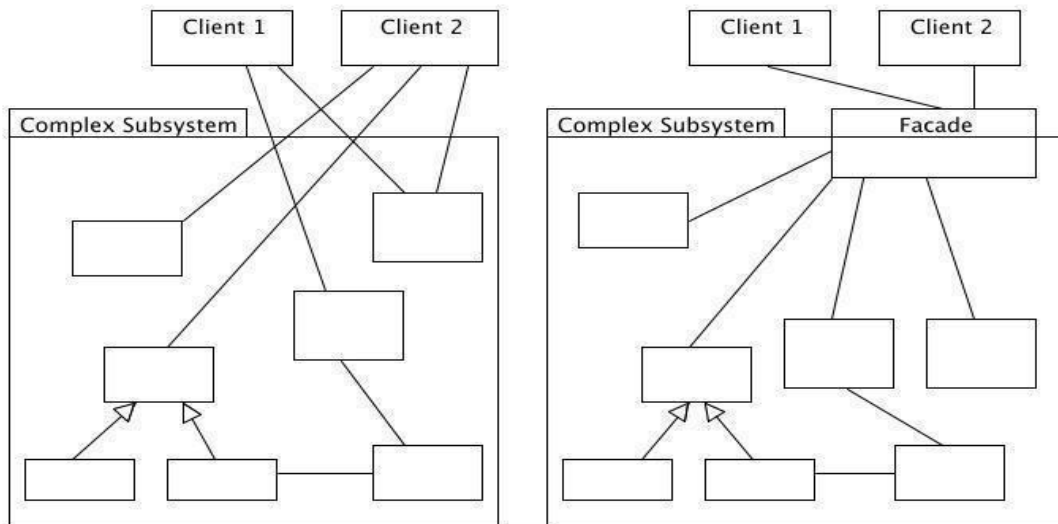

Lab Session # 12

Objective: To understand Façade and Singleton Design Patterns

Facade Pattern:

The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher level interface that makes the subsystem easier to use.



What Does Façade Mean??

A facade or façade is generally one exterior side of a building, usually, but not always, the front. The word comes from the French language, literally meaning “frontage” or “face”.

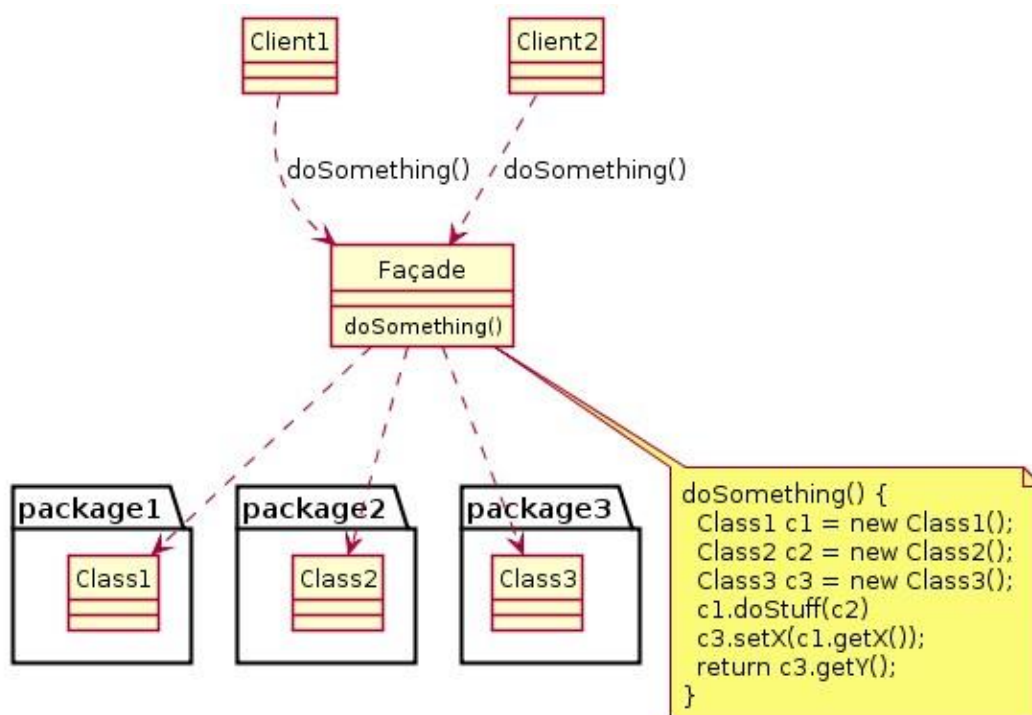
GOF → Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Participants

Facade: Knows which subsystem classes are responsible for a request. Delegates client requests to appropriate subsystem objects.

Subsystem classes: Implements subsystem functionality. Handle work assigned by Facade object. Have no knowledge of the facade; that is, they keep no references to it.

Let us look at a simple representation of this pattern.



Now that we have understood what a Facade actually is and what GOF have interpreted with this, we will go ahead and look at a very simple and real world application of this pattern.

This is an abstract example of how a client (“you”) interacts with a facade (the “computer”) to a complex system (internal computer parts, like CPU and HardDrive).

