COMP SOEN 6441(Section U Fall 2022)

Project Report

Submitted To: Dr. Constantinos Constantinides, P.Eng.

STUDENT NAME

Vasudev Sharma (Student ID 40236456)

Hammad Ali (Student ID 40220732)

1 Base Application

The base application or the project consists of a Command Line Interface which takes input from the user and displays the entire content of the desired database table onto the screen. The project has been developed using Java programming language. The relational database has been designed using MYSQL database because it is open source. The database consists of data taken from an Oracle API in the JSON format which consists data of several countries and cities along with their attributes such as population, language, area, etc. The relational database consists of three tables. The application's main goal is to fetch data from the database, map it to the domain objects and then it displays onto the screen for the user. The application makes use of parameterized queries to fetch the data from the database. The application makes use of Junit4 for testing and AspectJ for logging. For our application we did not use the entire dataset as it was too large, hence we only used a small portion of it.

2 Coding Standards

The following Java coding standards were followed to make the project: -

2.1 Naming Conventions

The Java naming conventions were followed while naming folders, packages, classes, methods, and variables (global as well as local) in our project for good maintenance and readability of code.

2.2 Error handling

Several errors occurred during the life cycle of the java project, but these errors were handled by using try-catch expressions. Some of the common exceptions that were caught is SQLException.

2.3 Comments

Comments are used for understanding and readability of the code easily. Many comments are added in our project which give out crucial details about the project. There are two types of comments that have been used, they are single line comments and multiple lines comments.

2.4 Annotations

Annotations help convey metadata to the program which ensures a smoother execution of our java code. The annotation used in our code are @Override (while overriding functions) and @Test (for testing).

2.5 File packages and structuring

The packaging and file structure was constructed in such a manner that classes or code with similar functionality was bundled together, separating them from unrelated functionality thus increasing modularity of the code.

2.6 Indentation and spacing

Appropriate indentation and spacing is good while coding because it is then easier for the readers to understand the code. In the project, all code is indented and spaced in the proper way.

2.7 Access Modifiers

Access modifiers are used to define the access to various elements of the Java code such as classes, methods, and variables, that is why it important to use adhere to the conventions while using them, so data mismanagement does not occur.

2.8 Javadoc

The comments enclosed in /** */ are used for generating Javadoc for improving understanding of the project.

2.9 Testing

To ensure a robust and smoothly working application, we have used Junit 4 to test our application. We have employed unit and integration testing.

2.10 Project management and development

We are using GitHub to store and manage several versions of our project. Also, we are using 'Iterative' software model for making our project.

The below figure gives a snapshot of the project and the coding standards used in it (Comments, Annotations, Naming conventions, Junit testing files, Error handling, packaging structure, Github , etc.): -

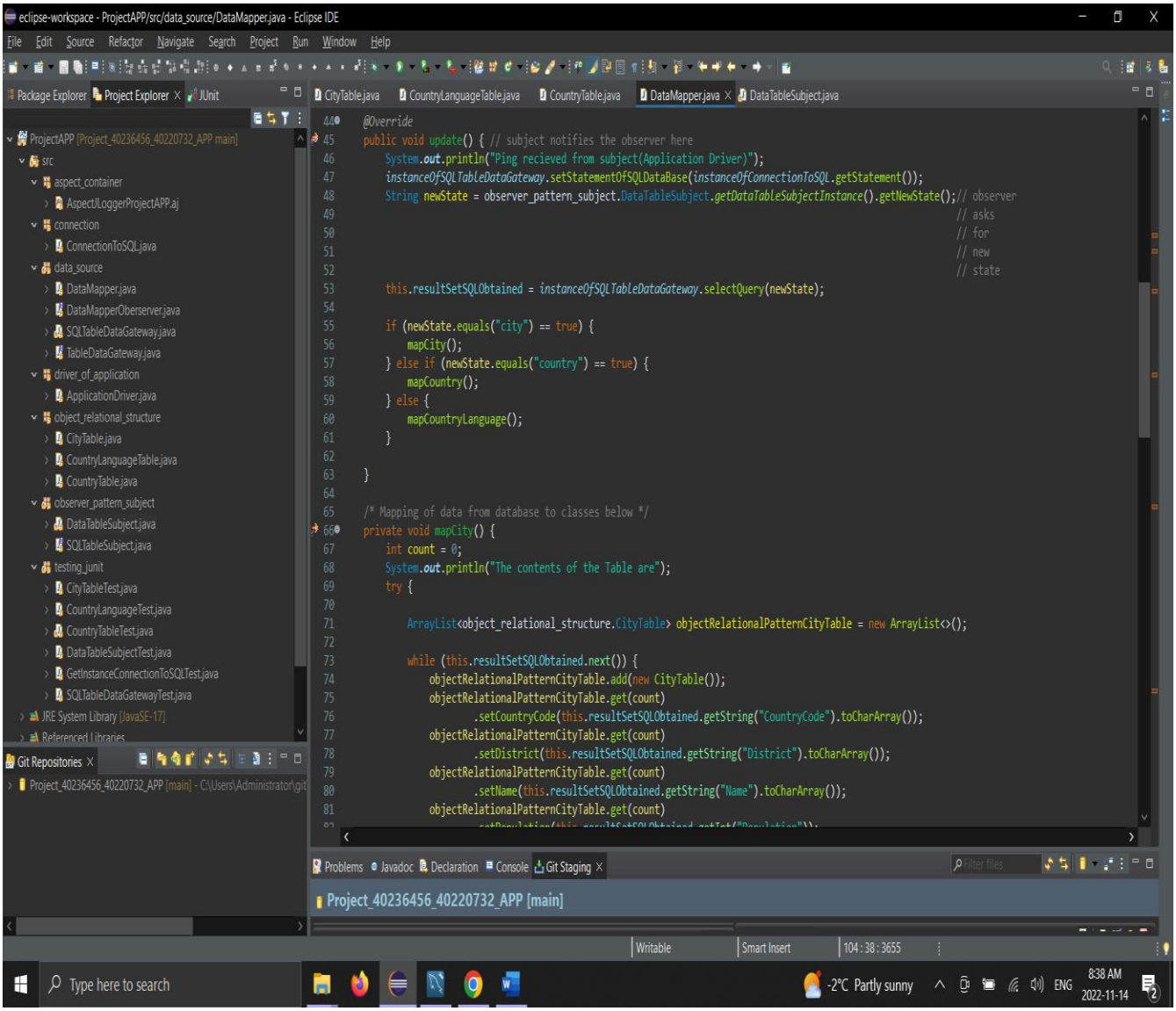


Figure 1. Snapshot of the project which shows the deployment of Java coding standards (as mentioned above)

3 Applicable Patterns

1. Object Relational Structural Pattern

The ‘table per class mapping’ or also called ‘vertical mapping’ has been deployed in the project. Each class can be mapped to a table in the MYSQL database, and each attribute corresponds to a column in their mapped tables. The advantage of using such a pattern is that it is easy to understand and implement, and all the columns of the table can used. There exists a one-to-one mapping between Java classes and MYSQL tables. The purpose of the object relational structural pattern in the project is to separate the Java programming part and the relational database. Each row in the database represents one instance of the java classes and columns depicts the attributes.

In the project source code, under package name ‘object_relational_structure’ we have three classes called CityTable, CountryLanguageTable and CountryTable, these classes correspond to the tables city, country and countrylanguage in the database. Theses classes also include accessors(getter) and mutators(setter) for the attributes. The following screenshots below give a visualization of this pattern: -

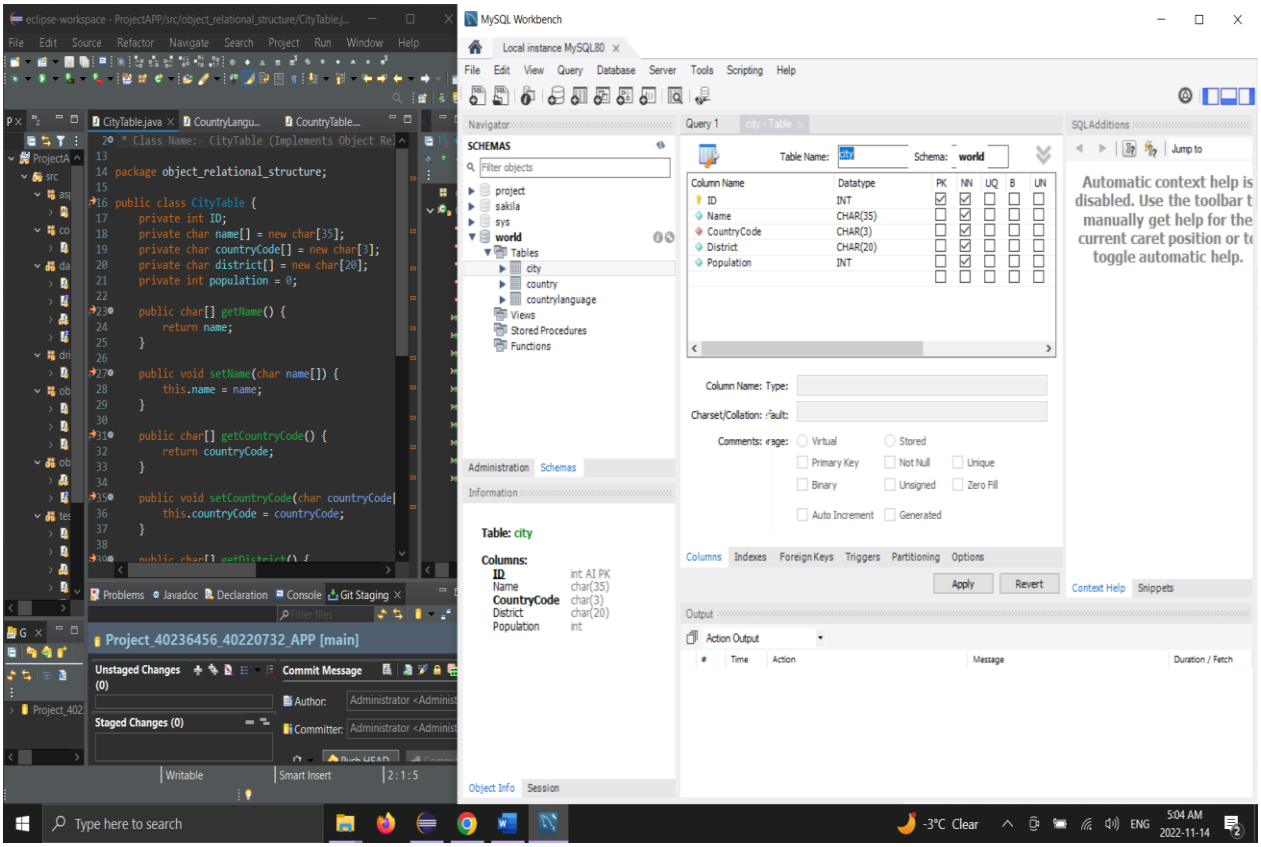


Figure 2. Vertical Mapping: CityTable.java to City (table in MySQL)

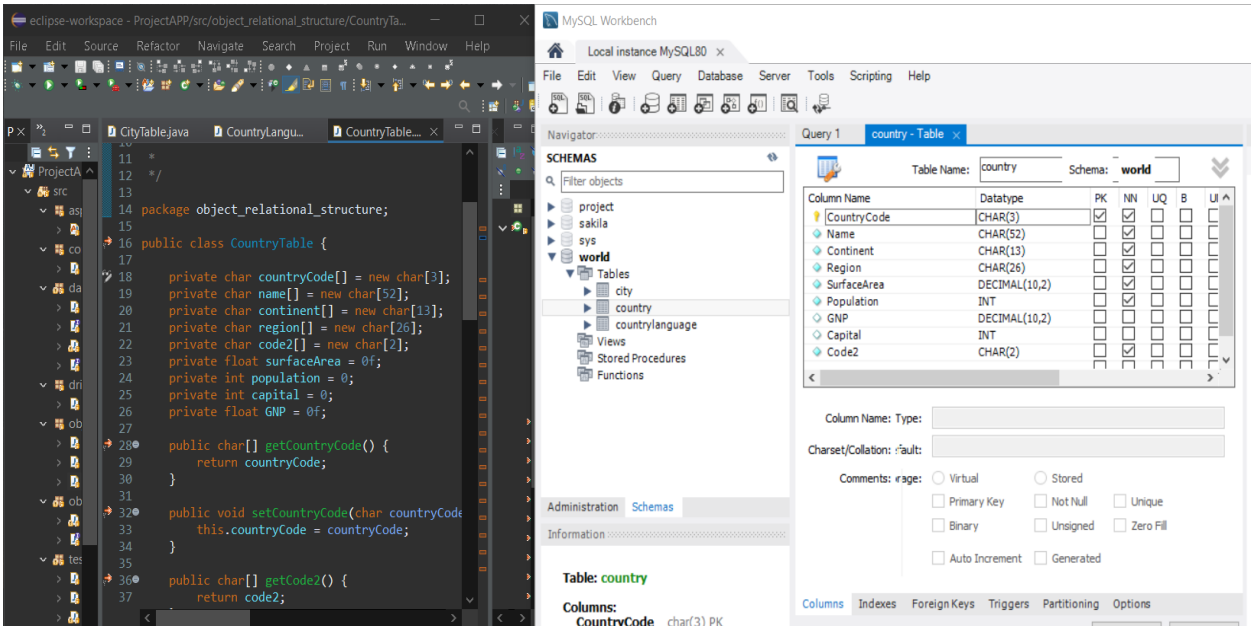


Figure 3. Vertical Mapping: CountryTable.java to Country (table in MySQL)

