CSCI5308 - ASSIGNMENT 3

Dhrumil Amish Shah (B00857606)
dh416386@dal.ca

CSCI5308 Assignment 3

Task 1 - Three SOLID principles applied in project Blood Book

1. **Single Responsibility Principle(SRP)**
   The Single Responsibility Principle(SRP) states that a class should only have a single(one) responsibility. Having a class with only one responsibility have the following advantages:
   - Tests are more focused on business logic and are easy to implement.
   - Classes are organized in a well-defined manner and, the searching becomes easy.
   - Lower coupling leads to better maintainability since there are fewer dependencies to manage.

   A description of the functionality or part of the code where this principle is applied (method, class or statement)
   - The Single Responsibility Principle(SRP) is applied to all the classes in the **src/main/java/backend** package of the Blood Book application. For better understanding, consider any class in any **model** package. The responsibility of that class in the model package is to store information, particularly to the task of that feature.
   - For example,
     - **User** class in package **backend.authentication.user.model** stores information related to the user using the Blood Book application
     - **BloodBank** class in package **backend.authentication.blood_bank.model** stores information related to the blood bank using the Blood Book application.
     - **BloodStock** class in package **backend.blood_stock_management.model** stores information related to the bloodstock of the logged-in blood bank.

   A description of why such SOLID principle was applied in this part of the code (method, class, or statement).
   - The purpose of having model classes is to simplify the data transfer inside the application (i.e., between controller and view).
   - For example,
     - When a user logs in to the application using the correct credentials, user data is stored inside a ResultSet object. Passing the ResultSet object to the frontend creates a resource leak if the ResultSet is not closed. Also, accessing any individual entry from the ResultSet instance is difficult than accessing it from the model class. The model class **User** is dedicated to handling user-related data to ease data access inside the application.

   A description of what problems would be triggered in your project if the SOLID principle is not applied.
   - Below are the problems that would occur if the Single Responsibility Principle is not applied to the application.

- It would be difficult to maintain the entire application because of higher coupling between the various components of the class and with the other classes of the application.
- Testing would be tedious and, the number of test cases for the class would be much larger than in a class implementing the Single Responsibility Principle.

2. Interface Segregation Principle(ISP)

The Interface Segregation Principle(ISP) states that a larger interface(i.e., interface with many methods) should be split into multiple smaller interfaces. ISP ensures that the implementing classes are only concerned about the methods that they are interested in.

A description of the functionality or part of the code where this principle is applied (method, class or statement)

- The Interface Segregation Principle(ISP) is applied to all the classes in the **controller** packages of the Blood Book application.
- For example,
  - The Blood Book application provides two ways to log in for users and blood banks (i.e., by email and contact number). Instead of having a single interface with two functionalities listed, two separate interfaces are created and implemented by respective controllers. The loginWithEmail method is defined in UserLoginWithEmailControllerDAO and BloodBankLoginWithEmailControllerDAO interfaces whereas loginWithContactNumber method is defined in UserLoginWithContactNumberControllerDAO and BloodBankLoginWithContactNumberControllerDAO interfaces. Thus, at any point, if we decide to have only one login functionality (i.e., by email or by contact number), we can simply implement only the relevant interface. Moreover, the Interface Segregation Principle implementation also provides the flexibility of having two different login methods for user and blood banks as we have separate interfaces for both of them. For instance, email login for user and contact number login for the blood bank.

A description of why such SOLID principle was applied in this part of the code (method, class, or statement).

- The purpose of having separate interfaces is that we are free to implement the functions that matter to us. Thus the implementing classes are only concerned about the methods that are of interest to them.
- For example,
  - The UserLoginWithEmailController class cares about user login with email. Hence it implements the UserLoginWithEmailControllerDAO interface only.
  - The UserLoginWithContactNumberController class cares about user login with contact number. Hence it implements the UserLoginWithContactNumberControllerDAO interface only.
  - The BloodBankLoginWithEmailController class cares about blood bank login with email. Hence it implements the BloodBankLoginWithEmailControllerDAO interface only.

- The BloodBankLoginWithContactNumberController class cares about blood bank login with contact number. Hence it implements the BloodBankLoginWithContactNumberControllerDAO interface only.

A description of what problems would be triggered in your project if the SOLID principle is not applied.

- Below are the problems that would occur if the Interface Segregation Principle is not applied to the application.
  - The classes would be forced to mandatorily implement methods that are not relevant to their primary function.
  - The classes would be required to throw NotImplementedException for unused methods.
  - The application will be coupled tightly with many classes dependent on each other.