The operations are somewhat as given below:

1. Processor is first frozen.
2. Memory is then loaded with the help of Hardrive information.
3. Processor then jumps to the location returned by the memory.
4. Processor then executes from that particular location.

First, we will see with the Facade pattern how the code looks like:

```
class CPU {
    public void freeze() { ... }
    public void jump(long position) { ... }
    public void execute() { ... }
}

class Memory {
    public void load(long position, byte[] data) { ... }
}

class HardDrive {
    public byte[] read(long lba, int size) { ... }
}
```

The above classes, viz. **CPU**, **Memory** and **HardDrive** form the **SUB-SYSTEMS**. Now if we were to write a client to call these methods, it would look something like below:

```
class Client {
    public static void main(String[] args) {
        CPU mycpu = new CPU();
        Memory mymemory = new Memory();
        HardDrive hd = new HardDrive();

        mycpu.freeze();
        mymemory.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));
        mycpu.jump(BOOT_ADDRESS);
        mycpu.execute();
    }
}
```

So from the above code:

1. The client knows about the subsystem implementation which is incorrect.
2. There is a strong coupling between Client and the subsystem which is incorrect.
3. There is no generality to the code, thus you need to keep disturbing client code even if the subsystem code changes.

Thus to avoid all the above problems, we will implement the same using a Facade design pattern.

Again, we will look at the subsystem which will remain the same.

```
/* Complex parts */

class CPU {
    public void freeze() { ... }
    public void jump(long position) { ... }
    public void execute() { ... }
}

class Memory {
    public void load(long position, byte[] data) { ... }
}
```

```
class HardDrive {  
    public byte[] read(long lba, int size) { ... }  
}
```

Now we will look at the Facade object participant which comes into the picture.

```
/* Facade */  
  
class ComputerFacade {  
    private CPU processor;  
    private Memory ram;  
    private HardDrive hd;  
  
    public ComputerFacade() {  
        this.processor = new CPU();  
        this.ram = new Memory();  
        this.hd = new HardDrive();  
    }  
  
    public void start() {  
        processor.freeze();  
        ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));  
        processor.jump(BOOT_ADDRESS);  
        processor.execute();  
    }  
}
```

Thus, we have clean implementation and a single object of this class will solve all our problems. Thus this class abstracts the subsystems.

Now we will look at the client.

```
class You {  
    public static void main(String[] args) {  
        ComputerFacade computer = new ComputerFacade();  
        computer.start();  
    }  
}
```

Facade Pattern

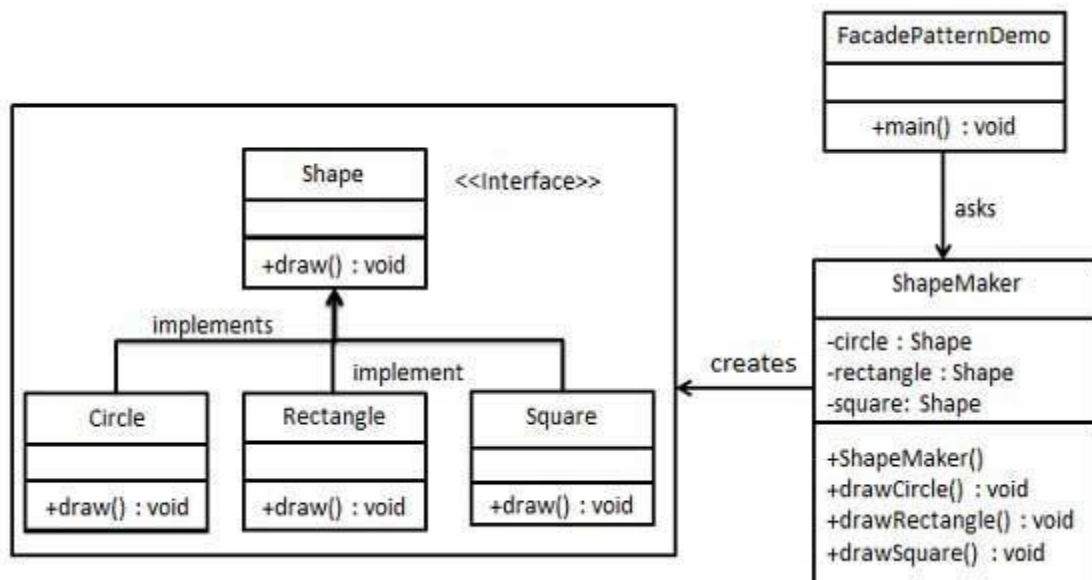
Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

Implementation

We are going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A facade class *ShapeMaker* is defined as a next step.

ShapeMaker class uses the concrete classes to delegate user calls to these classes. *FacadePatternDemo*, our demo class, will use *ShapeMaker* class to show the results.



Step 1

Create an interface.

