Step-by-Step Evolution of a Library Management System using Design Patterns

Affan Rauf

October 13, 2024

1 Introduction

In this tutorial, we will go through the evolution of a Library Management System (LMS) in Java. The design process utilizes various software engineering principles and design patterns. The system starts with basic Data Access Objects (DAO) and progressively incorporates the Facade Pattern, the Abstract Factory Pattern, and configuration-based dependency management. At each step, we evaluate the benefits of moving to the next iteration and discuss the design decisions made.

2 Basic Data Access Objects (DAOs)

Initially, we define DAOs for the `Book` and `Member` entities. These classes interact directly with the MySQL database.

```java
public class BookDAO {
    public void addBook(Book book) {
        String query = "INSERT INTO books (title, author) VALUES (?, ?)";
        PreparedStatement stmt = connection.prepareStatement(query);
        stmt.setString(1, book.getTitle());
        stmt.setString(2, book.getAuthor());

        int rowsAffected = stmt.executeUpdate();
        return rowsAffected > 0;
    }

    public Book getBookById(int id) {
        String query = "SELECT * FROM books WHERE id = ?";
        PreparedStatement stmt = connection.prepareStatement(query);
        stmt.setInt(1, bookId);
        ResultSet rs = stmt.executeQuery();

        if (rs.next()) {
            return new Book(rs.getInt("id"), rs.getString("title"), rs.getString("
                author"));
        }
        return null;
        return book;
    }
}

public class MemberDAO {
    public void addMember(Member member) {
        // Code to add a member to MySQL database
    }

    public Member getMemberById(int id) {
        // Code to fetch a member from MySQL database
        return member;
    }
}
```

Listing 1: BookDAO and MemberDAO for MySQL

3 Business Logic Layer with Concrete DAOs

Initially, the Business Logic Layer (BLL) classes like `BookBO` and `MemberBO` directly interacted with specific DAO implementations. This setup meant the BLL was tightly coupled to a MySQL-based data source, limiting flexibility and making it difficult to switch data sources without modifying the BLL classes themselves.

```
public class BookBO {
    private BookDAO bookDAO;

    public BookBO() {
        this.bookDAO = new BookDAO(); // Direct dependency on MySQL implementation
    }

    public void addBook(Book book) {
        bookDAO.addBook(book);
    }

    public Book getBook(int id) {
        return bookDAO.getBookById(id);
    }
}

public class MemberBO {
    private MemberDAO memberDAO;

    public MemberBO() {
        this.memberDAO = new MemberDAO(); // Direct dependency on MySQL implementation
    }

    public void addMember(Member member) {
        memberDAO.addMember(member);
    }

    public Member getMember(int id) {
        return memberDAO.getMemberById(id);
    }
}
```

Listing 2: BookBO and MemberBO Coupled with Concrete DAOs

This tight coupling makes it challenging to switch data storage solutions without altering multiple BLL classes. The design also violates the **Dependency Inversion Principle**, as the BLL directly depends on concrete implementations rather than abstractions.

4 Introducing Interfaces for DAOs

To support multiple data sources, we introduce `IBookDAO` and `IMemberDAO` interfaces, making the DAOs more modular and adaptable to new implementations.

```java
public interface IBookDAO {
    void addBook(Book book);
    Book getBookById(int id);
}

public interface IMemberDAO {
    void addMember(Member member);
    Member getMemberById(int id);
}
```

Listing 3: DAO Interfaces

By depending on these interfaces, we enable a more flexible business layer that doesn't rely on specific DAO implementations.

5 Decoupling Business Logic from Concrete DAOs using Interfaces and Dependency Injection

By introducing `IBookDAO` and `IMemberDAO` interfaces, we decouple the BLL from specific DAO implementations. The BLL classes are now flexible, only depending on DAO interfaces.

```java
public class BookBO {
    private IBookDAO bookDAO; // depend on abstraction

    public BookBO(IBookDAO bookDAO) {
        this.bookDAO = bookDAO; // dependency injection
    }

    public void addBook(Book book) {
        bookDAO.addBook(book);
    }

    public Book getBook(int id) {
        return bookDAO.getBookById(id);
    }
}

public class MemberBO {
    private IMemberDAO memberDAO; // depend on abstraction

    public MemberBO(IMemberDAO memberDAO) {
        this.memberDAO = memberDAO; // dependency injection
    }

    public void addMember(Member member) {
        memberDAO.addMember(member);
    }

    public Member getMember(int id) {
        return memberDAO.getMemberById(id);
    }
}
```

Listing 4: BookBO and MemberBO Decoupled with DAO Interfaces and Dependency Injection

This version of the BLL classes receives the DAO interfaces through **dependency injection**, which allows `BookBO` and `MemberBO` to be agnostic of the underlying data source implementation. As a result, switching between data sources (e.g., from MySQL to file-based) is as simple as injecting a different DAO implementation, without altering the BLL code.

