Design Level Refactoring

Refactor #1

In the third iteration of our project, a significant enhancement was made by introducing an interface for our Data Transfer Objects (DTOs). These DTOs, which include various system objects such as orders, reservations, customer accounts, and receipts, now follow a newly created interface. This improvement started with the creation of an interface named "QueryGenerator." This interface has two essential methods: "generateInsertQuery" and "generateUpdateQuery." All DTOs are now required to implement this interface, ensuring a standardized implementation of these methods. These methods are important for our database interaction, enabling the generation of insert and update queries.

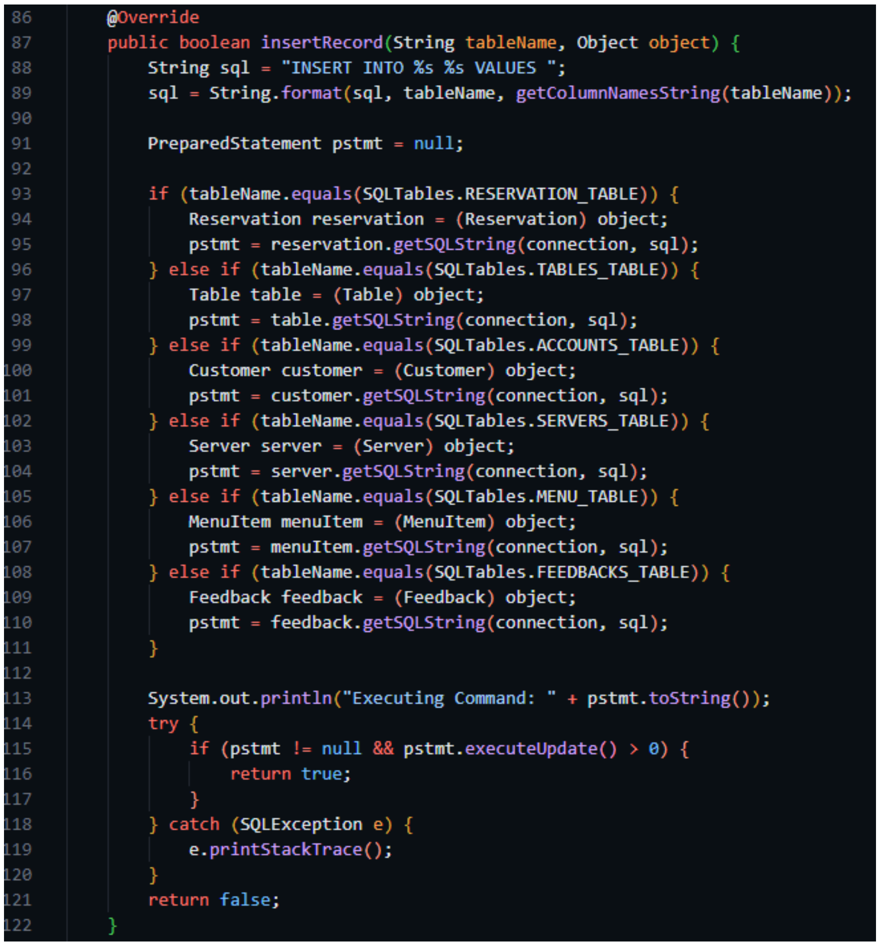
This approach enhances scalability by eliminating the need for conditional statements to determine the relevant database table and class, as previously encountered. The implementation simplifies code maintenance and scalability by replacing extensive conditional logic with a straightforward method call to "generateInsertQuery" or "generateUpdateQuery," depending on the operation. This method dynamically identifies and executes the appropriate query for the given object, thereby streamlining database interactions.

The "QueryGenerator" interface includes methods for creating insert and update statements, as demonstrated in the "QueryGenerator.java" class with the following signatures:

```
public PreparedStatement generateInsertStatement(Connection conn,
List<String> columns);
public PreparedStatement generateUpdateStatement(Connection conn,
List<String> columns);
```

These changes are integrated into our database implementation, as shown in the "DataBaseImpl.java" class, illustrating a clear before-and-after comparison of our approach to database interactions, significantly contributing to the project's scalability and maintainability.