3. Dependency Inversion Principle (DIP)

The Dependency Inversion Principle refers to the fact that modules depend on abstractions instead of concrete implementation. Following are the advantages of the Dependency Inversion Principle.

- Low coupling between multiple modules and within the same module.
- Reduce dependencies within the project.
- Modifications are easy to perform.

a. A description of the functionality or part of the code where this principle is applied (method, class or statement)

- The Dependency Inversion Principle(DIP) is applied to all concrete classes in **controller** packages of the Blood Book application.
- For example,
  - Consider the BloodBankLoginWithContactNumberController class. The parameterized constructor of the class is passed the DatabaseConnectionDAO which is an interface and not the actual DatabaseConnection implementation. Further, I am not instantiating the DatabaseConnection class in the BloodBankLoginWithContactNumberController class. So at any point in future, if I decide to add a new database configuration that connects to a different database, I can do so by creating a new class that implements the same DatabaseConnectionDAO interface and pass it as the constructor argument. Thus, I don't have to change anything in the BloodBankLoginWithContactNumberController class and, it resolves the dependency between BloodBankLoginWithContactNumberController and DatabaseConnectionDAO. In other words, I can pass any database connection implementation of the DatabaseConnectionDAO interface with no modification in the BloodBankLoginWithContactNumberController class.

b. A description of why such SOLID principle was applied in this part of the code (method, class, or statement).
   - The purpose of implementing Dependency Inversion Principle(DIP) for this feature is to remove coupling(dependency) between BloodBankLoginWithContactNumberController and DatabaseConnection classes. If instead of passing the interface DatabaseConnectionDAO, the implementation class is instantiated, and at any point in future, if we decide to change the database configuration or use a completely new database instance, then it will require heavy modifications in the BloodBankLoginWithContactNumberController class.
   - Applying the DIP principle for this feature provides flexibility in modification of the program for any enhancements proposed for the project.

c. A description of what problems would be triggered in your project if the SOLID principle is not applied.
   - Below are the problems that would occur if the Interface Segregation Principle is not applied to the application.
     - Higher dependency between various components of the project leads to higher coupling and thereby leads to complex implementation.
     - Increases the maintenance load of the application with the project components tightly coupled with each other.

Task 2 – Project structure

The Blood Book application is implemented primarily using three layers – frontend, backend and database and it follows the Model-View-Controller(MVC) architecture. The frontend package covers the user interface for all the users of the application, the backend deals with the business logic related to all the features of the application and, the database handles database connection required for establishing database-related connectivity and dealing with database related operations.

