Design Patterns

Table of Contents

What is a Design Pattern?

Design patterns are well-proved solution for solving the specific problem/task.
After multiple years of software engineering an understanding has been developed to adopt certain practices in order to achieve easily maintainable, retestable, reusable, low error prone code. It is like following a "Standard Operating Procedure" while building a software system.

Advantages of Design Patterns

1. They provide the solutions that help to define the system architecture.
2. They capture the software engineering experiences.
3. They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
4. Design patterns don't guarantee an absolute solution to a problem.
5. They provide clarity to the system architecture and the possibility of building a better system

When should we use the design patterns?

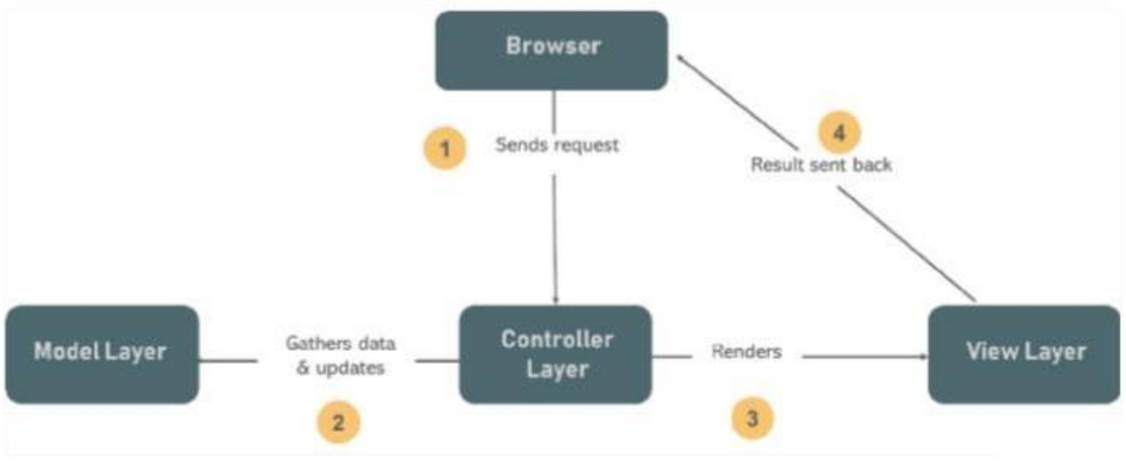
We must use the design patterns during the analysis and requirement phase of SDLC (Software Development Life Cycle).
Examples: We are going to learn about:

1. MVC Design Pattern
2. DAO Design Pattern

MVC Design Pattern

MVC design Pattern is one of the most implemented software design patterns in J2EE projects. It consists of three layers:

1. The **Model** Layer - Represents the business layer of the application
2. The **View** Layer - Defines the presentation of the application
3. The **Controller** Layer - Manages the flow of the application



Now J2EE has provided us with **3 technologies** that help us implement MVC design pattern. These are:

1. Servlets
2. JSP
3. JDBC

Servlets

- Servlets are meant to process **business logic.**
- It receives the data, processes it and produces the output.
- Servlets acts as **Controller** component
- The latest version of Servlet is 5.0

JSP

- JSP stands for **Java Server Pages**
- JSP is meant for presentational logic.
- When we want to display something to the end user, we use JSP.
- JSP acts as a **View** Component.
- The latest version of JSP is 3.0

JDBC

- JDBC stands for **Java Database Connectivity**
- JDBC connects a Java application to the database.
- It helps the java application to communicate with the database and implement CRUD operations.
- It acts as a **Model** component
- The latest version of JDBC is 4.3

DAO Design Pattern

This pattern is used to separate low level data accessing API or operations from high level business services or we can say that DAO pattern creates a persistence layer for service layer to use. Following are the participants in Data Access Object Pattern.

1. **Data Access Object Interface -** This **interface** defines the standard operations to be performed on a model object(s).
2. **Data Access Object concrete class -** This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.
3. **Model Object or Value Object -** This object is simple **POJO** containing get/set methods to store data retrieved using DAO class.

Implementation

We are going to create a Student object acting as a Model or Value **Pojo.StudentDao** is Data Access Object Interface.StudentDaoImpl is concrete class implementing Data Access Object Interface. DaoPatternDemo, our demo class, will use StudentDao to demonstrate the use of Data Access Object pattern.

Java Database Connectivity

Table of Contents

What is JDBC?