Shape.java

```
public interface Shape {  
    void draw();  
}
```

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Rectangle::draw()");  
    }  
}
```

Square.java

```
public class Square implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Square::draw()");  
    }  
}
```

```
}  
}
```

Circle.java

```
public class Circle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Circle::draw()");  
    }  
}
```

Step 3

Create a facade class.

ShapeMaker.java

```
public class ShapeMaker {  
    private Shape circle;  
    private Shape rectangle;  
    private Shape square;  
  
    public ShapeMaker() {  
        circle = new Circle();  
        rectangle = new Rectangle();  
        square = new Square();  
    }  
}
```

```

public void drawCircle(){
    circle.draw();
}

public void drawRectangle(){
    rectangle.draw();
}

public void drawSquare(){
    square.draw();
}
}

```

Step 4

Use the facade to draw various types of shapes.

FacadePatternDemo.java

```

public class FacadePatternDemo {

    public static void main(String[] args) {

        ShapeMaker shapeMaker = new ShapeMaker();

        shapeMaker.drawCircle();

        shapeMaker.drawRectangle();

        shapeMaker.drawSquare();

    }

}

```


Step 5

Verify the output.

```
Circle::draw()  
Rectangle::draw()  
Square::draw()
```

Facade Pattern Usage

1. Provide simple interface to complex subsystems
2. When there are many dependencies between clients and implementation class
3. You want to neatly layer your subsystem

To illustrate the facade pattern, Lets try to look into another small example. Let's try to implement a hypothetical facade object to work with some Windows Phone controller objects. First let me define the problem.

Every morning when I go for jogging, I have to make the following changes in my Windows phone device:

1. Turn off the wifi
2. Switch on the Mobile Data
3. Turn on the GPS
4. Turn on the Music
5. Start the Sports-Tracker

And after coming back from jogging, follwing needs to be done from my part:

1. Share Sports tracker stats on twitter and facebook
2. Stop the Sports Tracker
3. Turn off the Music
4. Turn off the GPS
5. Turn off the Mobile Data
6. Turn on the wifi

All this is being done manually right now. So to simulate the behavior lets first implement the subsystem i.e. the dummy controller classes.

```
class GPSController
{
    bool isSwitchedOn = false;

    public bool IsSwitchedOn
    {
        get
        {
            return isSwitchedOn;
        }
        set
        {
            isSwitchedOn = value;
            DisplayStatus();
        }
    }

    private void DisplayStatus()
    {
        string status = (isSwitchedOn == true) ? "ON" : "OFF";
    }
}
```

```

        Console.WriteLine("GPS Switched {0}", status);
    }
}

```

Other controllers like `MobileDataController`, `MusicController`, `WifiController` are also implemented in a similar way.

Now to emulate the App behavior.

```

class SportsTrackerApp
{
    public void Start()
    {
        Console.WriteLine("Sports Tracker App STARTED");
    }

    public void Stop()
    {
        Console.WriteLine("Sports Tracker App STOPPED");
    }

    public void Share()
    {
        Console.WriteLine("Sports Tracker: Stats shared on twitter and facebook.");
    }
}

```

So right now with What I am doing is changing all the settings manually and then starting the App manually. Which can be visualized in form of code as:

```

static void Main(string[] args)
{
    // The phone has been booted up and all the controllers are running
    GPSController gps = new GPSController();
    MobileDataController data = new MobileDataController();
    MusicController zune = new MusicController();
    WifiController wifi = new WifiController();

    ////////// Going for Jogging //////////

    // 1. Turn off the wifi
    wifi.IsSwitchedOn = false;

    // 2. Switch on the Mobile Data
    data.IsSwitchedOn = true;

    // 3. Turn on the GPS
    gps.IsSwitchedOn = true;

    // 4. Turn on the Music
    zune.IsSwitchedOn = true;
}

```

```

// 5. Start the Sports-Tracker
SportsTrackerApp app = new SportsTrackerApp();
app.Start();

////////// Back from Jogging //////////

Console.WriteLine();

// 0. Share Sports tracker stats on twitter and facebook
app.Share();

// 1. Stop the Sports Tracker
app.Stop();

// 2. Turn off the Music
zune.IsSwitchedOn = false;

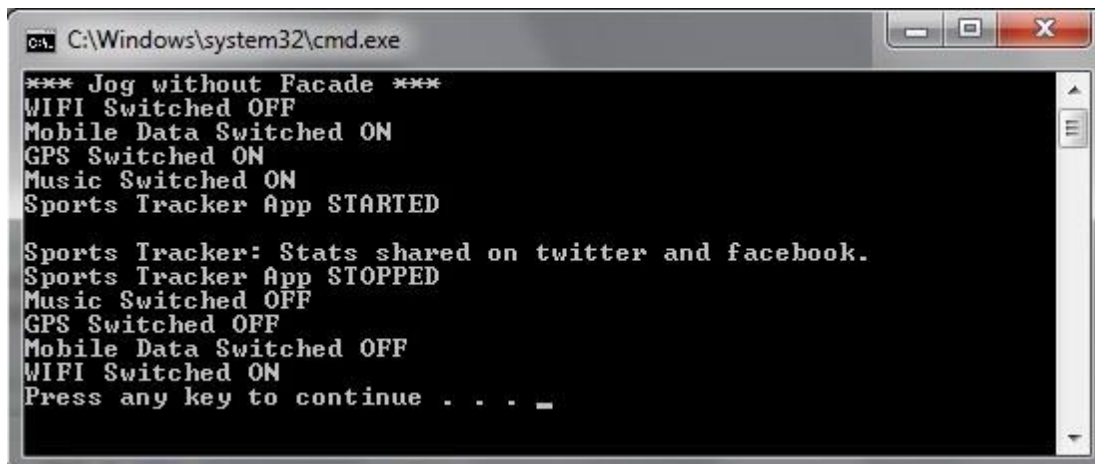
// 3. Turn off the GPS
gps.IsSwitchedOn = false;