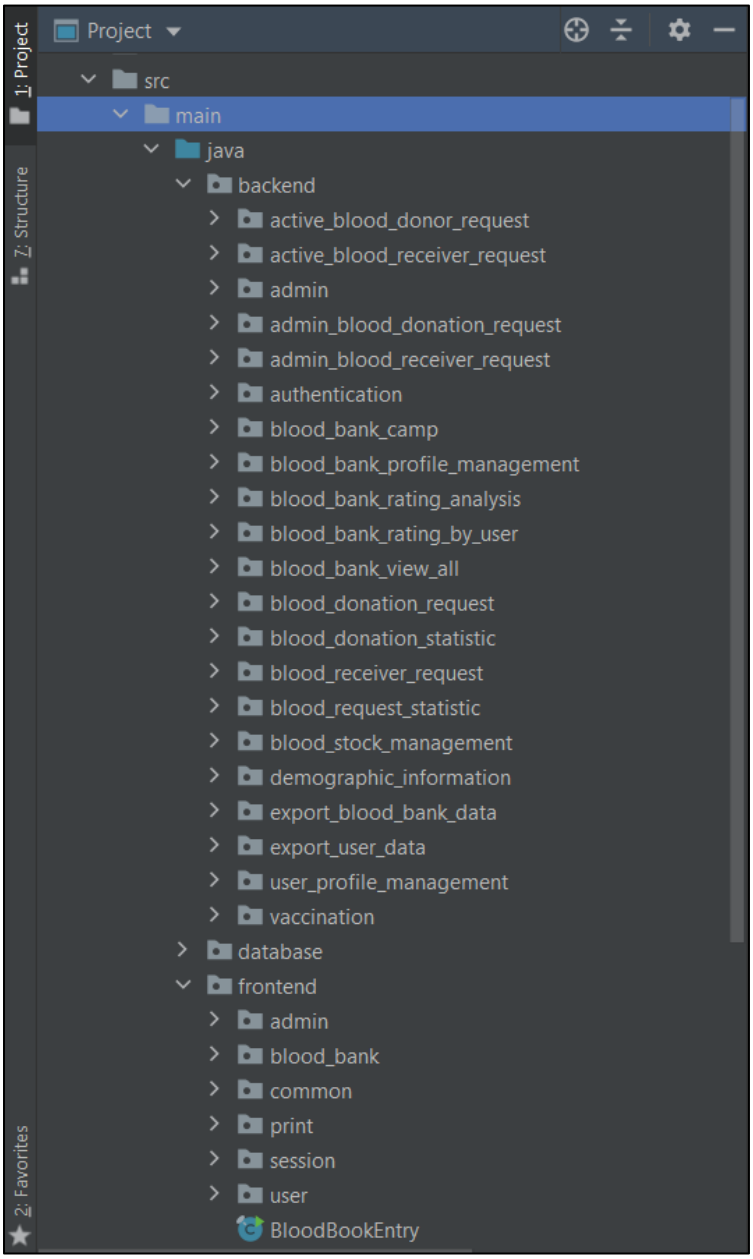

Figure 1 Project structure

1. Presentation Layer (Frontend) – View

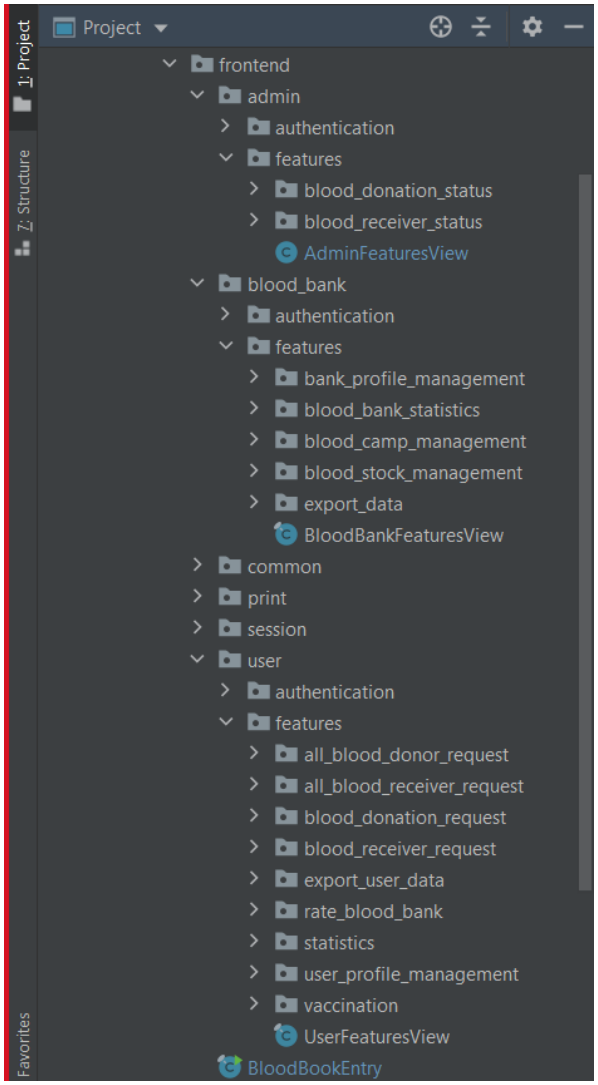

Figure 2 Frontend (Presentation layer) of the Blood Book application

The starting point of the application is BloodBookEntry class which consist of the main method. Once the user, blood bank or admin logs in successfully the session is stored inside the Session class in the session package. Depending on the user profile logging in, the features will be displayed for the respective users. All the methods dealing with printing statements on the console are placed under print package's BloodBookPrinter class. The common package shares all the view files common to all the three user profiles – users, blood banks and admin. The frontend part is the View part of the MVC architecture that is followed for all features of the application. Each View class is responsible for showing its own feature. For example, BloodStockManagementView for the bloodstock management feature, and BloodBankRatingAnalysisStatisticsView for the blood bank rating analysis feature.

2. Business Logic Layer (Backend) – Model and Controller

The **controller** package for every feature of the application deals with the business logic implemented for that feature. For every feature of the application we have followed the same structure wherein each feature has four packages namely **model**, **exception**, **database** and **controller**.

