactory {
    public static Codec extract(VideoFile file) {
        System.out.println("CodecFactory: Extracting codec from file:
" + file.getFilename());         return
file.getFormat().equals("mp4") ? new
MPEG4CompressionCodec() : new OggCompressionCodec();
    }
}
class BitrateReader {
    public static String read(String filename, Codec codec) {
System.out.println("BitrateReader: Reading file - " + filename
+ " with codec - " + codec.getType());
return "RAW_VIDEO_DATA";
    }
    public static String convert(String buffer, Codec codec) {
System.out.println("BitrateReader: Converting to format - " +
codec.getType());
        return "CONVERTED_VIDEO_DATA";
    }
}
class AudioMixer {
    public String fix(String result) {
        System.out.println("AudioMixer: Fixing audio...");
return result + " (Audio Fixed)";
    }
}

// We create a Facade class to hide the framework's complexity behind
a simple interface. class VideoConverter {
    public File convert(String filename, String format) {
System.out.println("\n=== Video Conversion Started ===");
        VideoFile file = new VideoFile(filename);
        Codec sourceCodec = CodecFactory.extract(file);
        Codec destinationCodec = format.equals("mp4") ? new
```

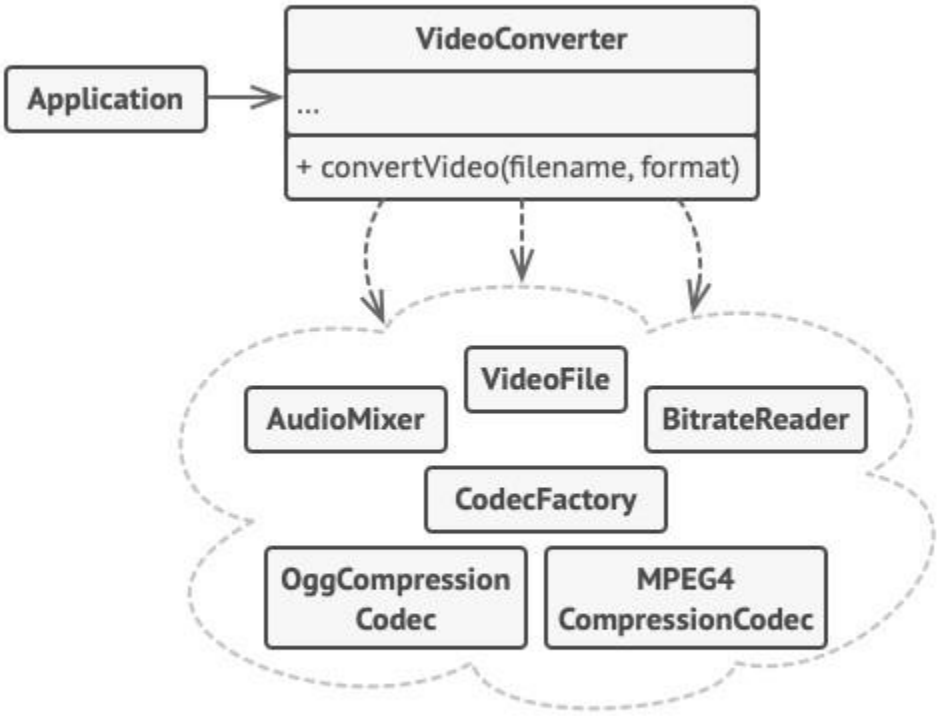
```
MPEG4CompressionCodec() : new OggCompressionCodec();
        String buffer = BitrateReader.read(filename, sourceCodec);
        String result = BitrateReader.convert(buffer,
destinationCodec);
        result = new AudioMixer().fix(result);
        System.out.println("=== Video Conversion Completed ===\n");
return new File(result, filename.replace(file.getFormat(), format));
    }
}

// File class to simulate file creation after conversion
class File {     private String data;     private
String filename;

    public File(String data, String filename) {
this.data = data;         this.filename =
filename;
    }
    public void save() {
        System.out.println("File: " + filename + " has been saved
successfully!\n");
    }
}

// Application class using the Facade public
class Application {
    public static void main(String[] args) {
        VideoConverter converter = new VideoConverter();
        File mp4 = converter.convert("funny-cats-video.ogg", "mp4");
mp4.save();
    }
}
```

UML Diagram:



An example of isolating multiple dependencies within a single facade class.

Output:

```
=== Video Conversion Started ===
CodecFactory: Extracting codec from file: funny-cats-video.ogg
BitrateReader: Reading file - funny-cats-video.ogg with codec - ogg
BitrateReader: Converting to format - mp4
AudioMixer: Fixing audio...
=== Video Conversion Completed ===

File: funny-cats-video.mp4 has been saved successfully!
```