Lesson 9 Façade and Memento

The Façade Pattern

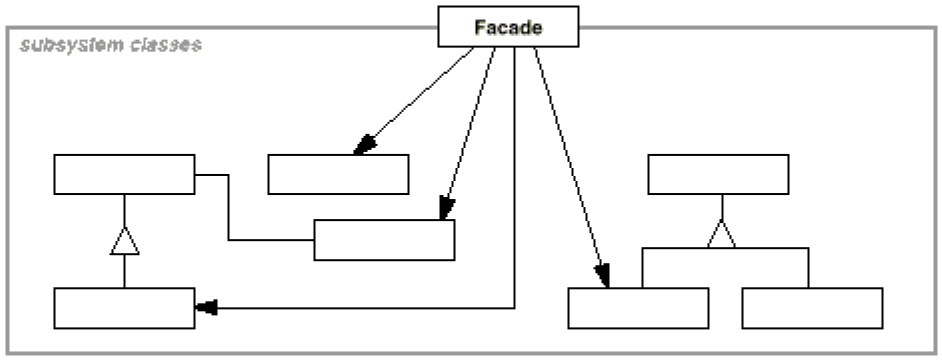
1. Intent

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.

2. Motivation

a. Structuring a system into subsystems helps reduce complexity.
b. Subsystems are groups of classes, and/or other subsystems.
c. The interface exposed by the classes in a subsystem or set of subsystems can become quite complex.
d. One way to reduce this complexity is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.

3. Structure



4. Consequences

a. It hides the implementation of the subsystem from clients, making the subsystem easier to use.
b. It promotes weak coupling between the subsystem and its clients. This allows you to change the classes that comprise the subsystem without affecting the clients.
c. It reduces compilation dependencies in large software systems.
d. However, it does not prevent sophisticated clients from accessing the underlying classes.
e. And Facade does not add any functionality, it just simplifies interfaces.

5. How to implement the Façade pattern?

Let’s see an example “LoanManager” class

```java
public class LoanManagerBefore {
      public void Process(LoanApplication application) {
            boolean personalDetailsAreValid = new PersonalDetailVerifier()
                        .verifyDetails(application);

            boolean professionalDetailsAreValid = new
ProfessionalDetailVerifier()
                        .verifyDetails(application);

            boolean creditHistoryVerified = new CreditHistoryVerifier()
                        .verify(application);

            boolean isPreferredDeveloper = new DeveloperVerifier()

      .isPreferredDeveloper(application.getDeveloperName());

            boolean isProjectApproved = isPreferredDeveloper ? true
                        : new ProjectVerifier()
.isBlackListedProject(application.getProjectName());

            boolean isEligible = personalDetailsAreValid
                        && professionalDetailsAreValid &&
creditHistoryVerified   && isProjectApproved;

            if (isEligible) {
                  NotificationService notificationService = new
NotificationService();
                  notificationService.notify(application);
            }
      }
      }
```

LoanManager refactored using the Façade pattern

```
public class LoanManagerRefactored {
      public void Process(LoanApplication application) {
            ILoanApplicationVerifier loanApplicationVerifier = new
            LoanApplicationVerifier();

            boolean isEligible =
            loanApplicationVerifier.isEligible(application);

            if (isEligible) {
                  NotificationService notificationService = new
                  NotificationService();
                  notificationService.notify(application);
            }
      }
}
```