- JDBC stands for **Java** **D**ata**B**ase **C**onnectivity.
- JDBC is a **Java API** used to connect and execute queries with the database.
- JDBC API consists of a set of **classes**, **interfaces** and **methods** to work with databases
- JDBC can be used to interact with every type of RDBMS such as MySQL, Oracle, Apache Derby, MongoDB, PostgreSQL, Microsoft SQL Server etc.
- It is a **part of JavaSE** (Java Platform, Standard Edition).
- JDBC API uses **JDBC drivers** to connect with the database.
- The current version of JDBC is **4.3**.

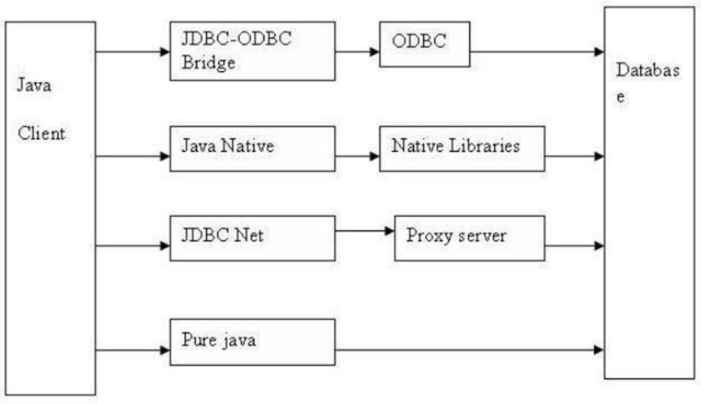
What is a Driver?

A driver is nothing but a piece of software required to connect to a database from Java program. JDBC drivers are client side adapters (**installed on the client machine**, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand.

Types of JDBC Drivers

There are four types of JDBC drivers:

1. Type-1 driver or JDBC-ODBC bridge driver
2. Type-2 driver or Native-API driver
3. Type-3 driver or Network Protocol driver
4. Type-4 driver or Thin driver



Pictorial representation of JDBC Drivers

Type-1 Driver

- This is the oldest JDBC driver, mostly used to connect database like MS Access from Microsoft Windows operating system.
- Type 1 JDBC driver actually translates JDBC calls into ODBC (Object Database connectivity) calls, which in turn connects to database.
- It converts the JDBC method calls into ODBC function calls.

Pros:

- Any database that provides an ODBC driver can be accessed

Cons:

- Features are limited and restricted to what ODBC driver is capable of
- Platform dependent as it uses ODBC which in turn uses **native O/S libraries**
- ODBC driver must be installed on client machine
- **Limited portability** as **ODBC driver is platform dependent** & may not be available for all platforms
- **Poor Performance** because of several layers of translation that take place before the program connects to database
- It is now **obsolete** and only used for development and testing.
- It has been removed from JDK 8 (1.8)

Type-2 Driver

- This was the second JDBC driver introduced by Java after Type 1, is hence it known as type 2.
- In this driver, performance was improved by **reducing communication layer**.
- Instead of talking to ODBC driver, **JDBC driver directly talks to DB** client using **native API**.
- That's why it's also known as **native API or partly Java driver**
- Type 2 drivers use the client side libraries of the database.
- The driver converts JDBC method calls into native database API calls.

Pros:

- Faster than JDBC-ODBC bridge as there is no conversion like ODBC involved
- Since it required native API to connect to DB client it is also less portable and platform dependent.

Cons:

- Client side libraries needs to be installed on client machine
- Driver is platform dependent
- Not all database vendors provide client side libraries
- Performance of type 2 driver is slightly better than type 1 JDBC driver.

Type-3 Driver

- This was the third JDBC driver introduced by Java, hence known as type 3.
- Type 3 driver makes use of middle tier between the Java programs and the database.
- Middle tier is an **application server** that converts JDBC calls into vendor-specific database calls.
- It was very different than type 1 and type 2 JDBC driver in sense that **it was completely written in Java** as opposed to previous two drivers which were not written in Java.
- That's why this is also known as all **Java driver**.
- This driver uses **3 tier approach i.e. client (java program), server and database**.
- So you have a Java client talking to a Java server and Java Server talking to database.
- Java client and server talk to each other using net protocol hence this type of JDBC driver is also known as Net protocol JDBC driver.
- This driver **never gained popularity** because database vendor was reluctant to rewrite their existing native library which was mainly in C and C++

Pros:

- No need to install any client side libraries on client machine
- Middleware application server can provide additional functionalities
- Database independence

Cons:

- Requires middleware specific configurations and coding
- May add extra latency as it goes through middleware server

Type-4 Driver

- Type 4 drivers are also called **Pure Java Driver**.
- This is the driver you are most likely using to connect to modern database like Oracle, SQL Server, MySQL, SQLLite and PostgreSQL.
- This driver is implemented in Java and directly speaks to database using its native protocol.
- It converts JDBC calls directly into vendor-specific database protocol.
- This driver includes all database calls in **one JAR file**, which makes it very easy to use.
- All you need to do to connect a database from Java program is to include JAR file of relevant JDBC driver.
- Because of **light weight**, this is also known as **thin JDBC driver**.
- Since this driver is also written in **pure Java**, **it is portable across all platforms**, which means you can use same JAR file to connect to MySQL even if your Java program is running on Windows, Linux or Solaris.
- Performance of this type of JDBC driver is also best among all of them because **database vendor liked this type** and all enhancements they make they also port for type 4 drivers.