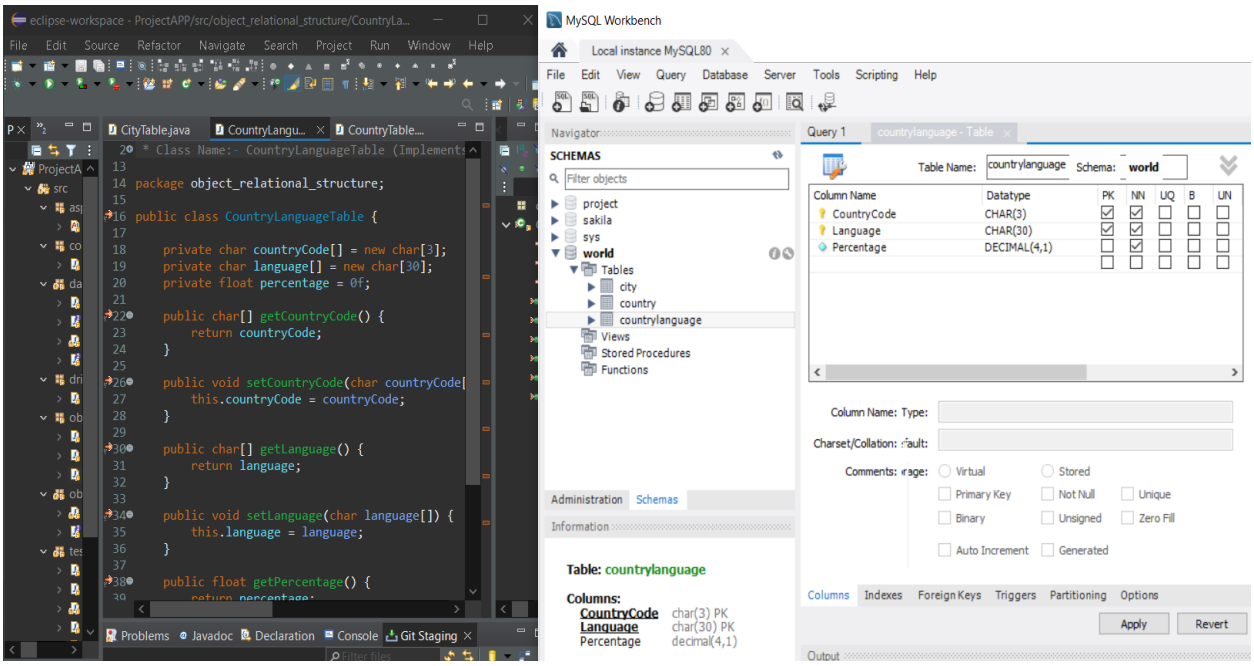


Figure 4. Vertical Mapping: CountryLanguageTable.java to Countrylanguage (SQL Table)

The ‘DataMapper.java’ is used to map the SQL data to the Java classes mentioned above. The below screenshot shows the use of mapCity(), mapCountry , mapCountryLanguage() functions in the DataMapper class used for mapping SQL data to the corresponding java classes:-

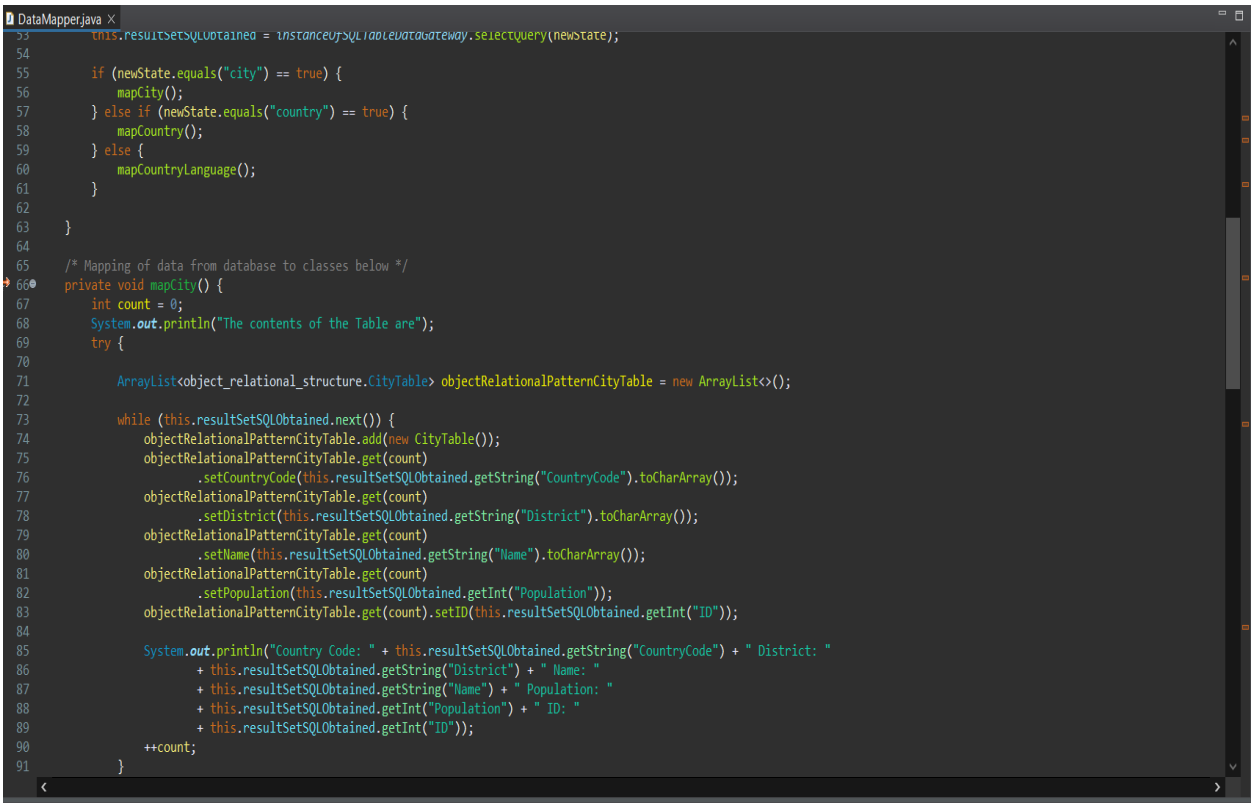


Figure 5. The screenshot highlights the mapping of SQL data obtained from executing a particular query to Java classes using the functions made by project developers.

2. Data Source Architectural Patterns

The data source architectural pattern used for accessing the data services layer of our application is 'Data Mapper combined with Table Data Gateway (TDG)'. The purpose of the data mapper is to set up communication between objects (Domain and Relational) and ensures that they are independent of each other hence this pattern was used in our project.

The input is fed to the Command Line Interface (CLI) in the presentation layer, which is passed onto the 'ApplicationDriver' class. The 'ApplicationDriver' passes it to the 'DataTableSubject.java' class which is the controller. The controller passes it onto the 'DataMapper.java' which then establishes communication between domain objects and relational database. Once mapping is done, the SQL output is printed and then we go back to the main menu.
The diagram below depicts in detail our data source architectural pattern: -

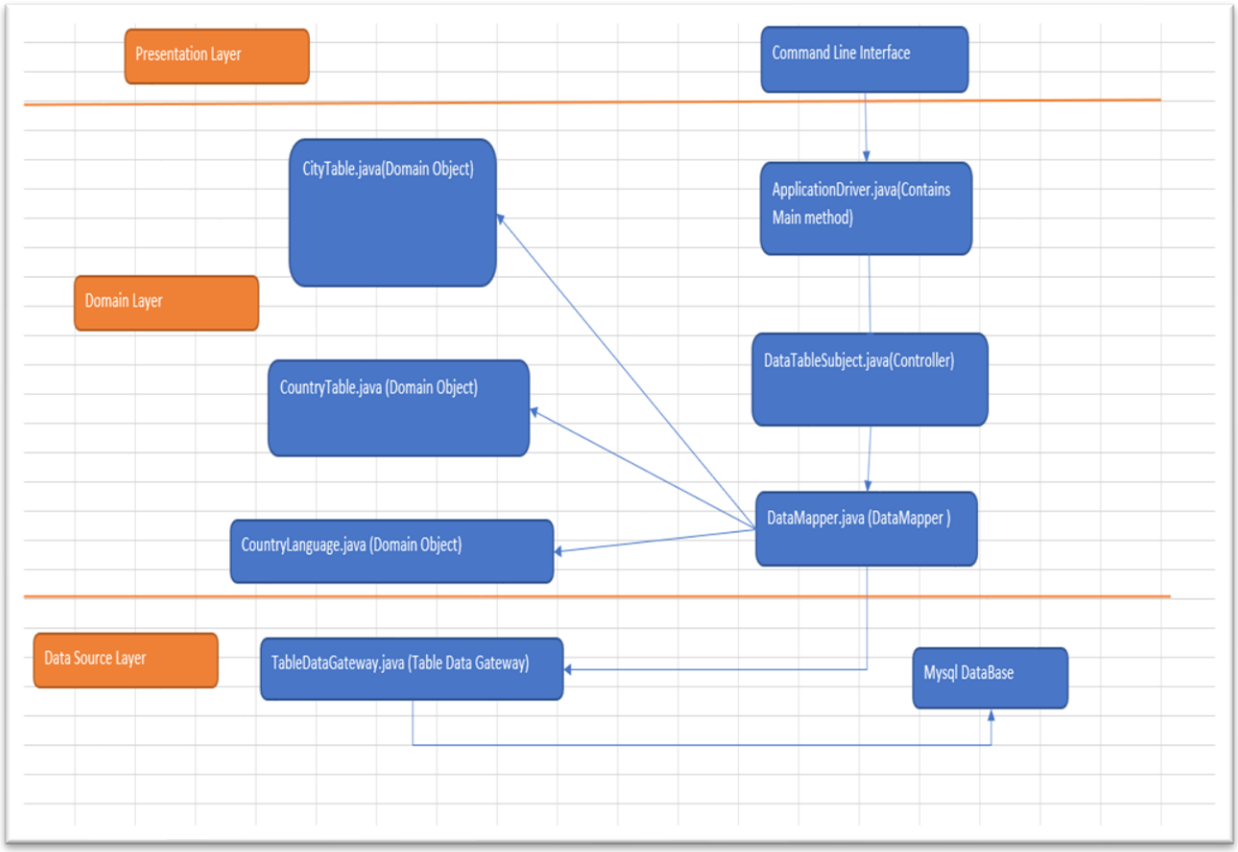


Figure 6. Data source architecture pattern for the project

3. Design Patterns

There are two design patterns that have been deployed in our code which are: -

1. Singleton design pattern
2. Observer design pattern (Pull)

Singleton Pattern

We are using singleton pattern in four different java classes in our project because we only want one instance of these classes. All four of these classes have a private constructor and a public-static method which returns the instance of the class.

1)The first class is the ConnectionToSQL class because we only want one connection to the database.

ConnectionToSQL.java

```
 * Class Name:- ConnectionToSQL (Deploys Singleton Design Pattern and creates connection to MYSQL database)

package connection;

import java.sql.*;

public class ConnectionToSQL {

    private static final String DATABASE_URL = "jdbc:mysql://localhost/WORLD";
    private static final String USERINFO = "root";
    private static final String PASSWORD = "Forky_123";
    private static ConnectionToSQL instanceConnectionToSQL = null;

    private ConnectionToSQL() { // Singleton Pattern. Only one connection should exist to the MYSQL database.

    }

    public static ConnectionToSQL getInstanceConnectionToSQL() {

        if (instanceConnectionToSQL == null) {
            instanceConnectionToSQL = new ConnectionToSQL();
        }
        return instanceConnectionToSQL;
    }

    public java.sql.Statement getStatement() {
        try {

            Connection connectionInstance = DriverManager.getConnection(DATABASE_URL, USERINFO, PASSWORD);
            Statement statmentConnection = connectionInstance.createStatement();
            return statmentConnection;

        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }

}
```

Figure 7. Singleton pattern in ConnectionToSQL.java

2)The second class is ‘DataMapper.java’ because we want only one instance of this class to transfer or map the data, multiple instances of these classes can cause synchronization problems while manipulating data.

DataMapper.java | DataTableSubject.java

```
 9  *It implements an interface for DataMappers
10  *
11  *
12  */
13
14 package data_source;
15
16 import object_relational_structure.*;
21
22 public class DataMapper implements DataMapperOberserver { // This is the observer in the Observer Design Pattern
23
24     private static DataMapper instanceDataMapper = null;
25
26     private static SQLTableDataGateway instanceOfSQLTableDataGateway = SQLTableDataGateway
27             .getInstanceSQLTableDataGateway();
28     private static ConnectionToSQL instanceOfConnectionToSQL = ConnectionToSQL.getInstanceConnectionToSQL();
29     private ResultSet resultSetSQLObtained = null;
30
31     private DataMapper() {
32
33     }
34
35     public static DataMapper getInstanceDataMapper() {
36
37         if (instanceDataMapper == null) {
38             instanceDataMapper = new DataMapper();
39         }
40         return instanceDataMapper;
41
42     }
43
44     @Override
45     public void update() { // subject notifies the observer here
46         System.out.println("Ping recieved from subject(Application Driver)");
47         instanceOfSQLTableDataGateway.setStatementOfSQLDataBase(instanceOfConnectionToSQL.getStatement());
48         String newState = observer_pattern_subject.DataTableSubject.getDataTableSubjectInstance().getNewState();// observer
49                                                                                                                 // asks
50                                                                                                                 // for
51                                                                                                                 // new
```

Figure 8. Singleton pattern in DataMapper.java

3)The third class is ‘SQLTableDataGateway.java’ because we want only one unit of communication with the database to avoid synchronization problems if multiple instances are existing.