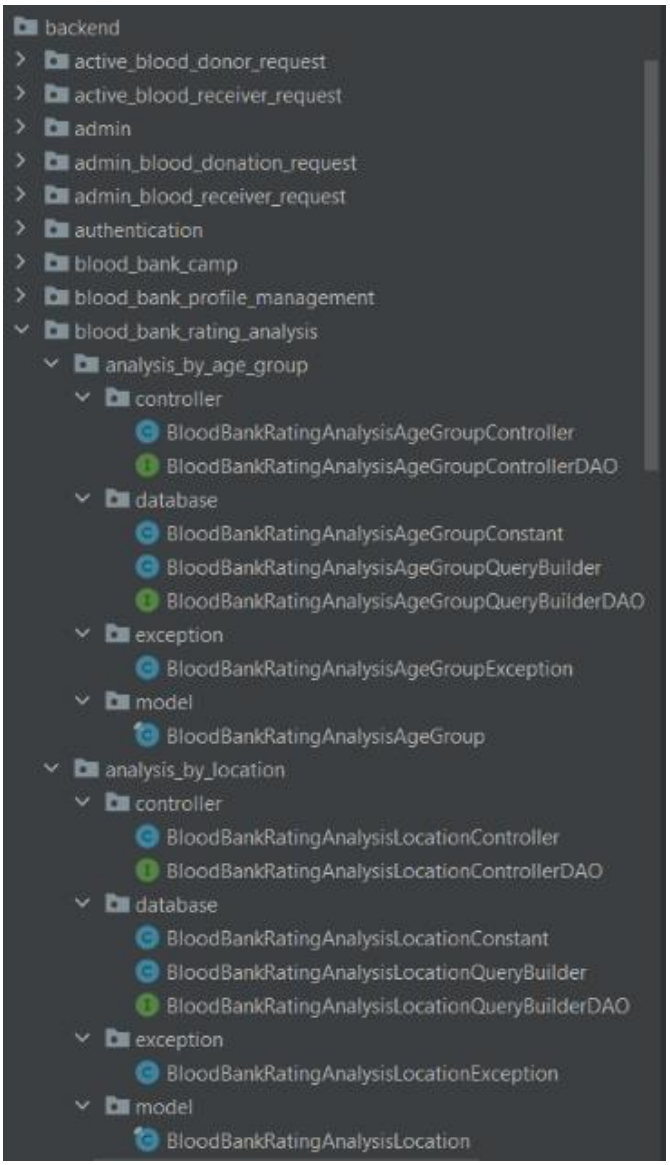

Figure 3 Backend (Business logic layer) of the Blood Book application

The business logic layer can be well explained using the blood bank rating analysis by location example. The controller package consists of all the files covering the business logic for the blood bank rating analysis by location feature. The business logic is applied to the data returned from the JAVA files in the database package. If any error occurs during the execution of the business logic, the custom exception covered in the class placed in the exception package of that feature is thrown. If the business logic is processed without any error, then the data is wrapped in the class file of the model package and transferred to the view which in turn displays it to the user.

3. Database Layer (Database)

The database layer of the project is implemented using three java files namely **DatabaseConnectionException**, **DatabaseConnectionDAO** and **DatabaseConnection** . A brief description of each of the files is given below:

- **DatabaseConnectionException** – This class implements a custom exception for any feature throughout the application. If there is any error related to the database, this class is instantiated with the appropriate error message and thrown to the view where it is displayed to the user.
- **DatabaseConnectionDAO** – This is an interface that is used by the **DatabaseConnection** class to establish a connection with the database for any feature requiring it. This has been designed by keeping in mind the Dependency Inversion Principle. If in future, the database configuration for the application changes, the class can implement the same interface with a different configuration file connecting to another database.
- **DatabaseConnection** – This is a class that gives a provision to every feature of the application to establish database connection using the method getDatabaseConnection(). Additionally, it also provides a method to clear the database connection.

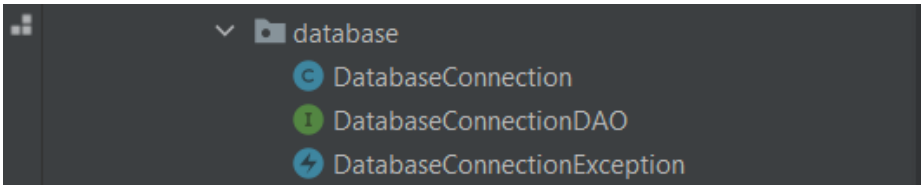

Figure 4 Database layer of the Blood Book application

Also, the **databaseConfig.properties** consists of the configuration details for all three environments of the database (development, test and production). We have used a flag variable which decides the database environment the project will connect to. Thus, a change in environment will require change in value of the flag variable only.

Task 3 – Five Design patterns applied in project Blood Book

1. **Singleton Design pattern**
   Singleton Design means designing a class in a way that the class has only one instance and the instance has global access throughout the project.