// 4. Turn off the Mobile Data
data.IsSwitchedOn = false;

// 5. Turn on the wifi
wifi.IsSwitchedOn = true;
}

```

Note: All this is to emulate the behavior of my Windows Phone. If we run this application the output will be:



```

C:\Windows\system32\cmd.exe
*** Jog without Facade ***
WIFI Switched OFF
Mobile Data Switched ON
GPS Switched ON
Music Switched ON
Sports Tracker App STARTED

Sports Tracker: Stats shared on twitter and facebook.
Sports Tracker App STOPPED
Music Switched OFF
GPS Switched OFF
Mobile Data Switched OFF
WIFI Switched ON
Press any key to continue . . . _

```

Now I have to write myself a facade application that will do all this for me internally. I will write an class that will simply expose two methods like **StartJogging** and **StopJogging** to me. internally it will take care of doing all these activities. So let me write a facade now:

```
class MyJoggingFacade
{
    // These handles will be passed to this facade in a real application
    // also on actual device these controllers will be singleton too.
    GPSController gps = new GPSController();
    MobileDataController data = new MobileDataController();
    MusicController zune = new MusicController();
    WifiController wifi = new WifiController();

    SportsTrackerApp app = null;

    public void StartJogging()
    {
        // 1. Turn off the wifi
        wifi.IsSwitchedOn = false;

        // 2. Switch on the Mobile Data
        data.IsSwitchedOn = true;

        // 3. Turn on the GPS
        gps.IsSwitchedOn = true;

        // 4. Turn on the Music
        zune.IsSwitchedOn = true;

        // 5. Start the Sports-Tracker
        app = new SportsTrackerApp();
        app.Start();
    }

    public void StopJogging()
    {
        // 0. Share Sports tracker stats on twitter and facebook
        app.Share();

        // 1. Stop the Sports Tracker
        app.Stop();

        // 2. Turn off the Music
        zune.IsSwitchedOn = false;

        // 3. Turn off the GPS
        gps.IsSwitchedOn = false;

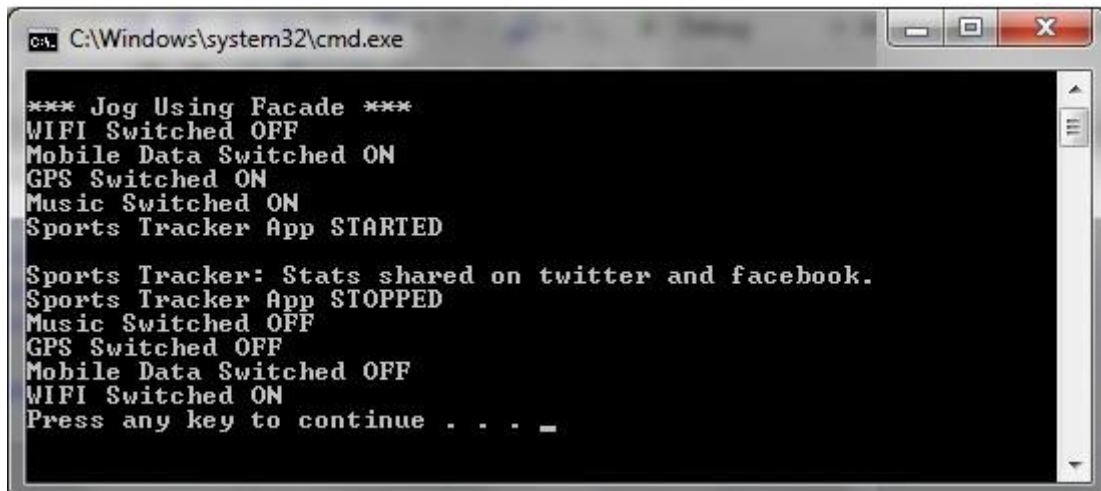
        // 4. Turn off the Mobile Data
        data.IsSwitchedOn = false;

        // 5. Turn on the wifi
        wifi.IsSwitchedOn = true;
    }
}
```

Now the same functionality could be achieved by doing a minimal amount of manual work from the user.