Lab 9-1

Refactor the demo class with the Façade pattern to simplify the client code.

```
public class JDBCdemo {
      public static void main(String[] arg) {
            Connection conn = null;
            PreparedStatement prep = null;
            CallableStatement call = null;
            ResultSet rset = null;
            try {
                  Class.forName("<driver>").newInstance();
                  conn = DriverManager.getConnection("<database>");
                  String sql = "SELECT * FROM <table> WHERE <column name> =
?";

                  prep = conn.prepareStatement(sql);
                  prep.setString(1, "<column value>");
                  rset = prep.executeQuery();
                  if (rset.next()) {
                        System.out.println(rset.getString("<column name"));
                  }
                  sql = "{call <stored procedure>( ?, ? )}";
                  call = conn.prepareCall(sql);
                  call.setInt(1, 1972);
                  call.registerOutParameter(2, java.sql.Types.INTEGER);
                  call.execute();
```

```
                System.out.println(call.getInt(2));
            } catch (SQLException e) {
                e.printStackTrace();
            } catch (InstantiationException e) {
                e.printStackTrace();
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } finally {
                if (rset != null) {
                    try {
                        rset.close();
                    } catch (SQLException ex) {
                        ex.printStackTrace();
                    }
                }
                if (prep != null) {
                    try {
                        prep.close();
                    } catch (SQLException ex) {
                        ex.printStackTrace();
                    }
                }
                if (call != null) {
                    try {
                        call.close();
                    } catch (SQLException ex) {
                        ex.printStackTrace();
                    }
                }
                if (conn != null) {
                    try {
                        conn.close();
                    } catch (SQLException ex) {
                        ex.printStackTrace();
                    }
                }
            }
        }
    }
```

The Memento Pattern

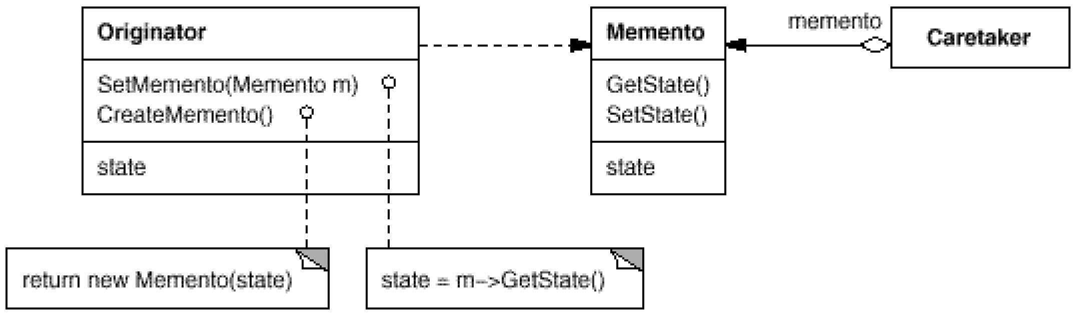
1. Intent

   Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

2. Motivation

   a. Sometimes it's necessary to record the internal state of an object. This is required when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors.
   b. You must save state information somewhere so that you can restore objects to their previous states.

3. Structure



4. Participants

   a. Memento
      - stores internal state of the Originator object. The memento may store as much or as little of the

originator's internal state as necessary at its originator's discretion.
- protects against access by objects other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento—it can only pass the memento to other objects. Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.

b. Originator
   - creates a memento containing a snapshot of its current internal state.
   - uses the memento to restore its internal state.

c. Caretaker
   - is responsible for the memento's safekeeping.
   - never operates on or examines the contents of a memento.

5. Applicability

Use the Memento pattern when

a. A snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, and
b. A direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

6. Implementation

a. Memento (a previous state or ‘snapshot’)

```
class Memento {
	private String state;

	public Memento(String state) {
		this.state = state;
	}

	String getState() {
		return state;
	}
}
```

b. Originator (its state gets externalized to be stored in a Memento object)

```
public class Originator {
	private String externalizedState;

	String getExternalizedState() {
		return externalizedState;
	}

	void setExternalizedState(String externalizedState) {
		this.externalizedState = externalizedState;
	}

	public Memento saveStateToMemento() {
		return new Memento(externalizedState);
	}

	public void getStateFromMemento(Memento Memento) {
		externalizedState = Memento.getState();
	}
}
```

c. CareTaker class (keeps a collection of previous states in case client needs to restore them or undo something)

```
class CareTaker {
	private List<Memento> mementoList = new ArrayList<Memento>();

	void add(Memento state) {
		mementoList.add(state);
	}

	Memento get(int index) {
		return mementoList.get(index);
	}
}
```

Lab 9-2

Develop a GUI that edits user information. You will need at least 3 buttons – load (from a database), undo, save. When you start, you use the ‘load’ button to read an existing profile for the user, if it does exist. If not, then create a new profile for the user. At any time during the edit work, you can always click on ‘undo’ to restore from errors, like what the MS Word allows you to do. Once you are done with the user profile, click ‘save’ button to store it (in the database).