Before



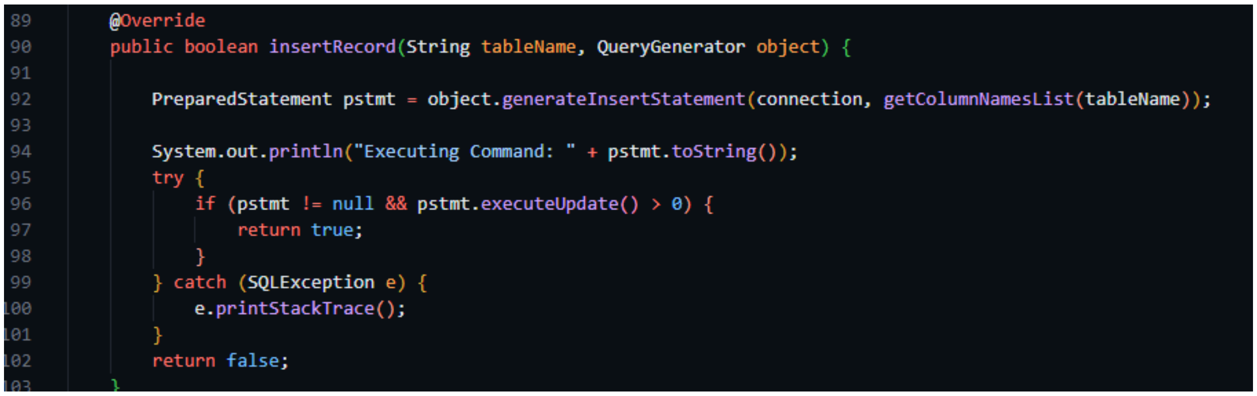
```java
    @Override
    public boolean insertRecord(String tableName, Object object) {
        String sql = "INSERT INTO %s %s VALUES ";
        sql = String.format(sql, tableName, getColumnNamesString(tableName));

        PreparedStatement pstmt = null;

        if (tableName.equals(SQLTables.RESERVATION_TABLE)) {
            Reservation reservation = (Reservation) object;
            pstmt = reservation.getSQLString(connection, sql);
        } else if (tableName.equals(SQLTables.TABLES_TABLE)) {
            Table table = (Table) object;
            pstmt = table.getSQLString(connection, sql);
        } else if (tableName.equals(SQLTables.ACCOUNTS_TABLE)) {
            Customer customer = (Customer) object;
            pstmt = customer.getSQLString(connection, sql);
        } else if (tableName.equals(SQLTables.SERVERS_TABLE)) {
            Server server = (Server) object;
            pstmt = server.getSQLString(connection, sql);
        } else if (tableName.equals(SQLTables.MENU_TABLE)) {
            MenuItem menuItem = (MenuItem) object;
            pstmt = menuItem.getSQLString(connection, sql);
        } else if (tableName.equals(SQLTables.FEEDBACKS_TABLE)) {
            Feedback feedback = (Feedback) object;
            pstmt = feedback.getSQLString(connection, sql);
        }

        System.out.println("Executing Command: " + pstmt.toString());
        try {
            if (pstmt != null && pstmt.executeUpdate() > 0) {
                return true;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }
```

After



```java
    @Override
    public boolean insertRecord(String tableName, QueryGenerator object) {

        PreparedStatement pstmt = object.generateInsertStatement(connection, getColumnNamesList(tableName));

        System.out.println("Executing Command: " + pstmt.toString());
        try {
            if (pstmt != null && pstmt.executeUpdate() > 0) {
                return true;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }
```

Refactor #2

The second major change we implemented was breaking down a large class into several smaller ones, specifically the ServiceUtil.java class. Initially, in our project, this class served multiple purposes, handling both server and table management tasks. At first, we thought it wasn't necessary to have separate classes for these tasks. However, by the second phase of our project, we realized that ServiceUtil was becoming too big and difficult to manage.

As a result, we decided to remove ServiceUtil completely and introduce two new classes: ServersService and TableService. The goal was to make the structure more organized by dividing responsibilities into more specific classes. This change made things much simpler to manage, almost like sorting items into their own compartments.

Another significant advantage of this approach was the improvement it brought to testing. Instead of dealing with a single, large JUnit test file, we were able to create individual test files for each class. This made the testing process much easier, as it eliminated the need to sift through a lot of code to locate the specific functions we needed to test.

As seen on the link below

https://github.com/Pouya-Sameni/PlatePlan/blob/ITR1/PlatePlan/src/misc/ServiceUtils.java

