ABOUT ROHIT JOSHI

Rohit Joshi is a Senior Software Engineer from India. He is a Sun Certified Java Programmer and has worked on projects related to different domains. His expertise is in Core Java and J2EE technologies, but he also has good experience with front-end technologies like Javascript, JQuery, HTML5, and JQWidgets.

Facade Design Pattern

Posted by: Rohit Joshi in Core Java September 30th, 2015

This article is part of our Academy Course titled Java Design Patterns.

In this course you will delve into a vast number of Design Patterns and see how those are implemented and utilized in Java. You will understand the reasons why patterns are so important and learn when and how to apply each one of them. Check it out here!

Table Of Contents

1. Introduction

In this lesson, we will discuss about another structural pattern, i.e., the Facade Pattern. But before we dig into the details of it, let us discuss a problem which will be solved by this particular Pattern.

Your company is a product based company and it has launched a product in the market, named Schedule Server. It is a kind of server in itself, and it is used to manage jobs. The jobs could be any kind of jobs like sending a list of emails, sms, reading or writing files from a destination, or just simply transferring files from a source to the destination. The product is used by the developers to manage such kind of jobs and able to concentrate more towards their business goal. The server executes each job at their specified time and also manages all underline issues like concurrency issue and security by itself. As a developer, one just need to code only the relevant business requirements and a good amount of API calls is provided to schedule a job according to their needs.

Everything was going fine, until the clients started complaining about starting and stopping the process of the server. They said, although the server is working great, the initializing and the shutting down processes are very complex and they want an easy way to do that. The server

A complex interface to the client is already considered as a fault in the design of the current system. But fortunately or unfortunately, we cannot start the designing and the coding from scratch. We need a way to resolve this problem and make the interface easy to access.

A Facade Pattern can help us to resolve this design problem. But before that, let us see about the Facade Pattern.

2. What is the Facade Pattern

The Facade Pattern makes a complex interface easier to use, using a Facade class. The Facade Pattern provides a unified interface to a set of interface in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

The Facade unifies the complex low-level interfaces of a subsystem in-order to provide a simple way to access that interface. It just provides a layer to the complex interfaces of the sub-system which makes it easier to use.

The Facade do not encapsulate the subsystem classes or interfaces; it just provides a simplified interface to their functionality. A client can access these classes directly. It still exposes the full functionality of the system for the clients who may need it.

A Facade is not just only able to simplify an interface, but it also decouples a client from a subsystem. It adheres to the Principle of Least Knowledge, which avoids tight coupling between the client and the subsystem. This provides flexibility: suppose in the above problem, the company wants to add some more steps to start or stop the Schedule Server, that have their own different interfaces. If you coded your client code to the facade rather than the subsystem, your client code doesn't need to be change, just the facade required to be changed, that's would be delivered with a new version to the client.

Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the facade may have to do the work of its own to translate its interface to subsystem interfaces. Clients that use the facade don't have to access its subsystem objects directly.

Please note that, a Facade same as an Adapter can wrap multiple classes, but a facade is used to an interface to simplify the use of the complex interface, whereas, an adapter is used to convert the interface to an interface the client expects.

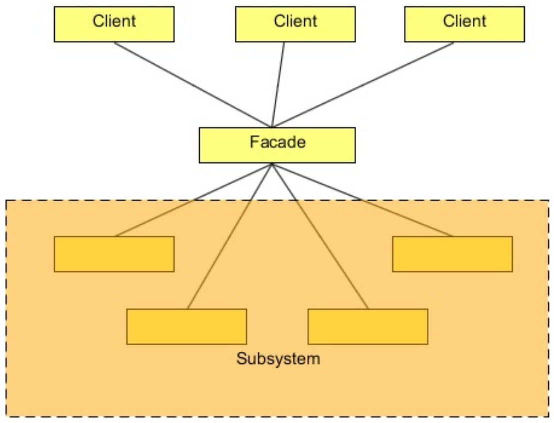

Figure 1

3. Solution to the problem

The problem faced by the clients in using the Schedule Server is the complexity brought by the server in order to start and stop its services. The client wants a simple way to do it. The following is the code that clients required to write to start and stop the server.

```java
ScheduleServer scheduleServer = new ScheduleServer();
```

To start the server, the client needs to create an object of the ScheduleServer class and then need to call the below methods in the sequence to start and initialize the server.

```java
scheduleServer.startBooting();
scheduleServer.readSystemConfigFile();
scheduleServer.init();
scheduleServer.initializeContext();
scheduleServer.initializeListeners();
scheduleServer.createSystemObjects();

System.out.println("Start working......");
System.out.println("After work done.........");
```

To stop the server, the client needs to call the following methods in the same sequence.

```java
scheduleServer.releaseProcesses();
scheduleServer.destory();
scheduleServer.destroySystemObjects();
scheduleServer.destoryListeners();
scheduleServer.destoryContext();
scheduleServer.shutdown();
```

To resolve this, we will create a facade class which will wrap a server object. This class will provide simple interfaces (methods) for the client. These interfaces internally will call the methods on the server object. Let us first see the code and then will discuss more about it.

```java
package com.javacodegeeks.patterns.facadepattern;

public class ScheduleServerFacade {

    private final ScheduleServer scheduleServer;

    public ScheduleServerFacade(ScheduleServer scheduleServer){
        this.scheduleServer = scheduleServer;
    }

    public void startServer(){

        scheduleServer.startBooting();
        scheduleServer.readSystemConfigFile();
        scheduleServer.init();
        scheduleServer.initializeContext();
        scheduleServer.initializeListeners();
        scheduleServer.createSystemObjects();
    }

    public void stopServer(){

        scheduleServer.releaseProcesses();
        scheduleServer.destory();
        scheduleServer.destroySystemObjects();
        scheduleServer.destoryListeners();
        scheduleServer.destoryContext();
        scheduleServer.shutdown();
    }

}
```

The above class

```
ScheduleServerFacade
```

is the facade class, which wraps a

```
ScheduleServer
```

object, it instantiates the server object through its constructor, and has two simple methods:

```
startServer()
```

and

```
stopServer()
```

. These methods internally perform the starting and the stopping of the server. The client is just needs to call these simple methods. Now, there is no need to call all the lifecycle and destroy methods, just the simple methods and the rest of the process will be performed by the facade class.

The code below shows how facade makes a complex interface simple to use.

```java
package com.javacodegeeks.patterns.facadepattern;

public class TestFacade {

    public static void main(String[] args) {

        ScheduleServer scheduleServer = new ScheduleServer();
        ScheduleServerFacade facadeServer = new ScheduleServerFacade(scheduleServer);
        facadeServer.startServer();

        System.out.println("Start working......");
        System.out.println("After work done.........");

        facadeServer.stopServer();
    }

}
```

Also, please note that, although the facade class has provided a simple interface to the complex subsystem, it has not encapsulated the subsystem. A client can still access the low-level interfaces of the subsystem. So, a facade provides an extra layer, a simple interface to the complex subsystem, but it does not completely hide the direct accessibility to the low-level interfaces of the complex subsystem.

4. Use of the Facade Pattern

Use the Facade Pattern, when:

1. You want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don’t need to customize it. A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.