```
static void Main(string[] args)
{
    MyJoggingFacade facade = new MyJoggingFacade();

    facade.StartJogging();
    Console.WriteLine();
    facade.StopJogging();
}
```

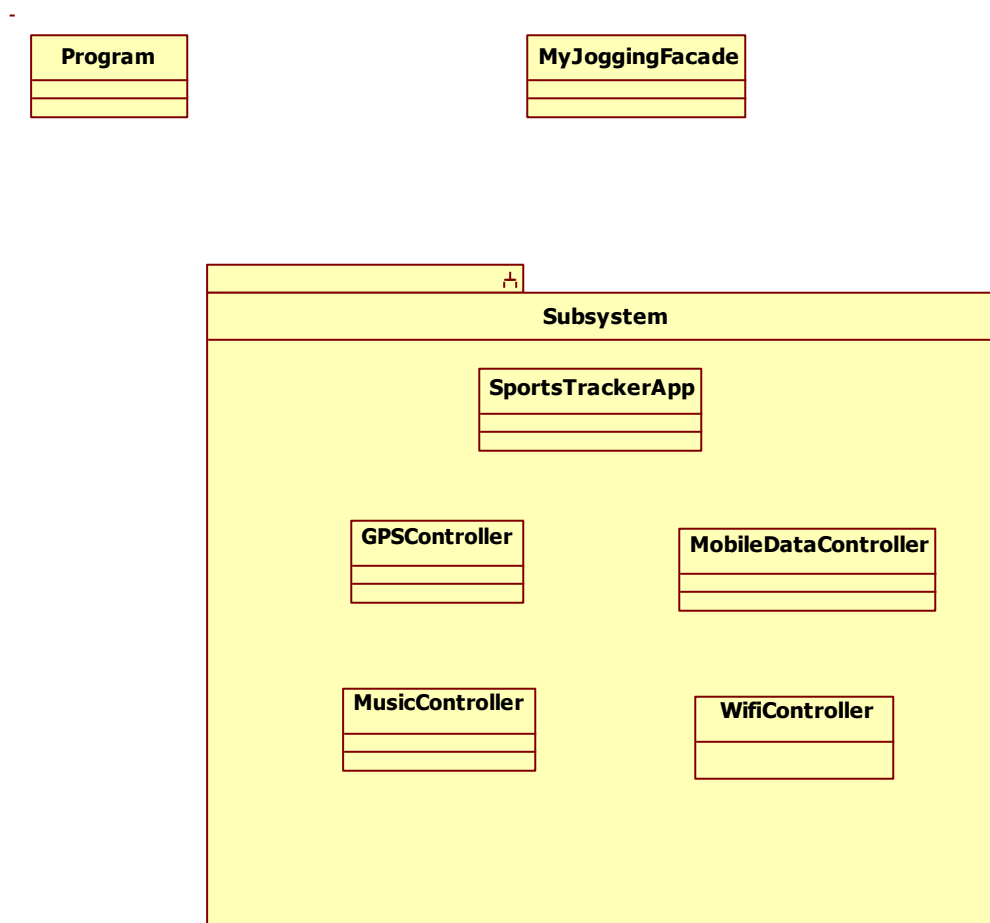


A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window has a black background with white text. The output of the application is as follows:

```
*** Jog Using Facade ***
WIFI Switched OFF
Mobile Data Switched ON
GPS Switched ON
Music Switched ON
Sports Tracker App STARTED