Pros:

- Written completely in Java hence platform independent
- Provides better performance than Type 1 and 2 drivers as there is no protocol specific conversion is required
- Better than Type 3 drivers as it doesn’t need additional middleware application servers
- Connects directly to database drivers without going through any other layer

Cons:

- Drivers are database specific

Questions and Answers

Question 1: Which driver should we use?

Answer:

- A Type 4 driver is preferred if Java application is accessing any 1 database such as Oracle, Sybase etc.
- In case multiple databases are accessed then a Type 3 driver would be preferable.
- Type 2 drivers are recommended, if Type 3 or 4 drivers are not available for the database.
- Type 1 drivers are not recommended for production deployment.

Question 2: What factors decide the choice of drivers?

Answer: There are 2 factors that decide the choice of drivers: **portability** and **performance**.

Question 3: Which driver is least used?

Answer: Type 1 JDBC driver is the poorest in terms of portability and performance. **It is no longer used.**

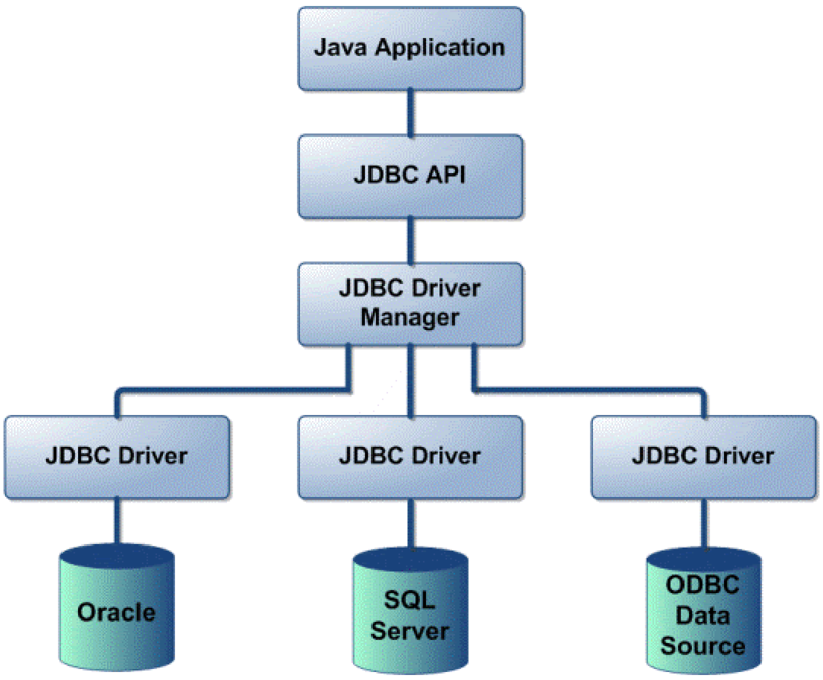
Question 4: Which is the best driver?

Answer: Type 4 JDBC driver is highly portable and gives the best performance.

Question 5: Which driver do we use to connect and transact with MySQL DB?

Answer: We use Type 4 driver

JDBC Architecture



1. **Java Application** is any application that likes to connect and transact with any database.
2. JDBC API
    1. It provides the application-to-DB Connection
    2. It provides the driver manager (**java.sql.DriverManager**)
    3. It uses the database specific driver to connect to heterogeneous databases.
3. Driver Manager
    1. The JDBC driver manager ensures that the **correct driver is used** to access each data source
    2. The driver manager is capable of supporting **multiple concurrent drivers** connected to **multiple heterogeneous databases**.
    3. One application can connect to different databases simultaneously.
4. **JDBC driver API** supports the JDBC Database connection.
    1. **Database vendors** provide the **JDBC drivers.**
    2. For example: MySQL vendor provides "**mysql-connector-java-8.0.19**" jar file that contains "**com.mysql.cj.jdbc.Driver**"

5. Databases
    1. **A Java application** can connect and transact with **multiple databases** simultaneously or one at a time.
    2. The vendors provide their specific drivers
    3. The Driver Manager takes care of all the drivers

Operations With DB

The following are the key operations we do with a database frequently.