5.1 Benefits of Decoupling with Interfaces and Dependency Injection

By combining interfaces with dependency injection, the design achieves the following benefits:

- **Enhanced Flexibility**: The BLL can easily switch between different DAO implementations, supporting various data sources without any changes in business logic.
- **Compliance with SOLID Principles**: Specifically, this design adheres to the **Dependency Inversion Principle** by ensuring that high-level modules depend on abstractions rather than concrete implementations.

By implementing these principles, the system becomes more maintainable, scalable, and adaptable to future changes.

6 File-based DAO Implementations

To support a file-based backend, we create file-based implementations of DAOs. These classes manage data in text files.

```
public class FileBookDAO implements IBookDAO {
    private String filePath;

    public FileBookDAO(String filePath) {
        this.filePath = filePath;
    }

    public void addBook(Book book) {
        BufferedWriter writer = new BufferedWriter(new FileWriter(filePath, true));
        String bookRecord = book.getId() + "," + book.getTitle() + "," + book.getAuthor();
        writer.write(bookRecord);
        writer.newLine();
        writer.close();
        return true;
    }

    public Book getBookById(int id) {
         BufferedReader reader = new BufferedReader(new FileReader(filePath));
        String line;
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split(",");
            int id = Integer.parseInt(parts[0]);
            if (id == bookId) {
                reader.close();
                return new Book(id, parts[1], parts[2]);  // id, title, author
```

```java
            }
        }
        reader.close();
        return null;
    }
}

public class FileMemberDAO implements IMemberDAO {
    private String filePath;

    public FileMemberDAO(String filePath) {
        this.filePath = filePath;
    }

    public void addMember(Member member) {
        // Code to write member details to a file
    }

    public Member getMemberById(int id) {
        // Code to read member details from a file
        return member;
    }
}
```

Listing 5: File-based BookDAO and MemberDAO

7 Implementing a Facade Pattern to Simplify Data Access

As the number of DAO interfaces grows, managing them in the Business Logic Layer (BLL) can become complex. To simplify this, we introduce the Facade Pattern, which provides a unified interface for interacting with all DAOs. The facade will streamline access to the various DAOs, allowing the BLL to use a single point of interaction.

7.1 Designing IDALFacade

The IDALFacade interface is designed to extend all existing DAO interfaces (IBookDAO, IMemberDAO, etc.). This approach ensures that the facade can directly expose all necessary methods from each DAO, making it easier to interact with multiple DAOs through a single interface.

```java
public interface IDALFacade extends IBookDAO, IMemberDAO {
    // Additional facade-specific methods can be added if needed
}
```

Listing 6: IDALFacade Extending DAO Interfaces

7.2 Implementing the DALFacade Class

The DALFacade class implements IDALFacade and provides concrete implementations for all methods declared in the DAO interfaces. It holds references to the individual DAOs and delegates the task to the appropriate DAO implementation.

```java
public class DALFacade implements IDALFacade {
    private IBookDAO bookDAO;
    private IMemberDAO memberDAO;

    public DALFacade(IBookDAO bookDAO, IMemberDAO memberDAO) {
        this.bookDAO = bookDAO;
        this.memberDAO = memberDAO;
    }

    @Override
    public void addBook(Book book) {
        bookDAO.addBook(book); // Delegates to bookDAO
    }

    @Override
    public Book getBookById(int id) {
        return bookDAO.getBookById(id); // Delegates to bookDAO
    }

    @Override
    public void addMember(Member member) {
        memberDAO.addMember(member); // Delegates to memberDAO
    }

    @Override
    public Member getMemberById(int id) {
        return memberDAO.getMemberById(id); // Delegates to memberDAO
    }

    // Additional methods can delegate tasks to other DAOs as needed
}
```

Listing 7: DALFacade Delegates to Specific DAOs

7.3 Benefits of Using the Facade Pattern

The DALFacade class simplifies the interactions between the BLL and the DAOs by providing a single access point. This design offers several advantages:

- **Simplified Interface**: The BLL interacts with the DALFacade rather than multiple DAOs, reducing complexity.
- **Easier Maintenance**: Changes to the underlying DAO implementations or the addition of new DAOs can be managed within the facade without affecting the BLL.
- **Enhanced Flexibility**: The DALFacade allows switching between different DAO implementations by simply updating the DAO instances it uses, which is particularly beneficial when paired with the Abstract Factory for DAO creation.