Sports Tracker: Stats shared on twitter and facebook.
Sports Tracker App STOPPED
Music Switched OFF
GPS Switched OFF
Mobile Data Switched OFF
WIFI Switched ON
Press any key to continue . . . _
```

Before summing up let us compare the class diagram of this implementation with the facade pattern:



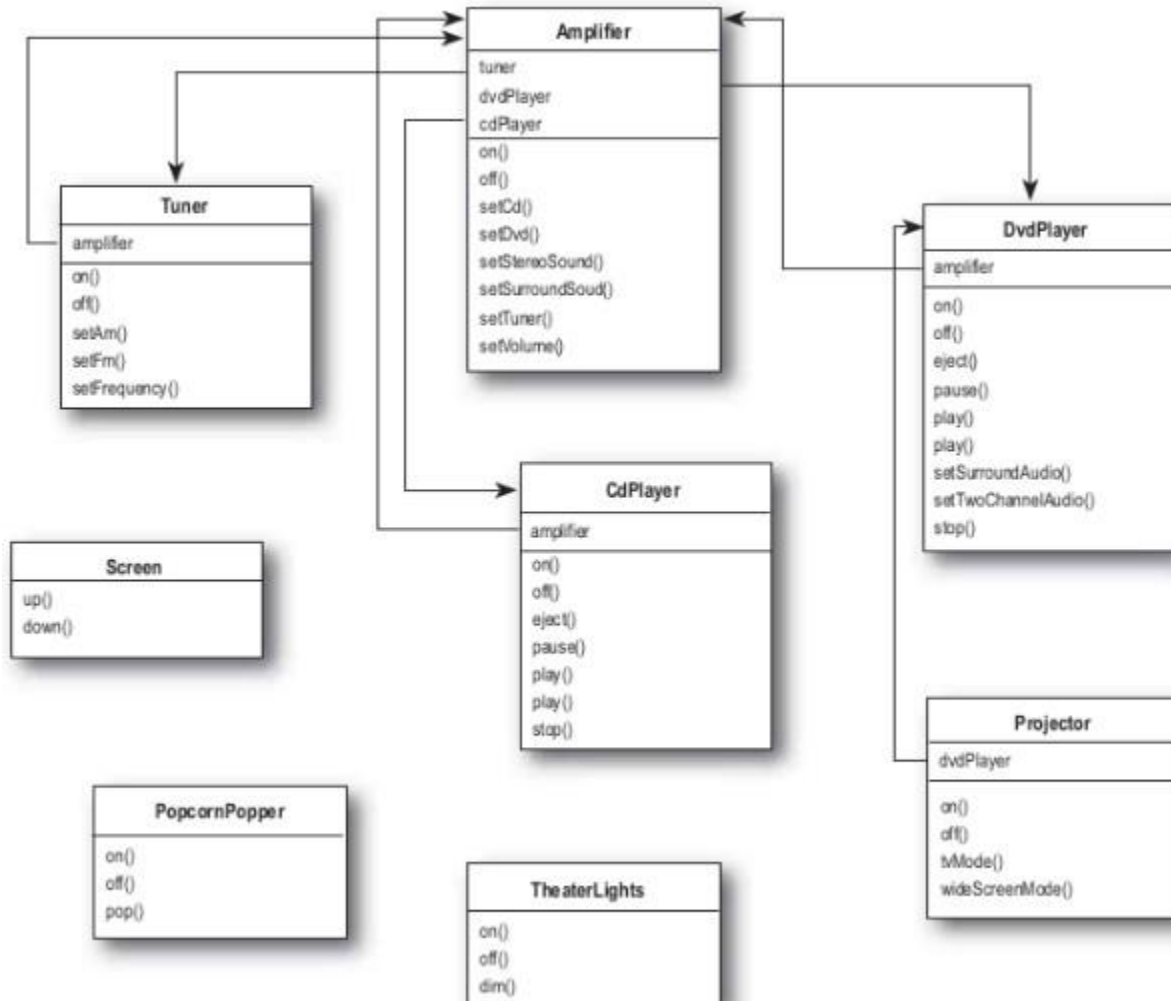
Note: The sample application talks about Apps and controllers. The real world implementation of these must be in form of separate application. We have considered all of them as classes of single application just for illustrating the facade pattern. If we actually implement such an application that will also be a facade but it will be a facade application rather than implementation of facade pattern.

In this example we have discussed about facade pattern. Often this pattern is confused with adapter pattern but in fact adapter pattern actually presents an altered interface of a system to the client and the