SQLTableDataGateway.java

```
/* Class Name:- SQLTableDataGateway (Implements Data Source Pattern(TDG) and has Singleton Design Pattern)

package data_source;

import java.sql.ResultSet;

public class SQLTableDataGateway implements TableDataGateway {

    private String tableName;
    private ResultSet resultSetSQLReturned;
    private Statement statementOfSQLDataBase;

    private static SQLTableDataGateway instanceSQLTableDataGateway = null;

    private SQLTableDataGateway() { // singleton pattern

    }

    public static SQLTableDataGateway getInstanceSQLTableDataGateway() { // singleton pattern

        if (instanceSQLTableDataGateway == null) {
            instanceSQLTableDataGateway = new SQLTableDataGateway();
        }
        return instanceSQLTableDataGateway;
    }

    @Override
    public java.sql.ResultSet selectQuery(String tableName) { // parameterized query is run
        setTableName(tableName);

        String selectQuery = "select * from " + this.tableName+"";

        try {
            this.resultSetSQLReturned = statementOfSQLDataBase.executeQuery(selectQuery);
        } catch (SQLException e) {

            e.printStackTrace();
        }
        return this.resultSetSQLReturned;
```

Figure 9. Singleton pattern in SQLTableDataGateway.java

4)The fourth class is ‘DataTableSubject.java’ which is the controller in our application, we want the data handling pipeline in our project to have only one instance each to avoid data synchronization problems, otherwise multiple instances can change same data at the same time.

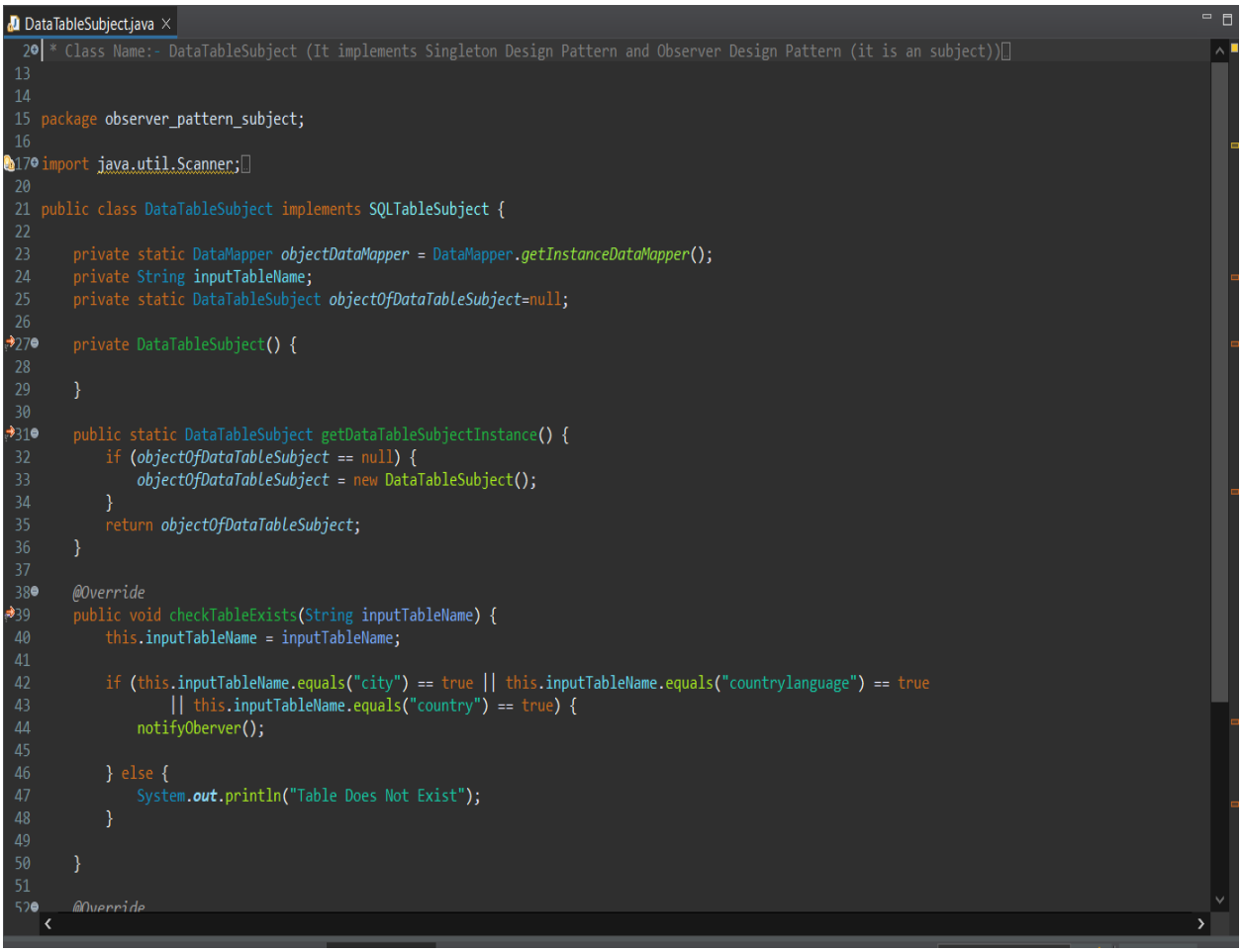


Figure 10. Singleton pattern in DataTableSubject.java

Observer Design Pattern (Pull)

The observer design pattern exists between the 'DataMapper.java' (Observer) and the 'DataTableSubject.java'(Subject). There can exist many to one relationship between the observer and the subject hence we have deployed this pattern.

In 'DataTableSubject.java' if a state change occurs it notifies the observer 'DataMapper.java' using the notifyObserver() method. Then in the update() method of 'DataMapper.java' the ping(message from Subject) is received, the observer then asks for the new state of the subject by invoking the call to getNewState() function inside the subject class. The getNewState() function in the 'DataTableSubject.java' returns the new state of the class to the 'DataMapper.java' (Observer). The following screenshots show the two classes and their implementation of the pattern: -

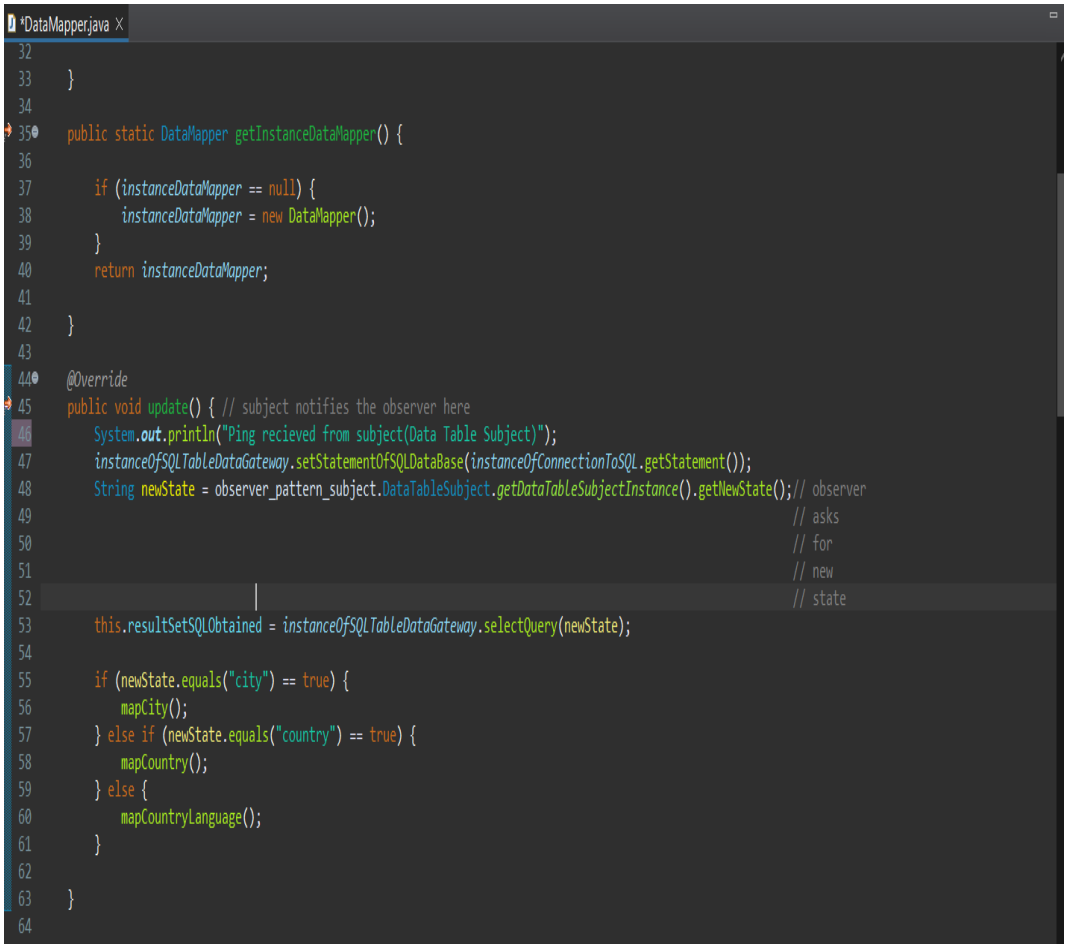


Figure 11. Observer side implementation in DataMapper.java (update() function)

The below screenshots show the subject side of things in DataTableSubject class

DataTableSubject.java

```
32        if (objectOfDataTableSubject == null) {
33            objectOfDataTableSubject = new DataTableSubject();
34        }
35        return objectOfDataTableSubject;
36    }
37
38    @Override
39    public void checkTableExists(String inputTableName) {
40        this.inputTableName = inputTableName;
41
42        if (this.inputTableName.equals("city") == true || this.inputTableName.equals("countrylanguage") == true
43                || this.inputTableName.equals("country") == true) {
44            notifyOberver();
45
46        } else {
47            System.out.println("Table Does Not Exist");
48        }
49
50    }
51
52    @Override
53    public void notifyOberver() {  // Observer Pattern || Observer ping sent of state change
54        objectDataMapper.update();
55
56    }
57
58    public String getNewState() {
59        return this.inputTableName;
60    }
61
62 }
63
```

Figure 12. Subject side implementation in DataTableSubject.java (notifyObserver() and getnewState() function)

Using an AspectJ at an intermediate step in the lifecycle of our project we captured a sequence diagram generated using PlanUML which shows the observer pattern in action. Check for steps [0011], [0012] and [0016] in figure 13 for the observer pattern execution.

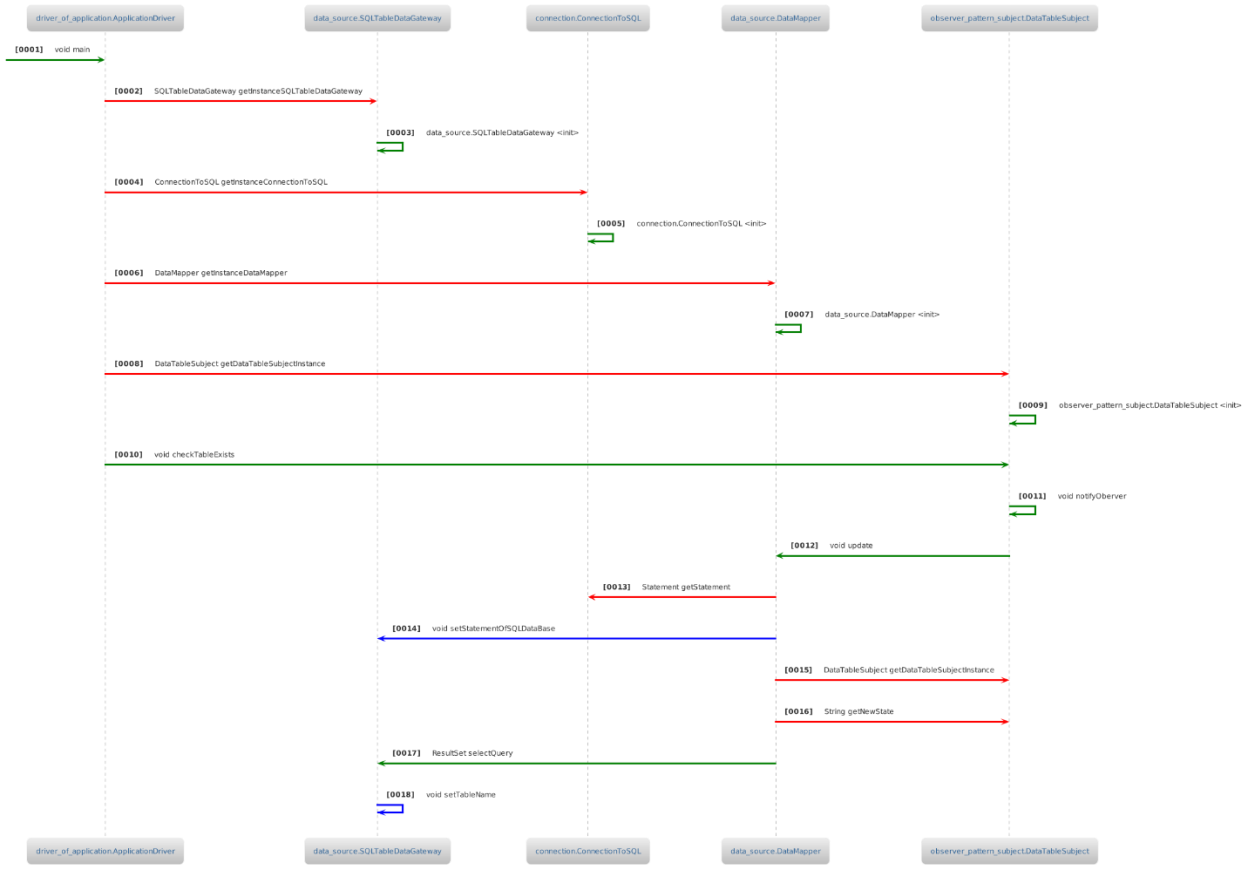


Figure 13. Observer pattern sequence diagram

4 Refactoring strategies

There are several refactoring strategies deployed across the project, they are discussed in more detail below: -

4.1 Extract Method Refactoring strategy

Initially all the operations for data mapping (between classes and database) were done inside a single function (update ()) in the 'DataMapper.java' class hence a lot of refactoring had to be done on it. The vast mapping operations were split and three different mapping functions (mapCity(), mapCountry() and mapCountryLanguage()) were created.