By implementing the Facade Pattern, the application becomes easier to understand and maintain as it evolves, as it promotes a clean separation between the business logic and the data access layer.

7.4 Impact of the Facade on Business Objects

Before implementing the Facade Pattern, the Business Objects (BOs) such as BookBO and MemberBO each required direct references to individual DAO interfaces, resulting in multiple dependencies. This setup made the BLL more complex, as each BO had to be aware of and manage the relevant DAO instances directly.

```java
public class BookBO {
    private IBookDAO bookDAO;

    public BookBO(IBookDAO bookDAO) {
        this.bookDAO = bookDAO;
    }

    // Business logic methods
}

public class MemberBO {
    private IMemberDAO memberDAO;

    public MemberBO(IMemberDAO memberDAO) {
        this.memberDAO = memberDAO;
    }

    // Business logic methods
}
```

Listing 8: BookBO and MemberBO with Direct DAO Dependencies

With the introduction of the `DALFacade`, the BOs no longer need to maintain direct dependencies on multiple DAOs. Instead, they depend on the `IDALFacade` interface, which provides access to all required DAO operations through a single, unified interface. This significantly simplifies the BOs, as they can now interact with the `DALFacade` without concerning themselves with the underlying DAO structure.

```java
public class BookBO {
    private IDALFacade dalFacade;

    public BookBO(IDALFacade dalFacade) {
        this.dalFacade = dalFacade;
    }

    public void addBook(Book book) {
        dalFacade.addBook(book); // Delegated through facade
    }

    public Book getBook(int id) {
        return dalFacade.getBookById(id); // Delegated through facade
    }
}

public class MemberBO {
    private IDALFacade dalFacade;

    public MemberBO(IDALFacade dalFacade) {
        this.dalFacade = dalFacade;
    }

    public void addMember(Member member) {
        dalFacade.addMember(member); // Delegated through facade
    }

    public Member getMember(int id) {
        return dalFacade.getMemberById(id); // Delegated through facade
    }
}
```

Listing 9: BookBO and MemberBO Using the DALFacade

7.5 Benefits of Using a Facade in Business Objects

Using the Facade Pattern in BOs provides several advantages:

- **Reduced Dependencies**: BOs now depend on a single `IDALFacade` interface, reducing the need for multiple DAO dependencies.
- **Simplified Code**: The BOs are cleaner and easier to maintain, as they no longer need to manage individual DAOs directly.
- **Increased Flexibility**: If the underlying data access implementation changes, updates can be made within the `DALFacade` without modifying the BOs.

By introducing the Facade Pattern, we achieve a cleaner, more cohesive BLL, allowing BOs to focus solely on business logic without getting entangled in data access concerns. This approach enhances modularity, maintainability, and scalability as the system evolves.

8 Abstract Factory Pattern for Dynamic Switching

To dynamically switch between MySQL and file-based DAOs based on configuration, we implement the Abstract Factory Pattern. This pattern facilitates the creation of related objects without specifying their concrete classes.

8.1 IDAOFactory Interface

The `IDAOFactory` interface defines methods for creating DAOs.

```java
public interface IDAOFactory {
    IBookDAO createBookDAO();
    IMemberDAO createMemberDAO();
}
```

Listing 10: IDAOFactory Interface

8.2 AbstractDAOFactory Abstract Class

```java
public abstract class AbstractDAOFactory implements IDAOFactory {

}
```

Listing 11: AbstractDAOFactory Abstract Class

8.3 Concrete Factory Implementations

Two concrete factories are created for MySQL and file-based DAOs.

```java
public class MySQLDAOFactory extends AbstractDAOFactory {
    public IBookDAO createBookDAO() {
        return new BookDAO();
    }

    public IMemberDAO createMemberDAO() {
        return new MemberDAO();
    }
}

public class FileDAOFactory extends AbstractDAOFactory {
    public IBookDAO createBookDAO() {
        return new FileBookDAO("books.txt");
    }

    public IMemberDAO createMemberDAO() {
        return new FileMemberDAO("members.txt");
    }
}
```

Listing 12: MySQL and File-based DAL Factories

9 Introducing the Abstract Factory for Dynamic DAL Instantiation with a Configuration File

As the system grows, flexibility becomes crucial in allowing the system to switch between different data access layers (DALs) without modifying the core business logic. We achieve this flexibility using the **Abstract Factory** design pattern combined with Java's ability to load configuration properties from a file.