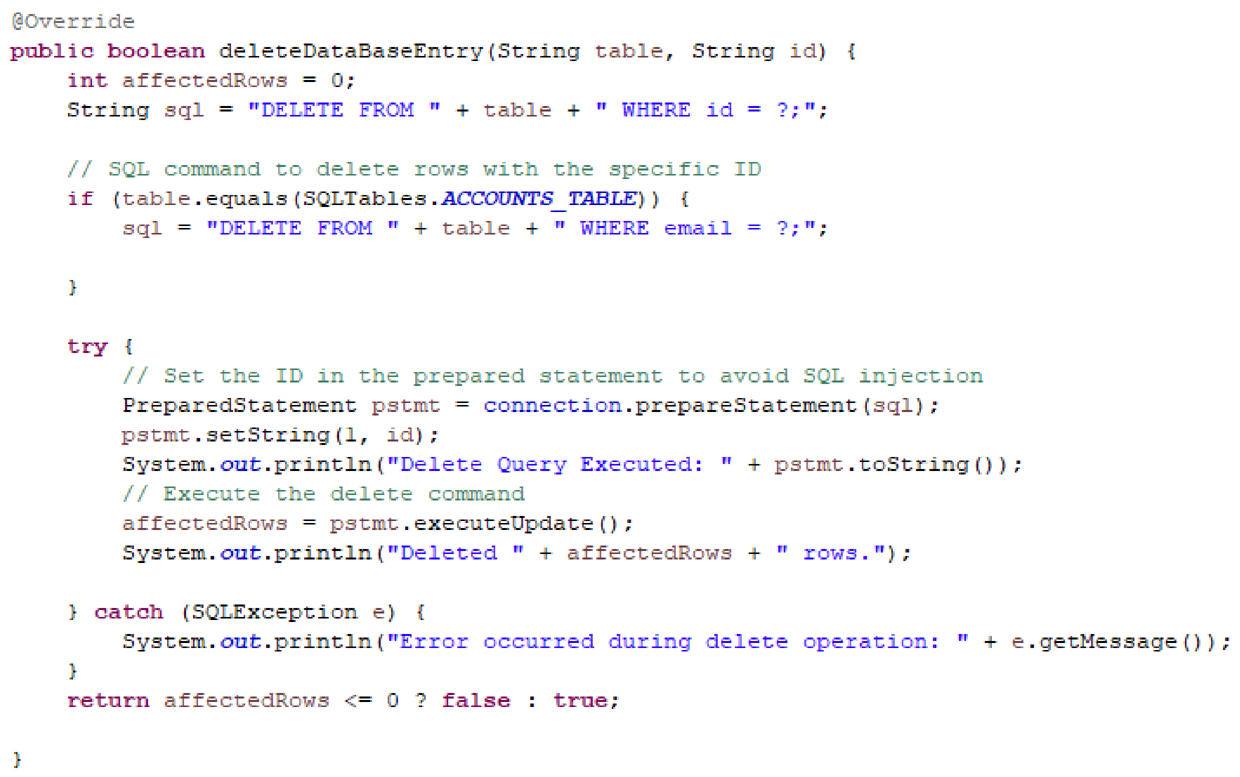
Our first iteration had a larger ServiceUtil class without a clear goal. This is now changed to the following two service interfaces.

Tables Service:
https://github.com/Pouya-Sameni/PlatePlan/blob/ITR3/PlatePlan/src/service_interfaces/TablesService.java

Servers Service:
https://github.com/Pouya-Sameni/PlatePlan/blob/ITR3/PlatePlan/src/service_interfaces/ServerService.java

Refactor #3

The third significant improvement we made was during iteration 2. We noticed that as our collection of objects and tables grew, we constantly had to create methods for deleting entries from these tables. However, all our tables had a similar structure with a unique ID column. So, we combined these delete methods into one. Now, in future iterations, we only need to pass the ID and table name to delete an entry. This streamlined our database interface and simplified our code, reducing it to a single method. The images below show the before and after of our interface and the code changes.



```java
@Override
public boolean deleteDataBaseEntry(String table, String id) {
    int affectedRows = 0;
    String sql = "DELETE FROM " + table + " WHERE id = ?;";

    // SQL command to delete rows with the specific ID
    if (table.equals(SQLTables.ACCOUNTS_TABLE)) {
        sql = "DELETE FROM " + table + " WHERE email = ?;";

    }

    try {
        // Set the ID in the prepared statement to avoid SQL injection
        PreparedStatement pstmt = connection.prepareStatement(sql);
        pstmt.setString(1, id);
        System.out.println("Delete Query Executed: " + pstmt.toString());
        // Execute the delete command
        affectedRows = pstmt.executeUpdate();
        System.out.println("Deleted " + affectedRows + " rows.");

    } catch (SQLException e) {
        System.out.println("Error occurred during delete operation: " + e.getMessage());
    }
    return affectedRows <= 0 ? false : true;

}
```

Additionally, we observed that when fetching data from the database, we repeatedly had to set certain fields for each object type. For instance, for objects like reservations. To streamline this process, we created a database converters class. This class has methods for converting database responses into Java objects. One method converts a single database response into a Java object, while the other converts query responses into lists of Java objects. This approach made it easier to convert PostgreSQL database responses into Java objects. See the implementation of our database converters below.