Before Refactoring:

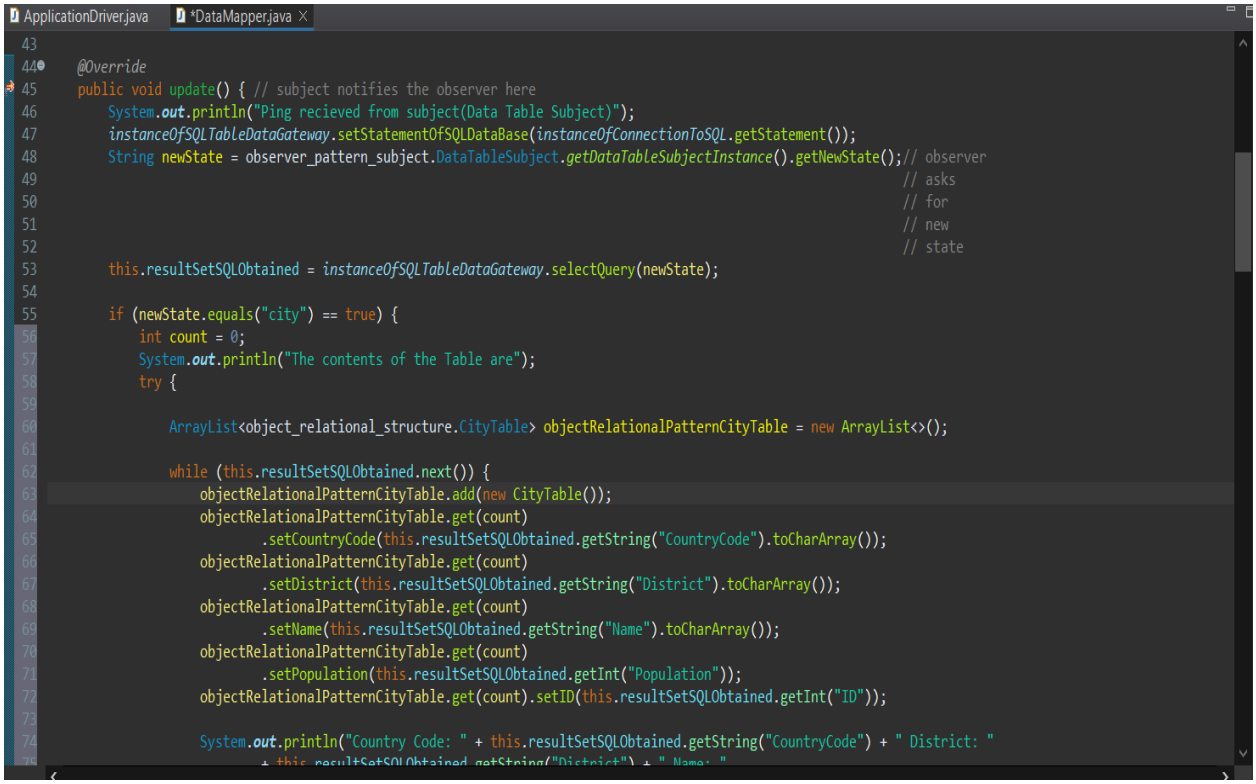


Figure 14. Before refactoring of Data mapper, we do mapping in single function

After Refactoring:

DataMapper.java

```
public static DataMapper getInstanceDataMapper() {

    if (instanceDataMapper == null) {
        instanceDataMapper = new DataMapper();
    }
    return instanceDataMapper;

}

@Override
public void update() { // subject notifies the observer here
    System.out.println("Ping recieved from subject(Data Table Subject)");
    instanceOfSQLTableDataGateway.setStatementOfSQLDataBase(instanceOfConnectionToSQL.getStatement());
    String newState = observer_pattern_subject.DataTableSubject.getDataTableSubjectInstance().getNewState();// observer
                                                                                                            // asks
                                                                                                            // for
                                                                                                            // new
                                                                                                            // state
    this.resultSetSQLObtained = instanceOfSQLTableDataGateway.selectQuery(newState);

    if (newState.equals("city") == true) {   //after refactor
        mapCity();
    } else if (newState.equals("country") == true) {
        mapCountry();
    } else {
        mapCountryLanguage();
    }

}
```

Figure 15. After refactoring we only do message passing (function calls) to mapper functions.

4.2 Extract Class Refactoring Strategy Refactoring strategy

Initially the ApplicationDriver.java had database connection and several other unrelated tasks in it which were later removed by creating two new class ConnectionToSQL.java and DataTableSubject.java

Before Refactoring:

Lot of useless code in ApplicationDriver.java

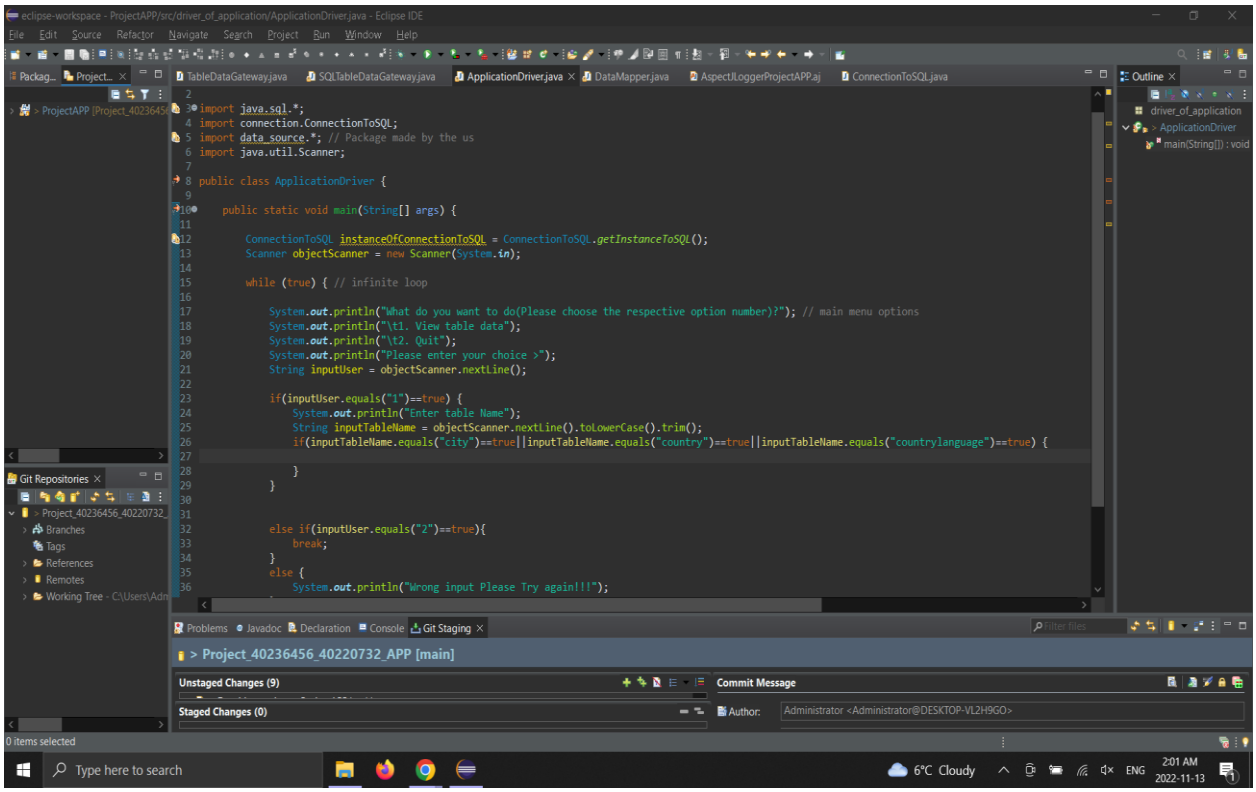


Figure 16.a Before refactoring we have unnecessary table comparison tasks in the application driver.

```java
package driver_of_application;

import java.util.Scanner;

import connection.ConnectionToSQL;
import observer_pattern_subject.*;

public class ApplicationDriver {

    private static final String DATABASE_URL = "jdbc:mysql://localhost/WORLD";
    private static final String USERINFO = "root";
    private static final String PASSWORD = "Forky_123";
    private static ConnectionToSQL instanceConnectionToSQL = null;

    public static void main(String[] args) {

        Scanner objectScanner = new Scanner(System.in);
```

Figure 16.b Before refactoring we have unnecessary connection parameters in the application driver.

After Refactoring:

We created two new class (ConnectionToSQL.java and DataTableSubject.java) and they took away the unnecessary code in ApplicationDriver.java. The below screenshots show the refactoring done: -