9.1 Reading the Factory Class from a Properties File

To make the system more configurable, we store the name of the desired factory class in a `config.properties` file. This file can be updated easily without requiring any changes to the core code and its recompilation. Java's `Properties` class allows us to read from this configuration file.

Here is the updated implementation of the `AbstractDAOFactory` class, which reads the factory class name from a `config.properties` file and uses reflection to instantiate the correct factory at runtime.

```java
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

public abstract class AbstractDAOFactory implements IDAOFactory {

    private static IDAOFactory instance = null;

    public static final IDAOFactory getInstance() {
        if (instance == null) {
            String factoryClassName = null;
            try (FileInputStream input = new FileInputStream("config.properties"))
                 {
```

```java
                Properties prop = new Properties();
                prop.load(input);
                factoryClassName = prop.getProperty("dal.factory");

                Class<?> clazz = Class.forName(factoryClassName); // Load class by name
                instance = (IDAOFactory) clazz.getDeclaredConstructor().newInstance(); // Instantiate class
            } catch (IOException e) {
                e.printStackTrace();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return instance;
    }
}
```

Listing 13: AbstractDAOFactory with Config File

9.2 How it Works

- **Loading the Properties File**: The system reads the `config.properties` file using a `FileInputStream`. The `Properties` object loads the file and retrieves the value of the `dal.factory` key, which specifies the fully qualified class name of the desired factory.

- **Reflection for Class Loading**: Once the class name is read from the properties file, the factory is instantiated using Java Reflection (`Class.forName()` and `newInstance()`).

- **Handling Exceptions**: The code includes error handling for both file reading and class instantiation, ensuring that the system reports errors if the properties file is not found or the factory class cannot be loaded.

9.3 Configuring the Factory in `config.properties`

To define the factory class in the configuration file, we use the following format in `config.properties`:

```
# config.properties
dal.factory=com.example.MySQLDAOFactory
```

This entry tells the system to load the MySQL-specific DAO factory. To switch to a file-based DAL, simply change the entry to:

```
dal.factory=com.example.FileDAOFactory
```

9.4 Using the Factory in the Main Method

As before, the abstract factory is used in the `main` method to inject the correct DAOs into the business objects. Here's the updated flow for loading the factory class from the properties file:

```java
public class Main {
    public static void main(String[] args) {
        // Get DAO instances from the factory
        IBookDAO bookDAO = AbstractDAOFactory.getInstance().getBookDAO();
        IMemberDAO memberDAO = AbstractDAOFactory.getInstance().getMemberDAO();

        // Create the facade with the DAOs
        IDALFacade facade = new DALFacade(bookDAO, memberDAO);

        // Inject the facade into the business objects
        BookBO bookBO = new BookBO(facade);
        MemberBO memberBO = new MemberBO(facade);

        // Now business objects can use the facade for data access
        bookBO.addBook(new Book(1, "Clean Code", "Robert C. Martin"));
        Member member = memberBO.getMember(101);
        System.out.println("Member Name: " + member.getName());
    }
}
```

Listing 14: Factory Usage in Main Method

This updated approach now allows you to switch the DAL without modifying the code. Simply update the `config.properties` file to choose between different factory implementations (MySQL, file-based, etc.).

9.5 Benefits of Configuration-Based Abstract Factory

- **Easy Configuration**: Changing the data access layer is as simple as updating a configuration file. No code changes are needed.

- **Decoupling**: The business logic layer remains completely decoupled from the specific details of the data access layer.

- **Scalability**: New data access layers can be added in the future by implementing new factories and updating the `config.properties` file.

10 Conclusion

This step-by-step evolution of the Library Management System highlights how design patterns like Facade and Abstract Factory improve flexibility and adaptability. The system can now easily switch between MySQL and file-based implementations, or incorporate new data sources, without modifying the business logic.

Previously, we discussed how the BOs depended directly on the individual DAO interfaces (such as `IBookDAO` and `IMemberDAO`), leading to tight coupling between the Business Logic Layer (BLL) and the specific data access implementations. By introducing the Abstract Factory Pattern, we can decouple the BOs from specific DAO implementations, allowing them to depend on an abstract factory that creates the necessary DAOs.

This change provides the flexibility to switch between different data access mechanisms (e.g., MySQL-based or file-based) without modifying the BOs. The factory selection is made at runtime based on a configuration file, and the appropriate DAOs are injected into the BOs through the factory.