**Name:** Adeel Malik

**Student ID:** 80589156

**Façade vs Mediator**

Descriptions:

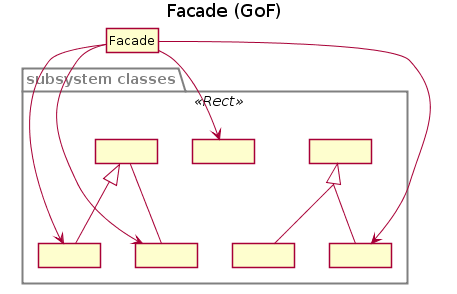
|  |  |
| --- | --- |
| **Façade** | **Mediator** |
| **Facade** links to various existing classes of a subsystem to add some typical functionality that simplifies use of the subsystem.  Façade only exposes the existing functionality from a different perspective  **Facade** gives you a simple interface which interacts on a set of coherent classes.  For example, a remote control for your house which controls all kind of equipment in your house would be a facade. You just interact with the remote control, and the remote control figures out which device should respond and what signal to send. | **Mediator** links to various colleagues that have to collaborate, so as to minimize the knowledge the colleagues have about each other. Minimizing knowledge has the side effect of reducing coupling between colleagues (they only know the mediator) and increasing their cohesion (they generally have less to worry about since they don't know about the bigger picture).  Mediator "adds" functionality because it combines different existing functionality to create a new one.  **Mediator** takes cares of communication between two objects, without the two objects need to have a reference to each other directly.  A real-life example is sending a letter, you post your letter and the postal service picks it up and makes sure that it will be delivered at the recipient. Without you telling them what route they should take. That is the mediator does. |
| Key notes:   1. A simple interface is required to access a complex system. 2. The abstractions and implementations of a subsystem are tightly coupled. 3. Need an entry point to each level of layered software. 4. System is very complex or difficult to understand. | Key notes:   1. Mediator pattern is useful when the communication logic between objects is complex, we can have a central point of communication that takes care of communication logic. 2. We should not use mediator pattern just to achieve lose-coupling because if the number of mediators will grow, then it will become hard to maintain them |

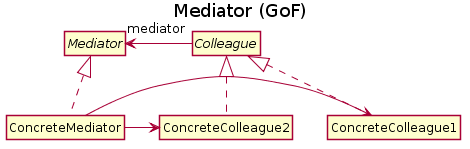
Similarities:

|  |  |
| --- | --- |
| **Façade** | **Mediator** |
| 1. In both patterns, the centralized class assumes responsibility for the complexity of dealing with the classes it is linked to. 2. Both promote loose coupling. 3. Both abstract the functionality of the sub system. | |

Differences:

|  |  |
| --- | --- |
| **Façade** | **Mediator** |
| 1. Facade does not add any functionality, Mediator does. 2. Subsystem components are not aware of Facade. 3. Mediator's colleagues are aware of Mediator and interact with it. | |





Example:

You have a logging system. From that logging system, you can either log to a file, to a socket, or to a database. Using the facade design pattern, you would "hide" all the relationships from existing functionality behind a single "interface" the one that the facade exposes.

Client code:

Logger logger = new Logger();

logger.initLogger("someLogger");

logger.debug("message");

The implementation may involve the interaction of many objects. But at the end, the functionality already exists. Probably the "debug" method is implemented as follows:

Implementation:

class Logger {

private LoggerImpl internalLogger;

private LoggerManager manager;

public void initLogger( String loggerName ) {

this.internalLogger = manager.getLogger( loggerName );

}

public void debug( String message ) {

this.internalLogger.debug( message );

}

}

The functionality already exists. The facade only hides it. In this hypothetical case, the LoggerManager handles the creation of the correct logger, and the LoggerImpl is a package private object that has the "debug" method. This way the Facade is not adding functionality he is just delegating to some existing objects

In the other hand the mediator add the new functionality by combining different objects.

Same Client code:

Logger logger = new Logger();

logger.initLogger("someLogger");

logger.debug("message");

Implementation:

class Logger {

private java.io.PrintStream out;

private java.net.Socket client;

private java.sql.Connection dbConnection;

private String loggerName;

public void initLogger( String loggerName ) {

this.loggerName = loggerName;

if ( loggerName == "someLogger" ) {

out = new PrintStream( new File("app.log"));

} else if ( loggerName == "serverLog" ) {

client = new Socket("127.0.0.1", 1234 );

} else if( loggerName == "dblog") {

dbConnection = Class.forName()... .

}

}

public void debug( String message ) {

if ( loggerName == "someLogger" ) {

out.println( message );

} else if ( loggerName == "serverLog" ) {

ObjectOutputStrewam oos =

new ObjectOutputStrewam( client.getOutputStream());

oos.writeObject( message );

} else if( loggerName == "dblog") {

Pstmt pstmt = dbConnection.prepareStatment( LOG\_SQL );

pstmt.setParameter(1, message );

pstmt.executeUpdate();

dbConnection.commit();

}

}

}

In this code, the mediator is the one that contains the business logic to create the appropriate "channel" to log and also to make the log into that channel and "creating" the functionality.

Of course, there are better ways to implement this using polymorphism, but the point here is to show how the mediator "adds" new functionality by combining existing functionality (the sample didn't show much detais) but imagine the mediator, read from the database the remote host where to log, then creates a client and finally write to that client print stream the log message. This way the mediator would be "mediating" between the different objects.

Finally, the facade is a structural pattern, that describes the composition of the objects, while the mediator is a behavioral pattern, that describes the way the objects interact.

References:

<http://stackoverflow.com/>

<http://www.eli.sdsu.edu>