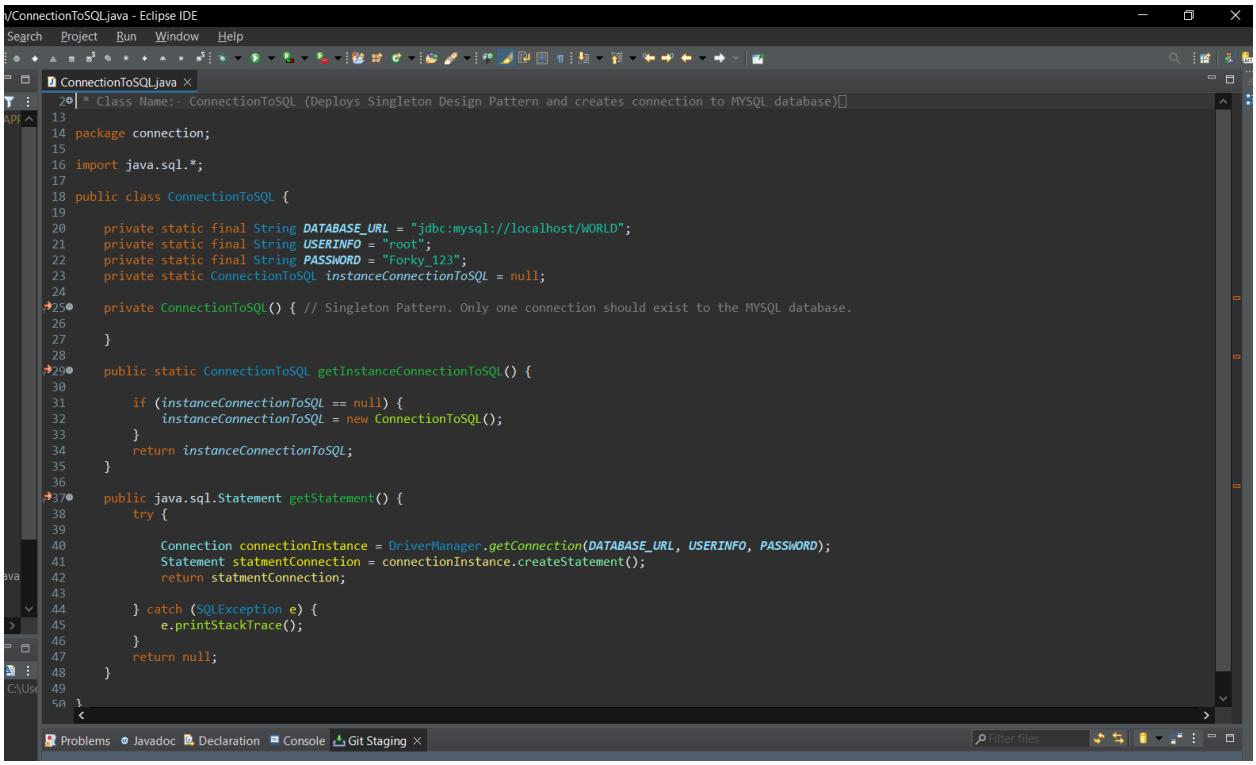


Figure 17.a New class created for establishing connection

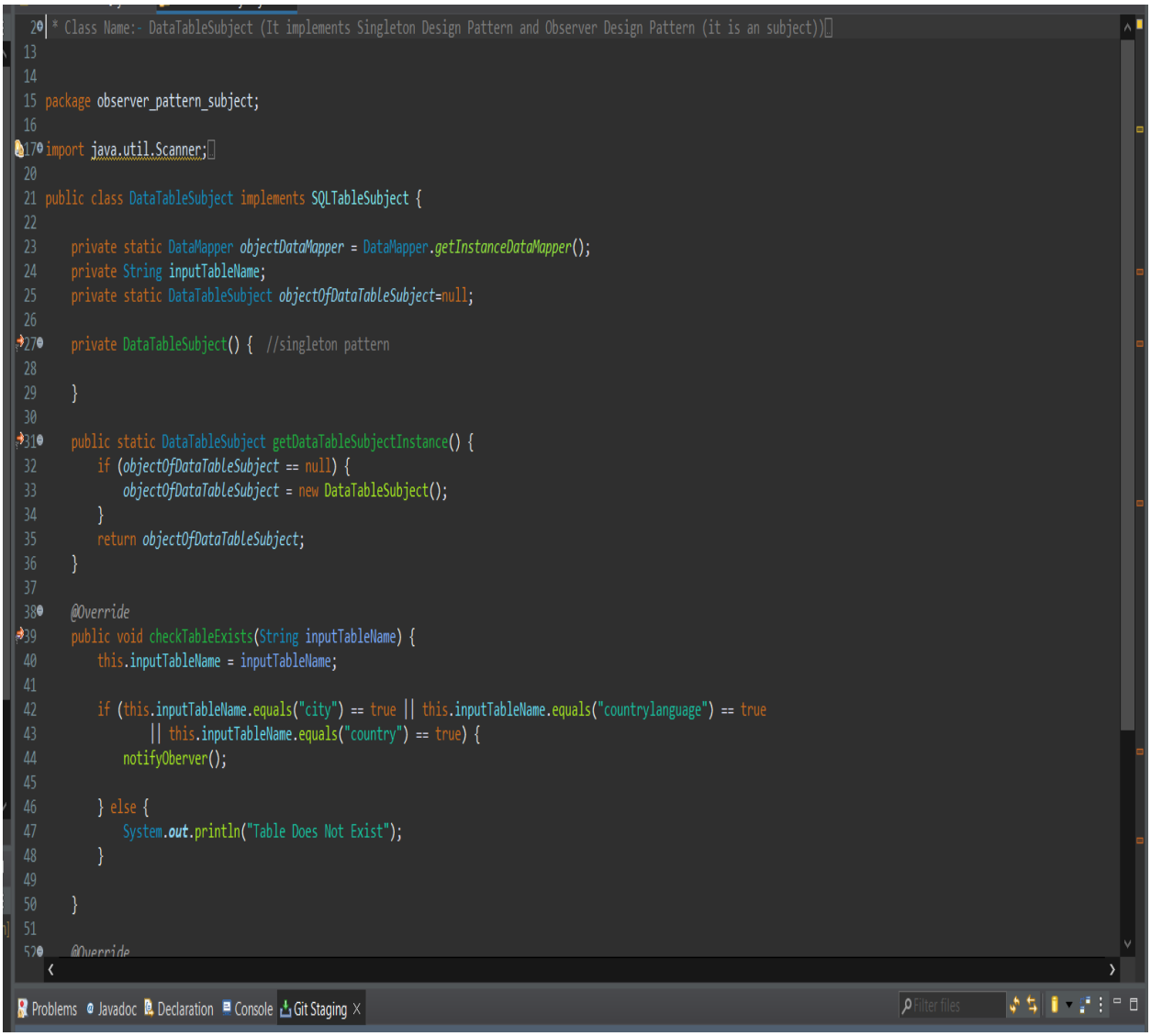


Figure 17.b New class DataTableSubject which now acts as a controller.

4.3 Hide Delegate Refactoring strategy

As discussed earlier in Data Source Architectural Patterns (in this documentation) the DataMapper separates the domain object (Java classes) and the data source layer (TDG and database), earlier TDG was directly handling data flow, but after adding the data mapper, it has separated the domain objects and data source objects, hence hiding them from each other and making them independent.

Before Refactoring:

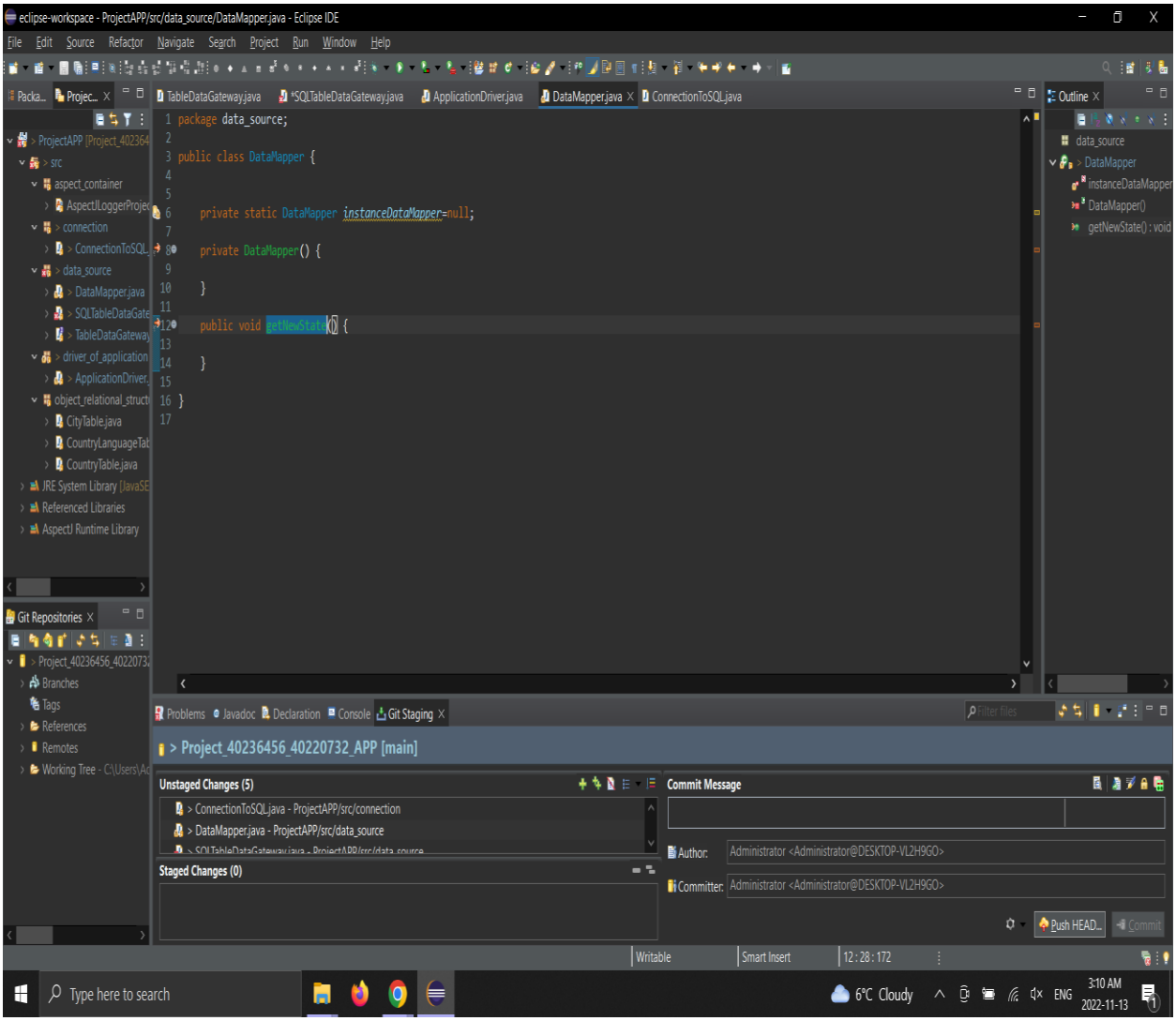


Figure 18.a Data mapper does not do anything here, everything done in TDG initially.

After Refactoring:

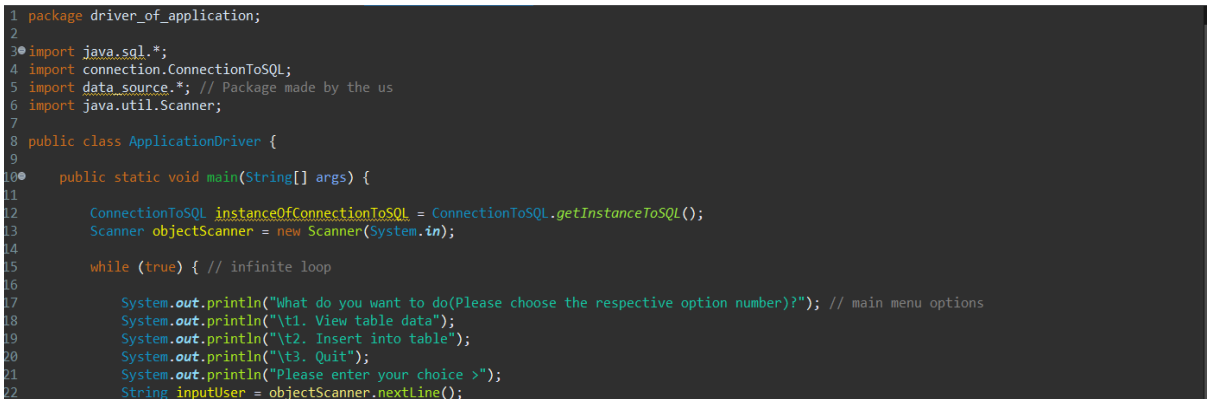
```
/* Class Name:- DataMapper (It implements Singleton Pattern and Observer Pattern (it is an observer))

package data_source;

import object_relational_structure.*;

public class DataMapper implements DataMapperOberserver { // This is the observer in the Observer Design Pattern

    private static DataMapper instanceDataMapper = null;

    private static SQLTableDataGateway instanceOfSQLTableDataGateway = SQLTableDataGateway
            .getInstanceSQLTableDataGateway();
    private static ConnectionToSQL instanceOfConnectionToSQL = ConnectionToSQL.getInstanceConnectionToSQL();
    private ResultSet resultSetSQLObtained = null;
    private static ArrayList<object_relational_structure.CityTable> objectRelationalPatternCityTable = new ArrayList<>();
    private static ArrayList<object_relational_structure.CountryTable> objectRelationalPatternCountryTable = new ArrayList<>();
    private static ArrayList<object_relational_structure.CountryLanguageTable> objectRelationalPatternCountryLanguageTable = new ArrayList<>();
    private DataMapper() {  //singleton pattern

    }

    public static DataMapper getInstanceDataMapper() {

        if (instanceDataMapper == null) {
            instanceDataMapper = new DataMapper();
        }
        return instanceDataMapper;

    }

    @Override
    public void update() { // subject notifies the observer here
        System.out.println("Ping recieved from subject(Data Table Subject)");
        instanceOfSQLTableDataGateway.setStatementOfSQLDataBase(instanceOfConnectionToSQL.getStatement());
        String newState = observer_pattern_subject.DataTableSubject.getDataTableSubjectInstance().getNewState();// observer
                                                                                                                 // asks
                                                                                                                 // for
                                                                                                                 // new
                                                                                                                 // state
        this.resultSetSQLObtained = instanceOfSQLTableDataGateway.selectQuery(newState);

        if (newState.equals("city") == true) {   //after refactor
            mapCity();
        } else if (newState.equals("country") == true) {
```

Figure 18.b Data mapper does all the data mapping and TDG only runs queries.

4.4 Change Association Refactoring strategy

Initially we had visibility from Applicationdriver.java to ConnectionToSql.java, but with changing project this was not needed anymore so we removed this visibility.

Before Refactoring:



```java
package driver_of_application;

import java.sql.*;
import connection.ConnectionToSQL;
import data_source.*; // Package made by the us
import java.util.Scanner;

public class ApplicationDriver {

    public static void main(String[] args) {

        ConnectionToSQL instanceOfConnectionToSQL = ConnectionToSQL.getInstanceToSQL();
        Scanner objectScanner = new Scanner(System.in);

        while (true) { // infinite loop

            System.out.println("What do you want to do(Please choose the respective option number)?"); // main menu options
            System.out.println("\t1. View table data");
            System.out.println("\t2. Insert into table");
            System.out.println("\t3. Quit");
            System.out.println("Please enter your choice >");
            String inputUser = objectScanner.nextLine();
```

Figure 19.a Unnecessary association is present in ApplicationDriver.java

After Refactoring

```java
* Class Name:- Application Driver(contains main function)

package driver_of_application;

import java.util.Scanner;

public class ApplicationDriver {


    private static DataTableSubject objectDataTableSubject = DataTableSubject.getDataTableSubjectInstance();;
    public static void main(String[] args) {

        Scanner objectScanner = new Scanner(System.in);
        String inputUser;
        String inputTableName;
```

Figure 19.b only necessary association is present in ApplicationDriver.java

In Datamapper.java and DataTableSubject.java both required visibility of each other initially this was not the case but later after refactoring both were mad bi-directional.