1. Connect to DataBase
2. Execute Queries
    1. **C**reate/Insert Data
    2. **R**etrieve Data
    3. **U**pdate Data
    4. **D**elete Data
3. Close Connections/Resources

JDBC 4.3 API

JDBC 4.0 API is mainly divided into two package

- java.sql
- javax.sql

java.sql package

This package include classes and interface to perform almost all JDBC operation such as creating and executing SQL Queries.

Important classes and interface of java.sql package

classes/interface	**Description**
java.sql.BLOB	Provide support for BLOB(Binary Large Object) SQL type.
java.sql.Connection	Creates a connection with specific database
java.sql.CallableStatement	Execute stored procedures
java.sql.CLOB	Provide support for CLOB(Character Large Object) SQL type.
java.sql.Date	Provide support for Date SQL type.
java.sql.Driver	Create an instance of a driver with the DriverManager.
java.sql.DriverManager	This class manages database drivers.
java.sql.PreparedStatement	Used to create and execute parameterized query.
java.sql.ResultSet	It is an interface that provide methods to access the result row-by-row.
java.sql.Savepoint	Specify savepoint in transaction.
java.sql.SQLException	Encapsulate all JDBC related exception.
java.sql.Statement	This interface is used to execute SQL statements.

First JDBC Program

A succesfull operation/transaction with the database involves the exceution of **several small steps.** Each step has a **significance** of its own. To write efficient database code, all these steps must be correctly implemented. These steps are:

- **Step 1.** Pre-requisites
- **Step 2.** Connect to database
- **Step 3.** Create Statement
- **Step 4.** Prepare the Query
- **Step 5.** Execute the Query and Collect Data
- **Step 6.** Close Resources

Lets ponder over the above steps one by one

Step 1. Pre-requisites

Before writing a JDBC API, we need to:

1. Install any database server (here we will use **MySQL** Server).
2. Install a **GUI to operate on** the database server (here, MySQL GUI - **MySQL Workbench**)
3. While installation, set a **username** and a **password** (We set username: **root** and password: **1234**)
4. Create a schema (We start with **schema - school**)
5. Create the first table (We create **table - students**)
6. Create columns

No	lumn Name	lumn Properties
		, Primary Key, Not Null, Unique, Auto Increment
	dent_name	rchar (45), Not Null
	dent_class	(2), Not Null
	dent_fees	uble, Not Null

7. Make sure that MySQL Server is running before writing the JDBC API.
8. Place the suitable connector jar file in **WEB-INF>>lib** folder. (We use **mysql-connector-java-8.0.23.jar**)

Step 2. Connect to Database

1. Import the correct packages.

```
port java.sql.*;
```

2. Load the JDBC Driver

```
Class.forName("com.mysql.cj.jdbc.Driver");
```
where **com.mysql.cj.jdbc.Driver** is the location of Driver Class

3. Establish the connection

Connection conn = DriverManager.getConnection("**jdbc:mysql://localhost:3306/<schema-name>**", "**<username>**", "**<password>**");

where jdbc:mysql://localhost:3306/<schema-name> is **Connection String**

Step 3. Create a Statement

```
Statement statement = conn.createStatement();
```

Step 4. Prepare the query

```
String insertQuery = "INSERT INTO students (student_name, student_class, student_fees) values('Abhishek Verma', 1, 5000.0)";
```

Step 5. Execute the query and collect data

```
noOfRowsInserted = stmt.executeUpdate(insertQuery);
```

Step 6. Close the Resources

```
statement.close();
nnection.close();
```

Connect to Database

The first step before we can do any database transactions is to **connect with the database.**

JDBC API to Connect with DB

1. Import the correct packages.

```
port java.sql.*;
```

2. Load the JDBC Driver

The first thing you need to do before you can open a JDBC connection to a database is to load the JDBC driver for the database. You load the JDBC driver like this:

Class.forName("com.mysql.cj.jdbc.Driver");
where **com.mysql.cj.jdbc.Driver** is the location of Driver Class

- You only have to load the driver **once for the whole application**.
- You do not need to load it before every connection opened.
- Only before the first JDBC connection opened.