original interface is not accessible. The facade pattern on the other hand provides a simpler interface of the subsystem. The original objects/system can still be accessed.

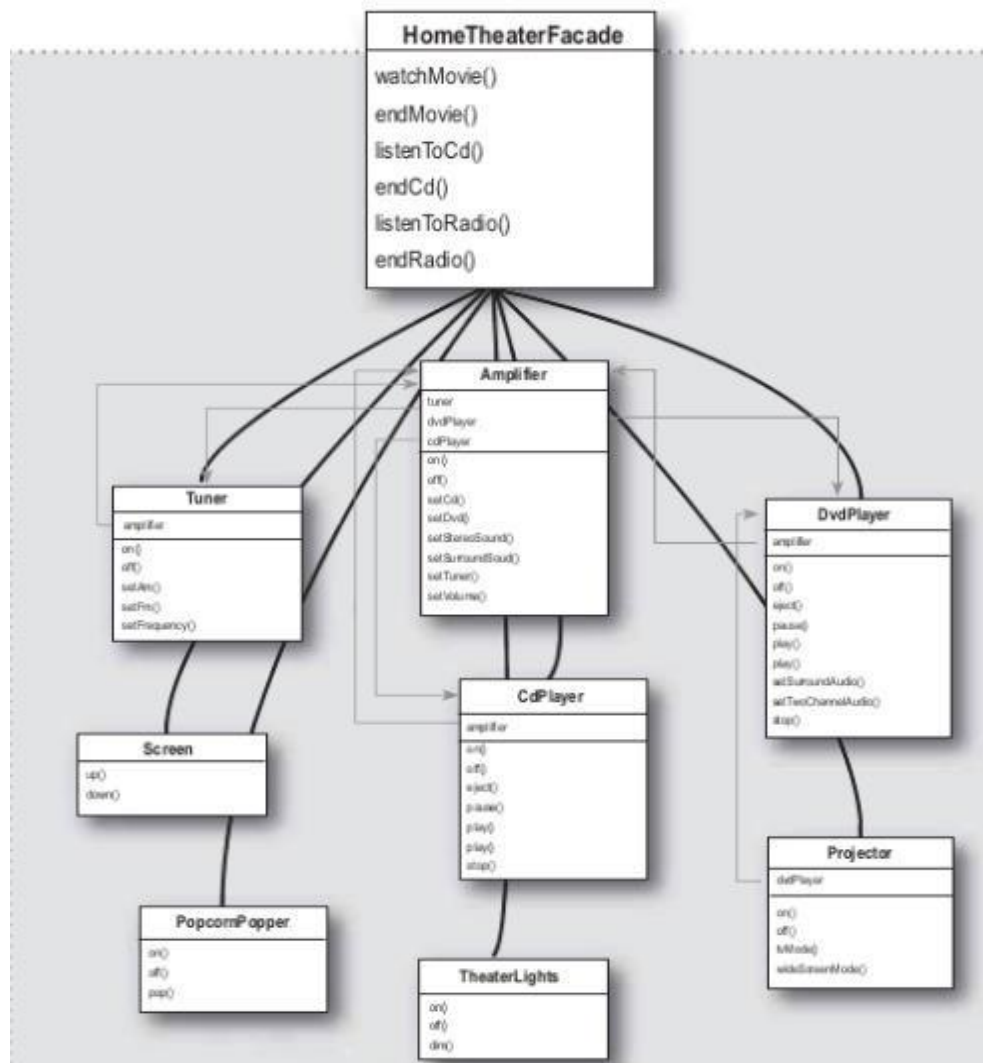
I hope this has been informative.

Another Real Life Example

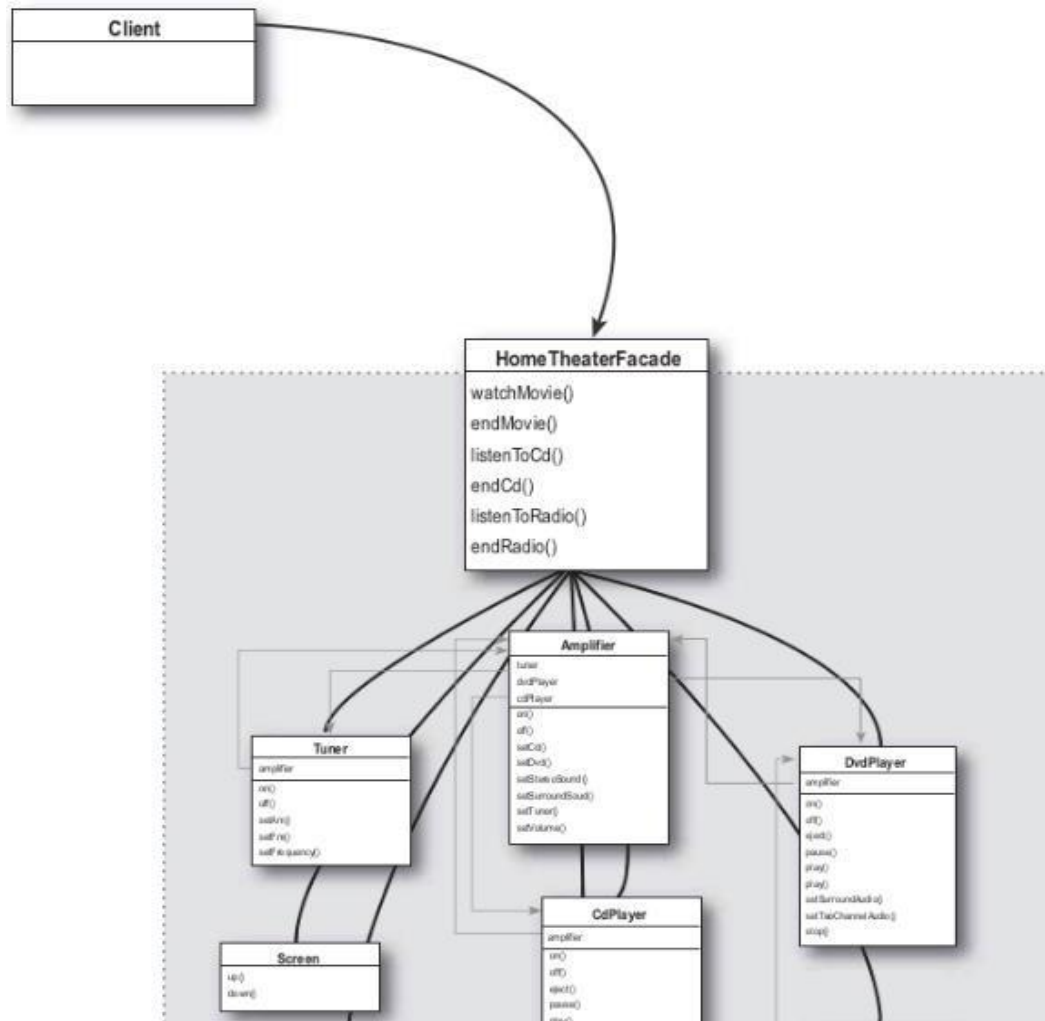
Many Components of Home Theater System



Home Theater System with Facade used



An example of how usage of Facade Pattern helps obey the principle of least knowledge



Singleton Pattern

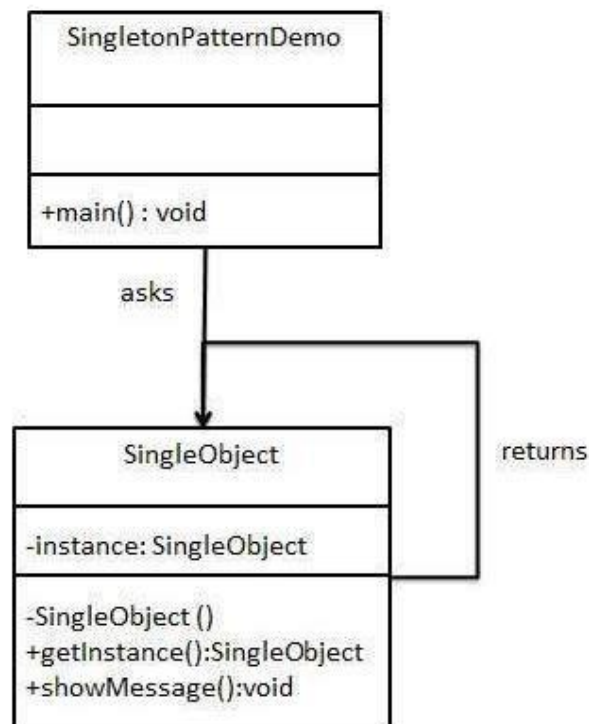
Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Implementation

We're going to create a *SingleObject* class. *SingleObject* class have its constructor as private and have a static instance of itself.

SingleObject class provides a static method to get its static instance to outside world. *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.



Step 1

Create a Singleton Class.

SingleObject.java