4.5 Conditional Refactoring strategy

Initially we had if else statements in the ApplicationDriver.java which were redundant and giving the same output hence we merged them:

Before Refactoring:

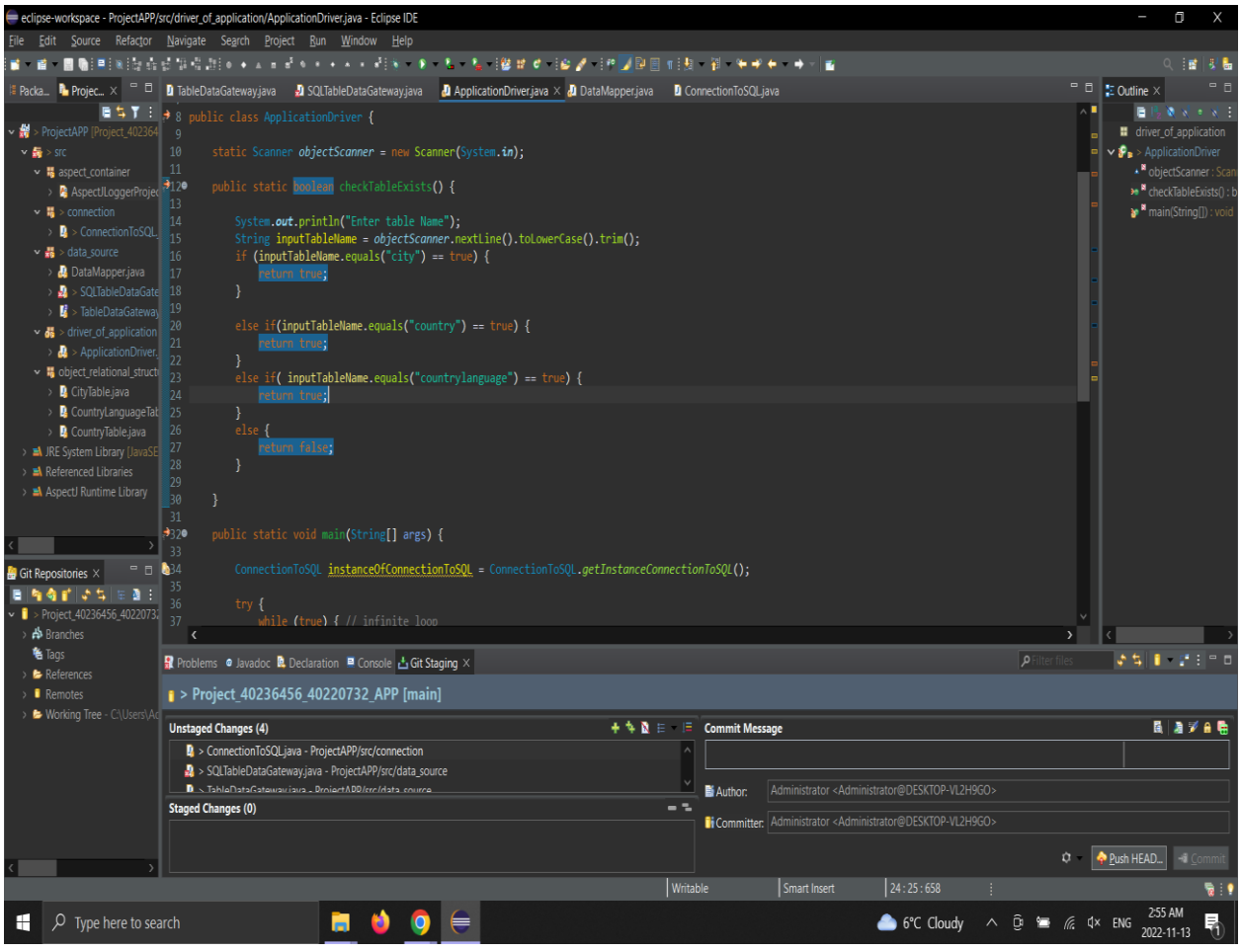
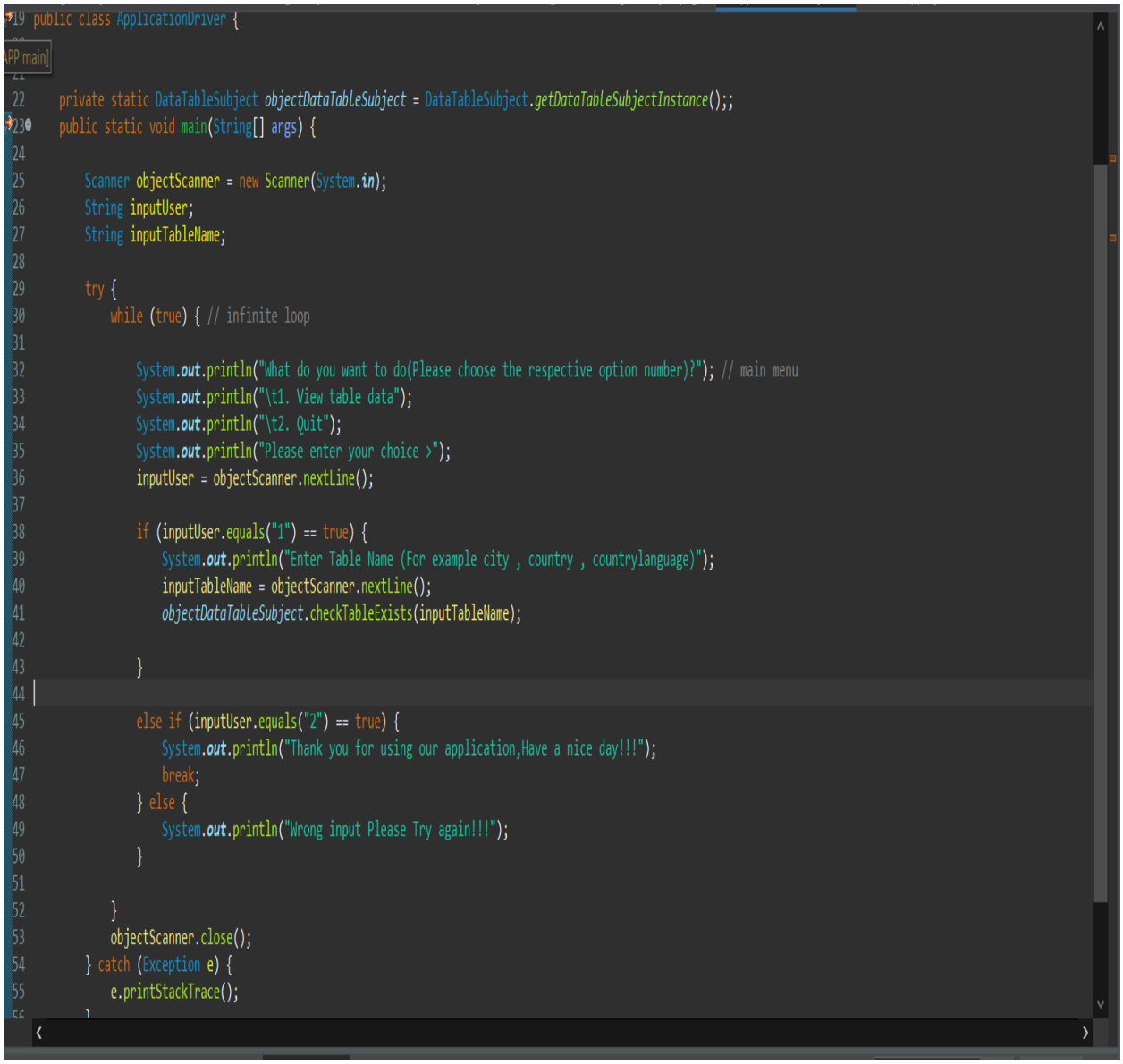


Figure 20.a Unnecessary if-else statements

After Refactoring:



```
19 public class ApplicationDriver {

APP main()

22     private static DataTableSubject objectDataTableSubject = DataTableSubject.getDataTableSubjectInstance();;
23     public static void main(String[] args) {
24
25         Scanner objectScanner = new Scanner(System.in);
26         String inputUser;
27         String inputTableName;
28
29         try {
30             while (true) { // infinite loop
31
32                 System.out.println("What do you want to do(Please choose the respective option number)?"); // main menu
33                 System.out.println("\t1. View table data");
34                 System.out.println("\t2. Quit");
35                 System.out.println("Please enter your choice >");
36                 inputUser = objectScanner.nextLine();
37
38                 if (inputUser.equals("1") == true) {
39                     System.out.println("Enter Table Name (For example city , country , countrylanguage)");
40                     inputTableName = objectScanner.nextLine();
41                     objectDataTableSubject.checkTableExists(inputTableName);
42
43                 }
44
45                 else if (inputUser.equals("2") == true) {
46                     System.out.println("Thank you for using our application,Have a nice day!!!");
47                     break;
48                 } else {
49                     System.out.println("Wrong input Please Try again!!!");
50                 }
51
52             }
53             objectScanner.close();
54         } catch (Exception e) {
55             e.printStackTrace();
56         }
```

Figure 20.b only required if-else statements.

4.6 Pull up Refactoring strategy

Changing of scope for variables from local to global because they were being reused again across different methods.

Before Refactoring:

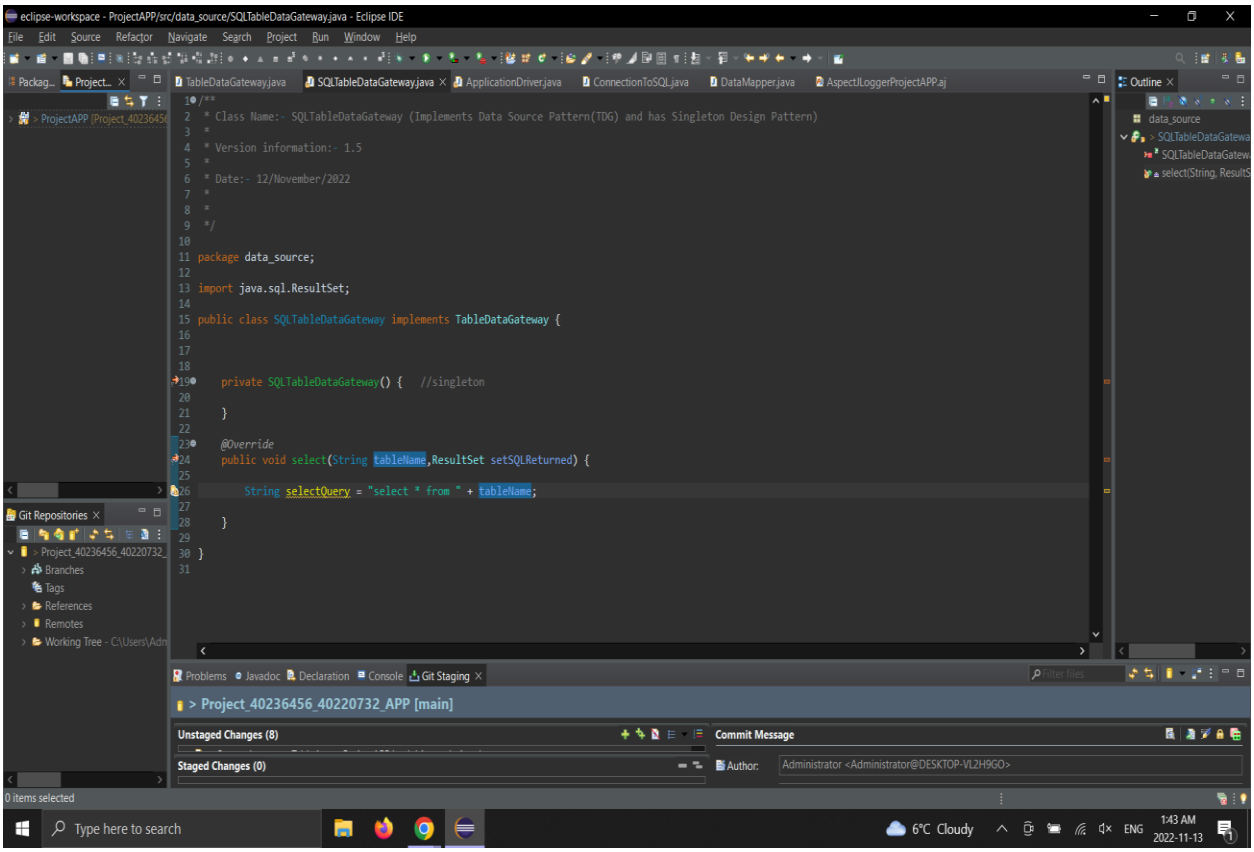


Figure 21.a Local variables being used.

After Refactoring:

```
package data_source;

import java.sql.ResultSet;

public class SQLTableDataGateway implements TableDataGateway {

    private String tableName;
    private ResultSet resultSetSQLReturned;
    private Statement statementOfSQLDataBase;

    private static SQLTableDataGateway instanceSQLTableDataGateway = null;
```

Figure 21.b global variables being used which promote code reusability.

5 Testing

Junit4 testing tool is used to test our java code. Integration and unit are the two techniques we have implemented.

5.1 Unit Testing

Unit testing is the type of software testing where we can do individual testing of classes and methods. In our project, Unit testing is done for different classes and methods as shown below:

5.1.1 CountryTableTest.java

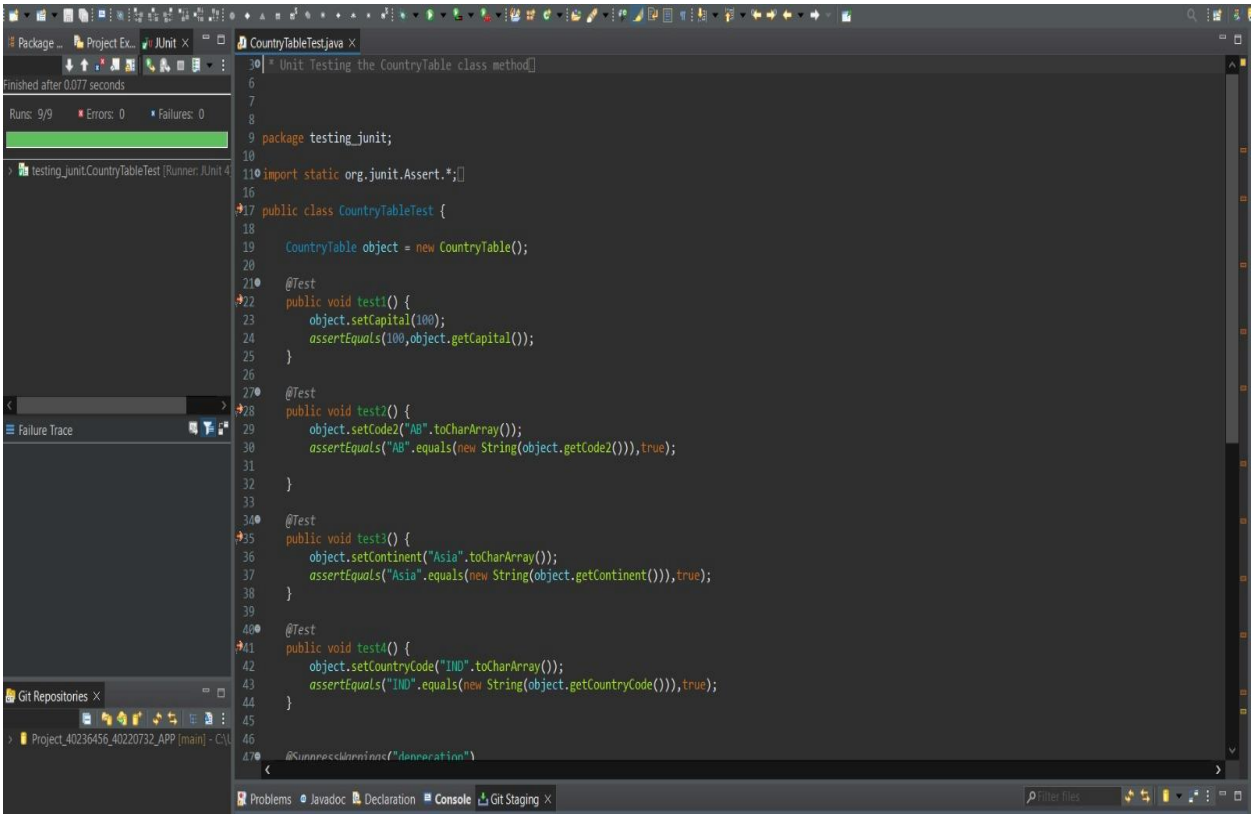
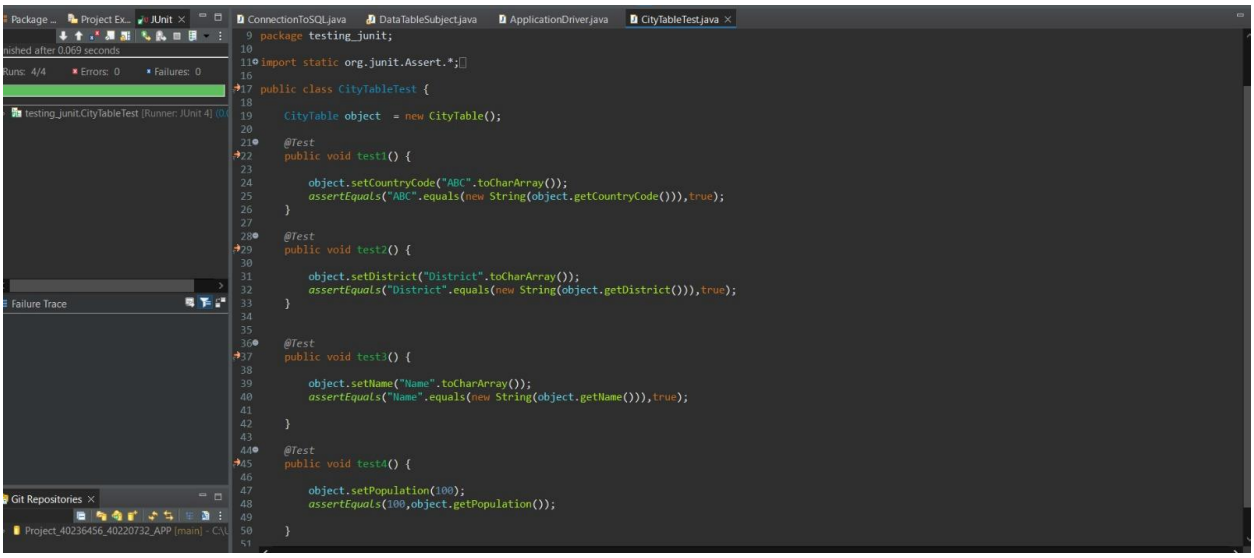


Figure 22: Unit Testing for methods inside the CountryTable class and all the test cases was successful.

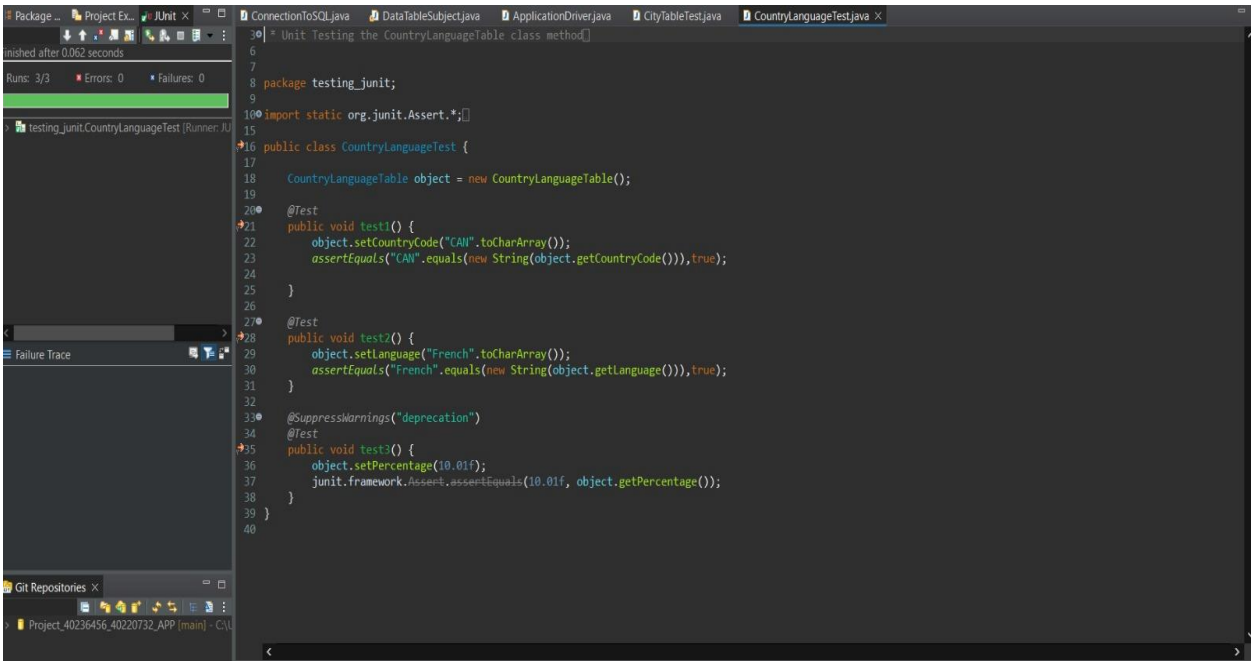
5.1.2 CityTableTest.java



```
package testing_junit;

import static org.junit.Assert.*;

public class CityTableTest {

    CityTable object  = new CityTable();

    @Test
    public void test1() {

        object.setCountryCode("ABC".toCharArray());
        assertEquals("ABC".equals(new String(object.getCountryCode())),true);
    }

    @Test
    public void test2() {

        object.setDistrict("District".toCharArray());
        assertEquals("District".equals(new String(object.getDistrict())),true);
    }


    @Test
    public void test3() {

        object.setName("Name".toCharArray());
        assertEquals("Name".equals(new String(object.getName())),true);

    }

    @Test
    public void test4() {

        object.setPopulation(100);
        assertEquals(100,object.getPopulation());

    }
```

Figure 23: Unit Testing for methods inside the CityTable class and all the test cases was successful.

5.1.3 CountryLanguageTest.java



```
/* Unit Testing the CountryLanguageTable class method

package testing_junit;

import static org.junit.Assert.*;

public class CountryLanguageTest {

    CountryLanguageTable object = new CountryLanguageTable();

    @Test
    public void test1() {
        object.setCountryCode("CAN".toCharArray());
        assertEquals("CAN".equals(new String(object.getCountryCode())),true);

    }

    @Test
    public void test2() {
        object.setLanguage("French".toCharArray());
        assertEquals("French".equals(new String(object.getLanguage())),true);
    }

    @SuppressWarnings("deprecation")
    @Test
    public void test3() {
        object.setPercentage(10.01f);
        junit.framework.Assert.assertEquals(10.01f, object.getPercentage());
    }
}
```

Figure 24: Unit Testing for methods inside the CountryLanguage class and all the test cases was successful.

5.1.4 SQLTableDataGatewayTest.java

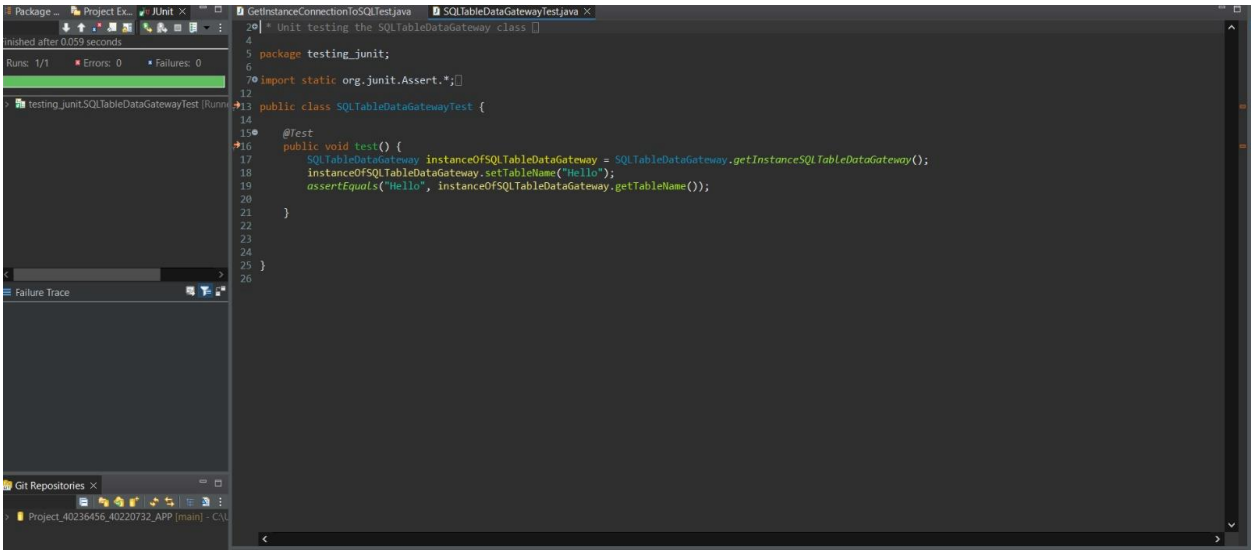


Figure 25: Unit Testing for methods inside the SQLTableDataGateway class and all the test cases was successful.

5.2 Integration Testing

Testing of more than one module is called integration testing and we have performed for the following classes: -

5.2.1 DataTableSubjectTestIntegration.java

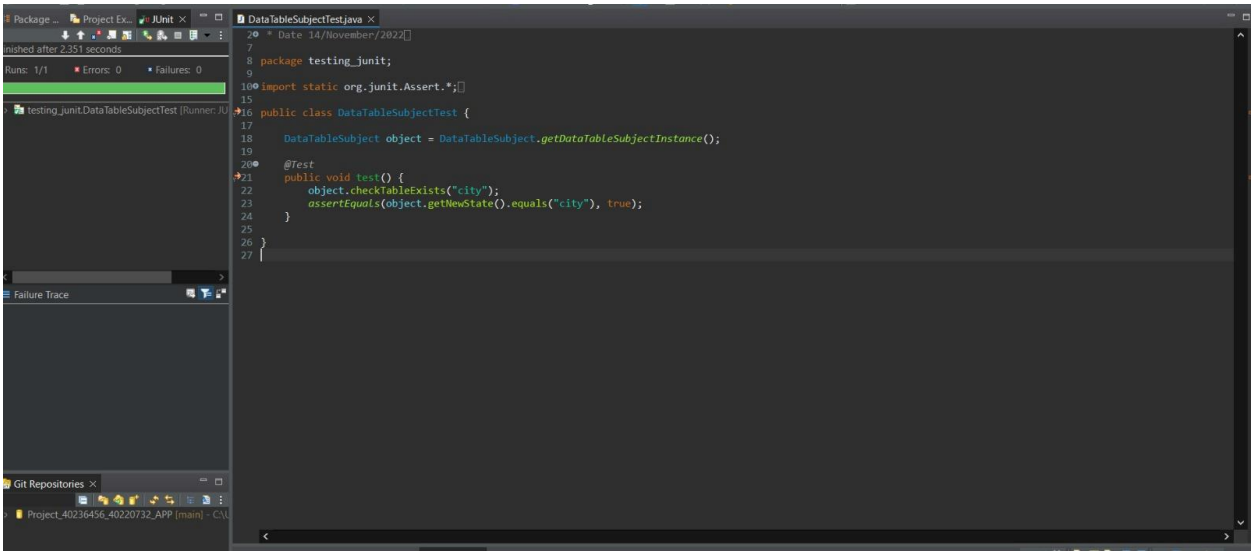


Figure 26: Integration Testing for methods inside the DataTableSubject class and all the test cases was successful.

5.2.2 GetInstanceConnectionToSQLTestIntegration.java

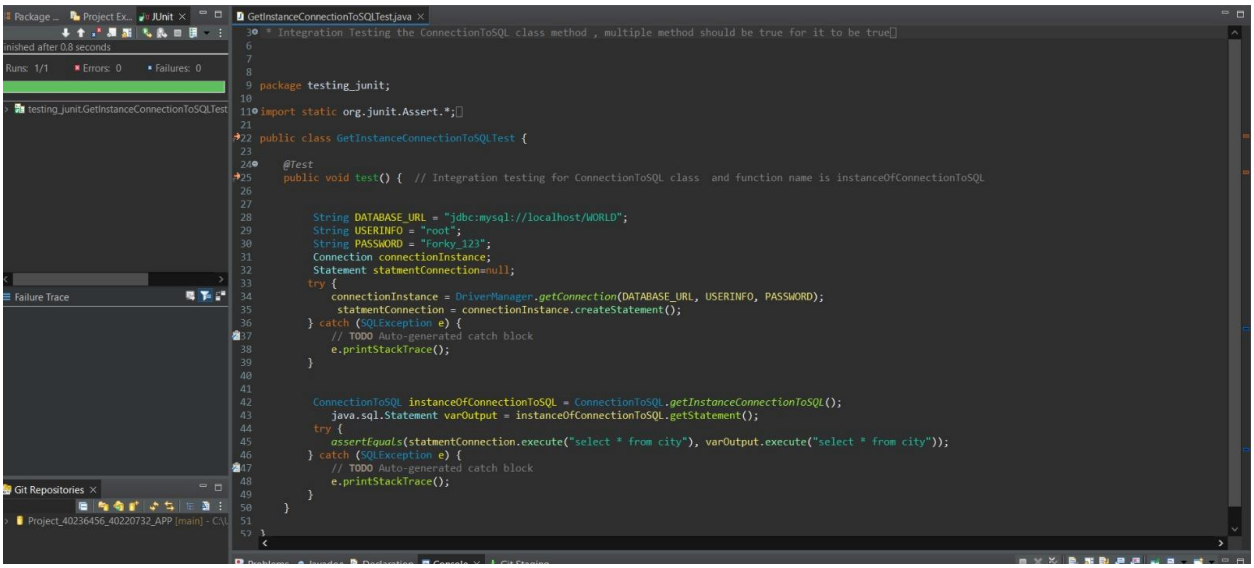


Figure 27: Integration Testing for methods inside the GetInstanceConnectionToSQL class and all the test cases was successful.

