Java Database Connectivity Project

Table of Contents

Java Database Connectivity Project

Overview of new technology associated with this project

JDBC and Design Patterns are used throughout the industry. These links will give you an overview on these topics.

What is JDBC?

Java Database Connectivity (**JDBC**[1]) is an application programming interface (API) for the programming language Java, which defines how a client may access a database. It is a Java-based data access technology used for Java database connectivity. It is part of the Java Standard Edition platform, from Oracle Corporation. It provides methods to query and update data in a database, and is oriented towards relational databases. A JDBC-to-ODBC bridge enables connections to any ODBC-accessible data source in the Java virtual machine (JVM) host environment.

What are design patterns[2]?

Design Patterns: Elements of Reusable Object-Oriented Software is a software engineering book describing software design patterns. The book's authors are Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides with a foreword by Grady Booch. The book is divided into two parts, with the first two chapters exploring the capabilities and pitfalls of object-oriented programming, and the remaining chapters describing 23 classic software design patterns. The book includes examples in C++ and Smalltalk.

It has been influential to the field of software engineering and is regarded as an important source for object-oriented design theory and practice. More than 500,000 copies have been sold in English and in 13 other languages. The authors are often referred to as the **Gang of Four** (**GoF**).[1]

What is an Abstract factory pattern[3]?

The **abstract factory pattern** provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes.[1] In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the theme. The client doesn't know (or care) which concrete objects it gets from each of these internal factories, since it uses only the generic interfaces of their products.[1] This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.[2]

An example of this would be an abstract factory class `DocumentCreator` that provides interfaces to create a number of products (e.g. `createLetter()` and `createResume()`). The system would have any number of derived concrete versions of the `DocumentCreator` class like `FancyDocumentCreator` or `ModernDocumentCreator`, each with a different implementation of `createLetter()` and `createResume()` that would create a corresponding object like `FancyLetter` or `ModernResume`. Each of these products is derived from a simple abstract class like `Letter` or `Resume` of which the client is aware. The client code would get an appropriate instance of the `DocumentCreator` and call its factory methods. Each of the resulting

[1] https://en.wikipedia.org/wiki/Java_Database_Connectivity

[2] https://en.wikipedia.org/wiki/Design_Patterns

[3] https://en.wikipedia.org/wiki/Abstract_factory_pattern

objects would be created from the same `DocumentCreator` implementation and would share a common theme (they would all be fancy or modern objects). The client would only need to know how to handle the abstract `Letter` or `Resume` class, not the specific version that it got from the concrete factory.

A **factory** is the location of a concrete class in the code at which objects are constructed. The intent in employing the pattern is to insulate the creation of objects from their usage and to create families of related objects without having to depend on their concrete classes.[2] This allows for new derived types to be introduced with no change to the code that uses the base class.

Project Developing a JAVA class library based upon Microsoft JDBC

You will have to download the JDBC driver.

Sample retrieval example

In the following example, the sample code sets various connection properties in the connection URL, and then calls the getConnection method of the DriverManager class to return a SQLServerConnection object. Next, the sample code uses the createStatement method of the SQLServerConnection object to create a SQLServerStatement object, and then the executeQuery method is called to execute the SQL statement. Finally, the sample uses the SQLServerResultSet object returned from the executeQuery method to iterate through the results returned by the SQL statement

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ConnectURL {
    public static void main(String[] args) {

        // Create a variable for the connection string.
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;user=
<user>;password=<password>";

        try (Connection con = DriverManager.getConnection(connectionUrl); Statement stmt =
con.createStatement();) {
            String SQL = "SELECT TOP 10 * FROM Person.Contact";
            ResultSet rs = stmt.executeQuery(SQL);

            // Iterate through the data in the result set and display it.
            while (rs.next()) {
                System.out.println(rs.getString("FirstName") + " " + rs.getString("LastName"));
            }
        }
        // Handle any errors that may have occurred.
        catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```