```
public class SingleObject {

    //create an object of SingleObject
    private static SingleObject instance = new SingleObject();

    //make the constructor private so that this class cannot be
    //instantiated
    private SingleObject(){}

    //Get the only object available
    public static SingleObject getInstance(){
        return instance;
    }

    public void showMessage(){
        System.out.println("Hello World!");
    }
}
```

Step 2

Get the only object from the singleton class.

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
    public static void main(String[] args) {

        //illegal construct
        //Compile Time Error: The constructor SingleObject() is not visible
    }
}
```

```
//SingleObject object = new SingleObject();

//Get the only object available
SingleObject object = SingleObject.getInstance();

//show the message
object.showMessage();
}
}
```

Step 3

Verify the output.

```
Hello World!
```

Example

A brief explanation of how to implement the Singleton pattern class in multithreading environment.

Often, a system only needs to create one instance of a class, and that instance will be accessed throughout the program. Examples would include objects needed for logging, communication, database access, etc.

So, if a system only needs one instance of a class, and that instance needs to be accessible in many different parts of a system, one control both instantiation and access by making that class a singleton.

A Singleton is the combination of two essential properties:

Ensure a class only has one instance.

Provide a global point of access to it.

You can find many examples on Singleton Pattern but this example will give you an insight about the basics on Singleton Pattern especially in the case of multithreading environment.

As stated above, a singleton is a class that can be instantiated once, and only once.

To achieve this we need to keep the following things in our mind.

1. Create a public Class (name SingletonSample).

```
public class SingletonSample  
{}
```

2. Define its constructor as private

```
private SingletonSample()  
{}
```

3. Create a private static instance of the class (name singleTonObject).

```
private volatile static SingletonSample singleTonObject
```

4. Now write a static method (name InstanceCreation) which will be used to create an instance of this class and return it to the calling method.

```
public static SingletonSample InstanceCreation()
{
    private static object lockingObject = new object();
    if(singleTonObject == null)
    {
        lock (lockingObject)
        {
            if(singleTonObject == null)
            {
                singleTonObject = new SingletonSample();
            }
        }
    }
    return singleTonObject;
}
```

Now we need to analyze this method in depth. We have created an instance of object named lockingObject, its role is to allow only one thread to access the code nested within the lock block at a time. So once a thread enters the lock area, other threads need to wait until the locking object gets released so, even if multiple threads try to access this method and want to create an object simultaneously, it's not possible. Further only if the static instance of the class is null, a new instance of the class is allowed to be created.

Hence only one thread can create an instance of this Class because once an instance of this class is created the condition of singleTonObject being null is always false and therefore rest all instances will contain value null.

5. Create a public method in this class, for example I am creating a method to display message (name DisplayMessage), you can perform your actual task over here.

```
public void DisplayMessage()
{
    Console.WriteLine("Hello Fastian My First Singleton Program");
}
```

6. Now we will create an another Class (name Program)

```
class Program
{}
```

7. Create an entry point to the above class by having a method name Main.

```
static void Main(string[] args)
{
    SingletonSample singleton = SingletonSample.InstanceCreation();
    singleton.DisplayMessage();
    Console.ReadLine();
}
```

Now we need to analyse this method in depth. As we have created an instance singleton of the class **SingletonSample** by calling the static method **SingletonSample.InstanceCreation()** a new object gets created and hence further calling the method **singleton.DisplayMessage()** would give an output "Hello Fastian My First Singleton Program".

Applicability & Examples

According to the definition the singleton pattern should be used when there must be exactly one instance of a class, and when it must be accessible to clients from a global access point. Here are some real situations where the singleton is used:

Example 1 - Logger Classes

The Singleton pattern is used in the design of logger classes. These classes are usually implemented as singletons, and provide a global logging access point in all the application components without being necessary to create an object each time a logging operation is performed.

Example 2 - Configuration Classes

The Singleton pattern is used to design the classes which provide the configuration settings for an application. By implementing configuration classes as Singleton not only that we provide a global access point, but we also keep the instance we use as a cache object. When the class is instantiated (or when a value is read) the singleton will keep the values in its internal structure. If the values are read from the database or from files this avoids the reloading the values each time the configuration parameters are used.

Example 3 - Accessing resources in shared mode

It can be used in the design of an application that needs to work with the serial port. Let's say that there are many classes in the application, working in a multi-threading environment, which needs to operate actions on the serial port. In this case a singleton with synchronized methods could be used to be used to manage all the operations on the serial port.