6 Architecture

For theory of architecture, read section 1,2 and 3. The architecture of our application can visualize using the given below diagrams:

6.1 ER -Diagram

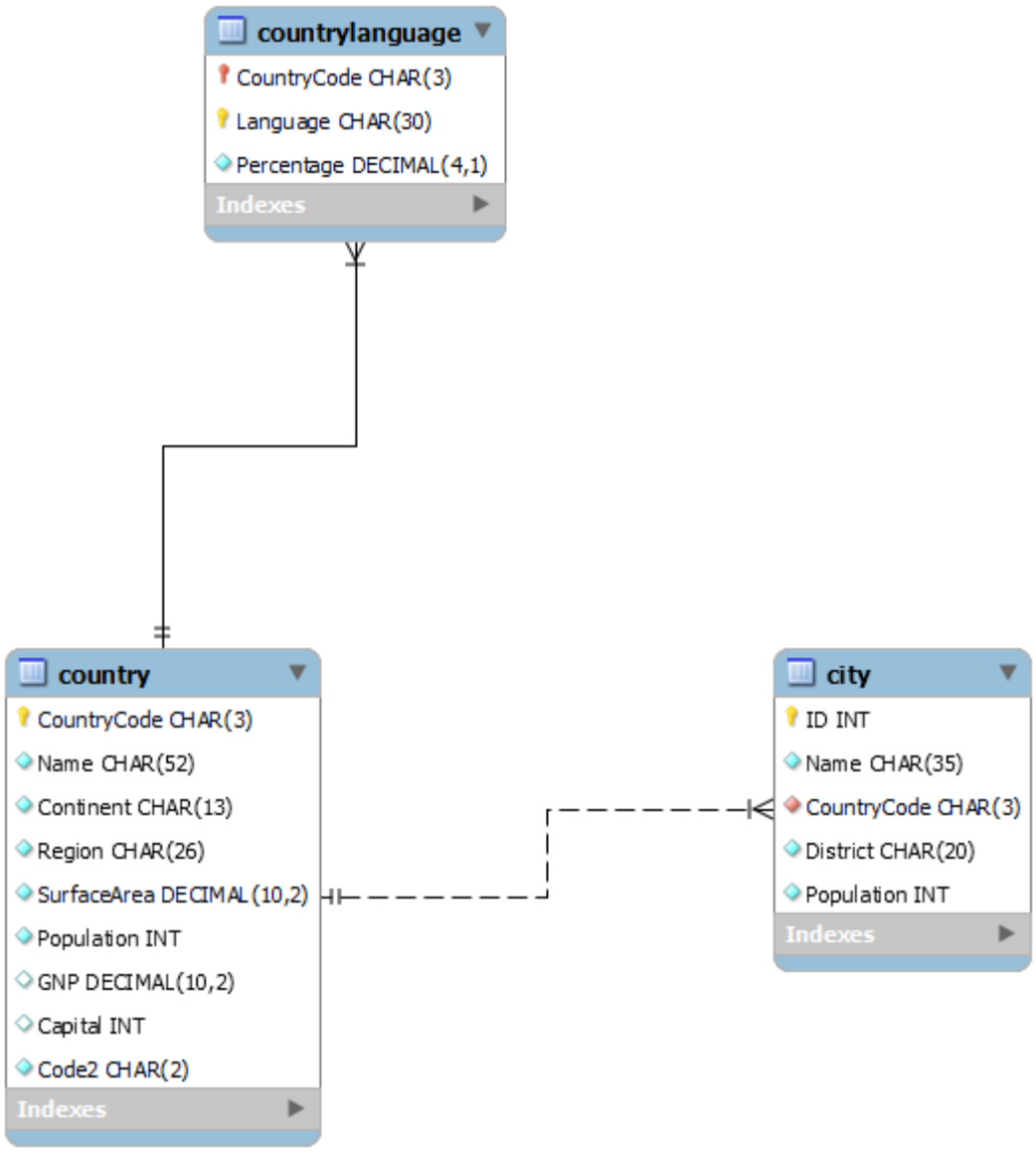


Figure 28: Entity Relationship Diagram

6.2 Class Diagram

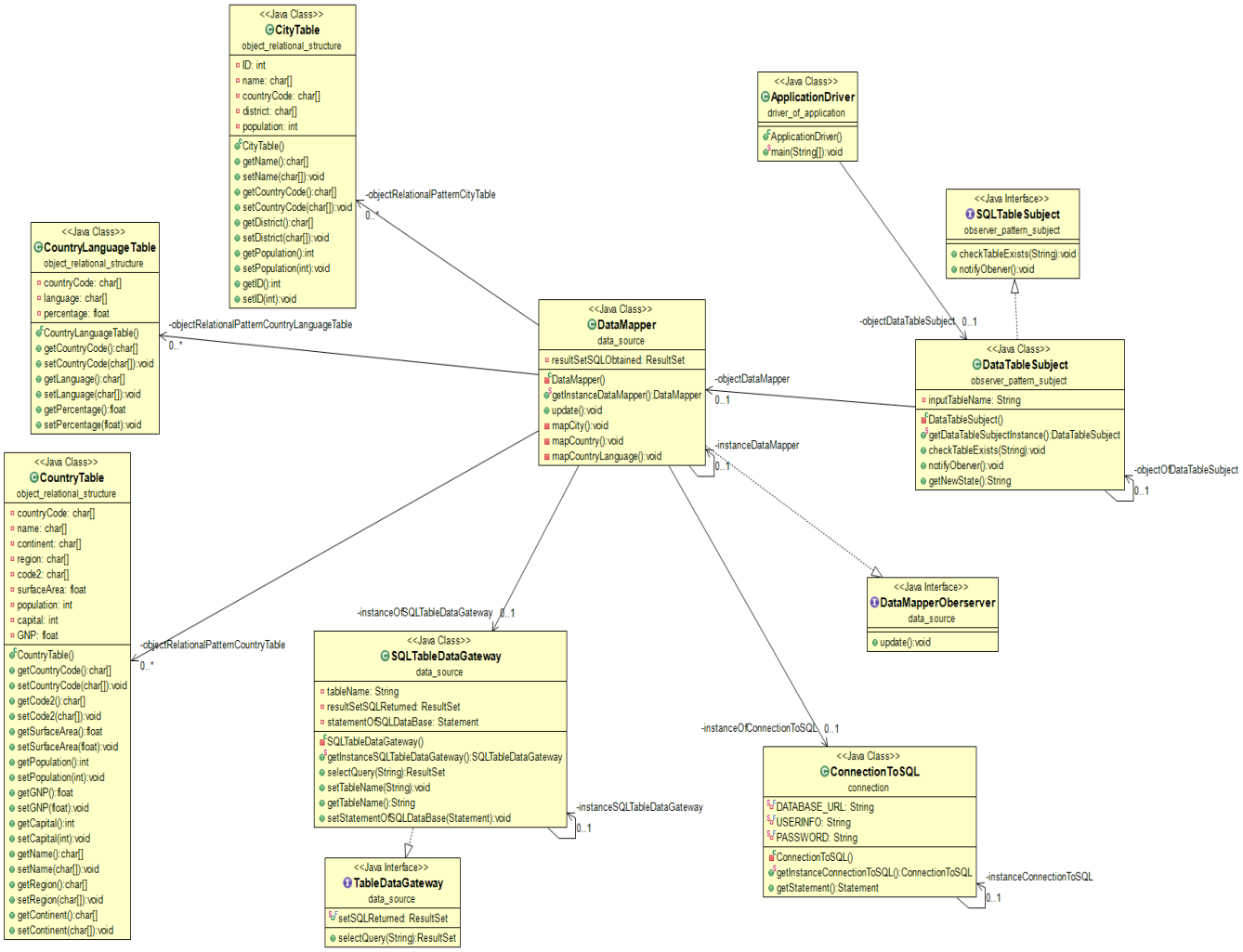


Figure 29: Class Diagram

6.3 Sequence Diagram

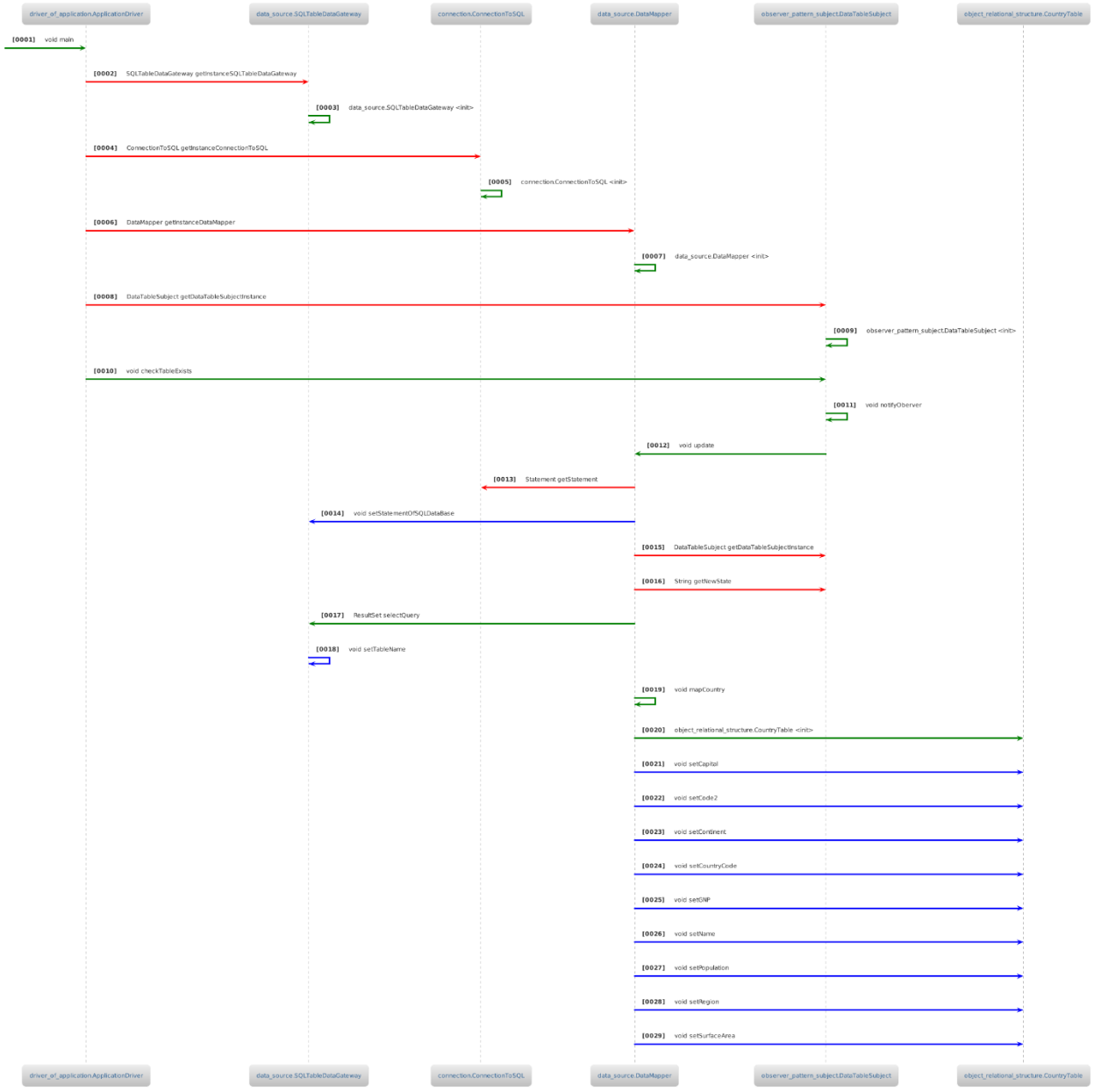


Figure 30: Sequence Diagram (using AspectJ and PlantUML) of our application while accessing only one record from database.

7 References

1) https://plantuml.com/
2) https://www.mysql.com/
3) https://www.oracle.com/java/technologies/javase/codeconventions-contents.html
4) https://www.tutorialspoint.com/java_mysql/index.htm
5) https://marketplace.eclipse.org/content/objectaid-uml-explorer

8 Note for Reader

GitHub: - https://github.com/Vasudev-Sharma-13/Project_40236456_40220732_APP/

(GitHub is open to public so no username and password)

To install the project, follow these steps: -

1)Download project and database from GitHub

2)Import in eclipse

3)Add external Jar files from internet to your project: - AspectJ, JDBC driver, Junit4, Objectaid.

4)make changes to AspectJLoggerProjectAPP.aj and ConnectionToSQL.java file for path and user details.

5)Run the application.