HowToDoInJava

Facade Design Pattern

Last Updated: August 30, 2021 By: Lokesh Gupta Structural Patterns Design Patterns

Facade design pattern 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.

1. When to use facade pattern

Facade pattern is one of **structural design pattern** among other Gang of Four design patterns. The facade pattern is appropriate when we have a complex system that we want to expose to clients in a simplified way. Its purpose is to hide internal complexity behind a single interface that appears simple from the outside.

Facade also decouples the code that uses the system from the details of the subsystems, making it easier to modify the system later.

2. Real world facade examples

To understand the facade, let's take a very simple example of a desktop computer machine. When we have to start a computer, all we have to do is press the start button. We really do not care what all things go inside the computer hardware and software. It is an example of Facade pattern.

In Java programming, we must have connected to a database to fetch some data. We simply call the method `dataSource.getConnection()` to get the connection but internally a lot of things happen such as loading the driver, creating connection or fetching connection from pool, update stats and then return the connection reference to caller method. It is another example of Facade pattern in the programming world.

Similarly, we can find a lot of more examples which hide lots of internal complexities and provide simple to use interface to the programmer to work with the system. All such are facade examples.

3. Facade design pattern example

Let's write our own facade implementation for demo purpose. In this example, we are creating a report generator which has multiple steps to create any report. For example, it shall first create report header, footer, add data rows, format the report and then write the report in desirable format (pdf, html, etc).

Using **ReportGeneratorFacade**, we will hide all these steps and expose easy to use methods.

```
Report.java
```

```
public class Report {

  private ReportHeader header;
  private ReportData data;
  private ReportFooter footer;

  public ReportHeader getHeader() {
    return header;
  }
  public void setHeader(ReportHeader header) {
    System.out.println("Setting report header");
    this.header = header;
  }
  public ReportData getData() {
    return data;
  }
  public void setData(ReportData data) {
    System.out.println("Setting report data");
    this.data = data;
  }
  public ReportFooter getFooter() {
    return footer;
  }
  public void setFooter(ReportFooter footer) {
    System.out.println("Setting report footer");
    this.footer = footer;
  }
}
```

ReportHeader.java

```
public class ReportHeader {

}
```

ReportFooter.java

```
public class ReportFooter {

}
```

ReportData.java

```
public class ReportData {

}
```

ReportType.java

```
public enum ReportType
{
  PDF, HTML
}
```

ReportWriter.java

```
public class ReportWriter {

  public void writeHtmlReport(Report report, String location) {
    System.out.println("HTML Report written");
```

```
    //implementation
  }

  public void writePdfReport(Report report, String location) {
    System.out.println("Pdf Report written");

    //implementation
  }
}
```

ReportGeneratorFacade.java

```
import javax.activation.DataSource;

public class ReportGeneratorFacade
{
  public static void generateReport(ReportType type, DataSource dataSource, String location)
  {
    if(type == null || dataSource == null)
    {
      //throw some exception
    }
    //Create report
    Report report = new Report();

    report.setHeader(new ReportHeader());
    report.setFooter(new ReportFooter());

    //Get data from dataSource and set to ReportData object

    report.setData(new ReportData());

    //Write report
    ReportWriter writer = new ReportWriter();
    switch(type)
    {
      case HTML:
        writer.writeHtmlReport(report, location);
        break;

      case PDF:
        writer.writePdfReport(report, location);
        break;
    }
  }
}
```

Let's test our facade implementation.

Main.java

```
import com.howtodoinjava.facade.ReportGeneratorFacade;
import com.howtodoinjava.facade.ReportType;

public class Main
{
  public static void main(String[] args) throws Exception
  {
    ReportGeneratorFacade reportGeneratorFacade = new ReportGeneratorFacade();

    reportGeneratorFacade.generateReport(ReportType.HTML, null, null);

    reportGeneratorFacade.generateReport(ReportType.PDF, null, null);
  }
}
```

Program Output.

```
Console

 Setting report header
 Setting report footer
 Setting report data
 HTML Report written


 Setting report header
 Setting report footer
 Setting report data
 Pdf Report written
```

4. FAQs

4.1. Advantages of facade pattern

Remember facade does not reduce the complexity. It only hides it from external systems and clients. So the primary beneficiary of facade patterns are client applications and other systems only.