3. Establish the connection

Connection conn = DriverManager.getConnection("**jdbc:mysql://localhost:3306/<schema-name>**", "**<username>**", "**<password>**");
where jdbc:mysql://localhost:3306/<schema-name> is **Connection String**

The following classes and interfaces are used to connect to the database:

S. No	Class/Interface Name	Function
1	Class (C)	Class is a class in which all the drivers should be registered which will be used by the Java Application
2	forName(String)	forName() is a static method in class Class which loads and register our driver dynamically (by calling DriverManager.registerDriver())
3	Connection (I)	The JDBC Connection class, java. sql. Connection , represents a database connection to a relational database. Before we can read or write data from and to a database via JDBC, we need to open a connection to the database.
4.	DriverManager (C)	The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method Class.forName() that calls DriverManager.registerDriver() automatically
5.	getConnection (String, String, String)	This method of Java DriverManager class attempts to establish a connection to the database by using the given database url, username and password

Close the Resources

You should explicitly close *Statements*, *ResultSets*, and *Connections* when you no longer need them, unless you declare them in a *try*-with-resources statement (available in JDK 7 and after).
To close a Statement, ResultSet, or Connection object that is not declared in a **try-with-resources** statement, use its close method.

```
 resultSet.close();
statement.close();
nnection.close();
```

Statement

- Once a connection is obtained we can interact with the database.
- The JDBC framework provides **3 interfaces** and related methods and properties with which we can send SQL or PL/SQL commands to operate on the database.
- These interfaces are:

1. Statement interface
2. PreparedStatement interface

Let us look at them one by one

Statement (I)

1. Statement is an **Interface** in Java (JDBC)
2. It is present in java.sql package
3. We can obtain a JDBC Statement from a JDBC Connection
4. It is used to execute **static SQL statements** at runtime against an RDBMS.
5. It can be used to execute SQL **DDL** statements, for example data retrieval queries (**SELECT**)
6. It can be used to execute SQL **DML** statements, such as **INSERT**, **UPDATE** and **DELETE**

Syntax

```
Statement stmt = null;
try {
  stmt = connection.createStatement( ); // conn is Connection object
  . . .
}
catch (SQLException e) {
  . . .
}
finally {
  if(stmt!=null)
    stmt.close();
}
```

CRUD Operations

Create Operation – INSERT

```
Statement statement = connection.createStatement();


String insertQuery = "INSERT INTO students (student_name, student_class, student_fees) values('Kinjal', 1,
5000.0)";


int noOfRowsInserted = statement.executeUpdate(insertQuery);
```

Retrieve Operation – SELECT

```
Statement statement = connection.createStatement();


String selectQuery = "SELECT * FROM students";
ResultSet resultSet = statement.executeQuery(selectQuery);
while (resultSet.next()) {
   resultSet.getInt("_id"); // resultSet.getInt(1);
   resultSet.getString("student_name"); // resultSet.getString(2);
```

```
    resultSet.getInt("student_class"); // resultSet.getInt(3);
    resultSet.getDouble("student_fees"); // resultSet.getDouble(4);
}
```

Update – UPDATE

```
Statement statement = connection.createStatement();

String updateQuery = "UPDATE students SET student_class = 12 WHERE student_class = 11";

int noOfRowsUpdated = statement.executeUpdate(updateQuery);
```

Delete – DELETE

```
Statement statement = connection.createStatement();

String deleteQuery = "DELETE FROM students WHERE student_class = 12";

int noOfRowsDeleted = statement.executeUpdate(deleteQuery);
```

Methods

int executeUpdate(String sqlQuery)

This method is used to execute a DML SQL Query (Insert, Updaate and Delete queries)

ResultSet executeQuery(String sqlQuery)

This method is used to retrieve data from the table and returns a result set

The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	**Recommended Use**
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

It has the following steps:

Create Statement

Step 4. Execute SQL command

Execute the Query via the Statement

You do so by calling its **executeQuery()** method, passing an SQL statement as parameter.

Step 5. Collect Information

There are different types of operations that we do with the DB such as insert, update, delete and retrieve. In every case we get different informations.

- In case of insert, update and delete operations, we get number of records inserted, updates or deleted.
- In case of retrieval, we get rows of data in a ResultSet

ResultSet

1. ResultSet is an Interface in Java (JDBC)
2. It is used to hold records that are returned as a result of a SELECT query
3. We need to travel/traverse/iterate through the ResultSet to get records

Create ResultSet

```
ResultSet resultSet = statement.executeQuery(sql);
```

Get the data from ResultSet

1. To iterate the ResultSet you use its **next()** method.
2. The next() method returns **true** if the ResultSet has a next record, and moves to the next record.
3. If there were no more records, next() returns **false**, and you can no longer extract data.

```
while(resultSet.next()) {
    System.out.println("Id = " + resultSet.getInt(1));
    System.out.println("Name = " + resultSet.getString(2));
System.out.println("Class = " + resultSet.getInt(3));
```

Step 5. Close the Resources

```
resultSet.close();
statement.close();
connection.close();
```