   - The singleton design pattern has been implemented for all the QueryBuilder classes of all our project features. Using singleton design pattern allows use of a single instance of that class at runtime. This ensures that multiple queries of the QueryBuilder class are not instantiated for use.

   - Singleton design pattern provides the following benefits in designing the project:
     - It saves memory by creating a single instance of that class.
     - It restricts simultaneous modification or access of that instance at runtime.
     - It improves the performance of the class implementing the singleton design pattern.

   - Not applying singleton design pattern to the project would lead to multiple classes of the project utilizing multiple instances of the QueryBuilder class. This would affect performance of the application and also increase the chances of exceptions at runtime.

2. **Abstract Factory Design pattern**
   The abstract factory design pattern aims at creating abstractions for related or dependent objects without specifying their concrete classes.

   - The abstract factory pattern is implemented for every functionality of the Blood Book project as we have created as many abstractions as possible for every feature. Using the abstract factory design pattern offers the following advantages:
     - It eases the transfer of information and wrapped instances among various components of the project
     - Implements consistency among instances of various classes of the project
     - The project components such as the methods are unaware of the implementations of the abstractions, keeping the code secure.

   - Implementing Abstract Factory Design patterns has the following provisions with respect to the overall performance of the project:
     - Abstract factory design pattern helps to enforce higher security within various modules of the project.
     - Moreover, it eases information transfer and maintenance of the project.
     - It also offers flexibility in any possible enhancements in the project in the near future.

   - If abstract factory pattern is not applied in the project, it would lead to the following issues:
     - The visibility of the program would be increased, thereby risking the security of the Blood Book application

- The implementation would be more complex leading to difficulty in sharing instances and parameters between the various modules of the project.

3. **Mediator pattern**
   Mediator pattern states to encapsulate the communication mechanism between two or more objects in the project.

   - This pattern has been implemented in the BloodBankLoginWithContactNumberController class where the constructor of the class encapsulates the deifinition of how the class will interact with the database component of the project by passing DatabaseConnectionDAO argument to the constructor.

   - This pattern eases the way the database operations are carried out for the class by passing the database connection abstraction as a parameter to the constructor. This offers the following advantages:
     - The classes are loosely coupled as abstraction is passed instead of implementations.
     - There is more security as the implemetation is hidden
     - The project is divided into smaller parts of code segments making it simpler and easy to handle.
     - The code is more flexible for enhancements possible in the near future.

   - If this design pattern is not applied, then the components of the project would be tightly coupled increasing the dependency between them. The project would be more susceptible to leakage of important project resources and wrapped objects.

4. **Decorator Design pattern**
   Decorator Design pattern asks to grant flexibility to add any functionality to the project or any class dynamically.

   - The Decorator design pattern can be applied to classes related to any functionality/feature of the project. This can be done by removing the final keyword from each of the classes, allowing them to be inherited by other classes or any new class that may be required in the future. Considering the scope of the project, we have kept the classes to remain final to restrict inheritance in the project.

   - Applying the Decorator Design pattern can provide the following advantages:
     - It will allow inheritance to various classes in the project.
     - It will allow us to add functionality to the existing application if there is any change or enhancement proposed.
     - It would allow us to have a single class to perform a dedicated task by extending the class instead of adding method or code to the existing classes.

   - Considering the current scope of the project as defined in the project proposal, the classes are designed in a way that they are closed for any extensibility or addition. However,

implementing this design pattern would allow us to maintain the SOLID design principles in the project.

5. **Chain of responsibility pattern**

The chain of responsibility pattern supports loose coupling the caller and receiver components by allowing other instances a chance to handle the request.

- This design pattern has been applied throughout the project as most features are loosely coupled with each other with a limited number of instances, decreasing the overuse of a single instance for multiple tasks. We have also used abstractions to be passed as arguments and not their implementations.
- Using this design pattern to implement the features of the project, provides the following benefits:
    - This enforces appropriate chaining of responsibility between the various layers of the project such as frontend, backend and database. Moreover, as we have followed the Model-View-Controller (MVC) architecture, a proper chaining of responsibilities between the instances of each of these classes for the features has been done.
    - Abstracting the parameters passed enforces security and hides implementation of instances and their methods.
- If we would have not applied this design pattern in the project, the following would have been the disadvantages:
    - Using a single instance of any class to complete multiple tasks throughout a single execution of the program would lead to overuse of the instance and possible **Timeout** errors.
    - Additionally, the program would have been less secure.