It provides a simple interface to clients i.e. instead of presenting complex subsystems, we present one simplified interface to clients. It can also help us to reduce the number of objects that a client needs to deal with.

4.2. Facade does not restrict access to sub-systems

A facade does not encapsulate the subsystem classes or interfaces. It just provides a simple interface (or layer) to make our life easier. We are free to expose any functionality of the subsystem or the whole subsystem itself.

It will just make the code look ugly, else it will work.

4.3. Facade pattern vs adapter pattern

In the adapter pattern, we try to alter an interface so that the clients is able to work with the system. Else the system will be difficult to use by the client (even not usable).

The facade pattern simplifies the interface. It presents the client with a simple interface to interact with (instead of a complex subsystem).

4.4. Facade pattern vs mediator pattern

In a mediator pattern implementation, subsystems are aware of the mediator. They talk to each other.

But in a facade, subsystems are not aware of the facade and the one-way communication is provided from facade to the subsystem(s).

4.5. Only one facade for a complex subsystem?

Not at all. We can create any number of facades for a particular complex subsystem. The idea is to make the system easier to use. It required creating N facades then please make them.

4.6. Challenges with facade design pattern

- Subsystems are connected with the facade layer. So, you need to take care of an additional layer of coding.
- When the internal structure of a subsystem changes, you need to incorporate the changes in the facade layer also.

Happy Learning !!

Recommended Reading:

1. Decorator Design Pattern in Java
2. Adapter Design Pattern in Java
3. Bridge Design Pattern
4. Composite Design Pattern
5. Flyweight Design Pattern
6. Proxy Design Pattern
7. Prototype design pattern in Java
8. Memento Design Pattern
9. Observer Design Pattern
0. Mediator Design Pattern

---

5 thoughts on “Facade Design Pattern”

Dazel

July 24, 2020 at 8:24 am

In challenges of Facade Design pattern, you mentioned upon the overhead of managing an extra layer of abstraction.
Changes to internal structure of the subsystem should follow the open-close principle right? Extension is acceptable but modifications reeks of bad design.

Reply

Lokesh Gupta

July 24, 2020 at 2:03 pm

That’s right.

Reply

Deepak

March 18, 2020 at 8:52 am

fix typo – interbnal

Reply

Lokesh Gupta

March 19, 2020 at 10:30 pm

Thanks, Deepak for reporting. Much appreciated !!

Reply

Yovel

March 11, 2020 at 7:28 pm

It looks exactly like the Factory pattern, what is the difference?

Reply

Leave a Comment

Name *

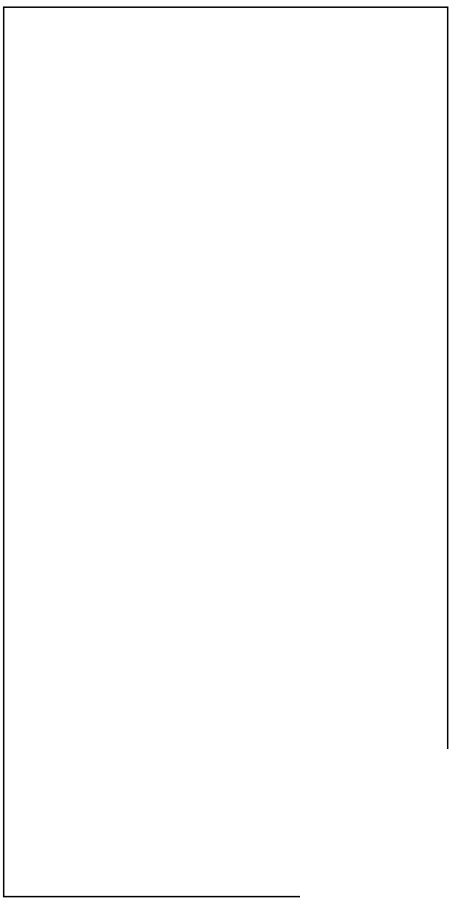
Email *

Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search …



HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

- REST API Tutorial