Dhrumil Rakesh Shah
B00870600

26th July, 2021

CSCI 5308 – Assignment 3

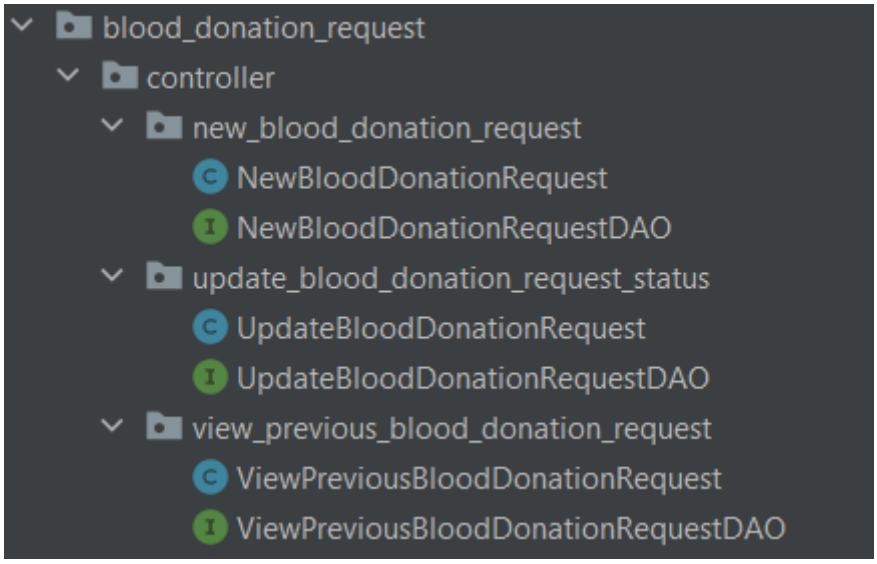
Task 1:

1. **Single Responsibility Principle**
   The very first and one of the most important principle applied in the Group 13 – Blood Book project is the Single Responsibility Principle.

   a. **Description of the functionality where Single Responsibility Principle is used:**
      i. **Blood Donation Request by User**
      The Blood Donation Request feature is a feature for the end user of the Blood Book application. In this feature there are three things that the end user is able to do
         - Create a new blood donation request
         - View all previous blood donation requests of that user
         - Request for blood donation fulfilment

      The above tasks in the blood donation request feature gives the user an end to end option to create a new blood donation request to viewing all the previous requests and also requesting for fulfilment.

      ii. Below is the screenshot of the structure of the classes being divided for each task within a parent package of "**blood_donation_request**".



      iii. As seen in the screenshot, under the **controller** package, there are separate packages for the all three tasks under the **blood donation request** feature.

      iv. Now, as mentioned below all the three classes can be seen. Each class focuses on a single functionality i.e. the **NewBloodDonationRequest** class just covers the code for user creating a new blood donation request whereas the **ViewPreviousBloodDonationRequest** class covers viewing all previous blood donation requests and similarly **UpdateBloodDonationRequest** focuses on updating

the current active requests status to request which is then either fulfilled/rejected by the Admin.

1. **NewBloodDonationRequest.java**
   This class has only one public method named **createNewRequest,** that will let user create a new blood donation request. Now, the **checkLatestRequest** private method is a helper method for the **createNewRequest** as it does the checking whether the current user has any active donation requests or not. If not then the user is allowed to create a new request.

```
public class NewBloodDonationRequest implements NewBloodDonationRequestDAO {

    private boolean checkLatestRequest(int userId) throws BloodDonationRequestException,
    DatabaseConnectionException { }

     @Override

     public boolean createNewRequest(User user, BloodDonationRequest bloodDonationRequest)
    throws BloodDonationRequestException, DatabaseConnectionException { }

}
```

2. **ViewPreviousBloodDonationRequest.java**
   This class has only one public method named **viewBloodDonationRequest**, that will let fetch all the previous blood donation requests of the current user from the database and store it in a list of **BloodDonationRequest** objects. Now, the **prepareBloodDonationRequestList**, is a helper method that is called by the **viewBloodDonationRequest** method passing the fetched result set.

```
public class ViewPreviousBloodDonationRequest implements ViewPreviousBloodDonationRequestDAO {

     private List<BloodDonationRequest> prepareBloodDonationRequestList(final int userId, final
    ResultSet bloodDonationResultSet) throws SQLException { }

     @Override

     public List<BloodDonationRequest> viewBloodDonationRequest(int userId) throws
    BloodDonationRequestException, DatabaseConnectionException { }

}
```

3. **UpdateBloodDonationRequest.java**
   This class has only one public method named **requestFulfilment**, that will ask the user if they wish to initiate a request for fulfilment of their active blood donation request. This class will then update the blood donation request status to **request** (meaning awaiting fulfilment), for the current active request.

```
public class UpdateBloodDonationRequest implements UpdateBloodDonationRequestDAO {

    @Override

    public boolean requestFulfilment(int userId) throws BloodDonationRequestException,
    DatabaseConnectionException { }

}
```

b. <u>Description of why Single Responsibility Principle was applied to this feature:</u>

i. The main reason for applying the Single Responsibility principle of the SOLID principle to the **Blood Donation Request** feature is that, being a single feature, it consists of three different tasks which are connected but have individual importance and complexity. Applying Single Responsibility principle makes the code look clean, easy to understand, separated functionalities for each class makes it easier for modifications. If in future I would need to make some modifications in the create new blood donation request part it would be very easy as I have a separate class for that. Segregating the tasks among different classes makes it easy for debugging as well. Having all the functionalities in a single class makes debugging a nightmare wasting a lot of time and resources.

c. <u>Description of what problems would be triggered in the project if Single Responsibility Principle wouldn't have been applied:</u>

i. The very first and the most crucial problem that would arise if Single Responsibility principle is not used is that code modification would become very difficult. As all different functionalities would be there in the same class, and would increase the code coupling causing issues.
ii. Another problem would be of code maintainability, as there would be a lot of code in a single class that would need to be managed.
iii. Another problem would be of code readability. Implementing all the separable functionalities in a single class makes code less readable and difficult to understand.
iv. Another problem that would arise is that, the Interface Segregation principle would be violated if all the three classes were to be clubbed into one single class.
v. Along with that the Open/Closed principle would also be violated as any modification in a single method of the three tasks would result in modification in the entire classes implementation.

2. <u>Interface Segregation Principle</u>

The second most important principle applied in the Group 13 – Blood Book project is the Interface Segregation principle.

a. <u>Description of the functionality where Interface Segregation Principle is used:</u>

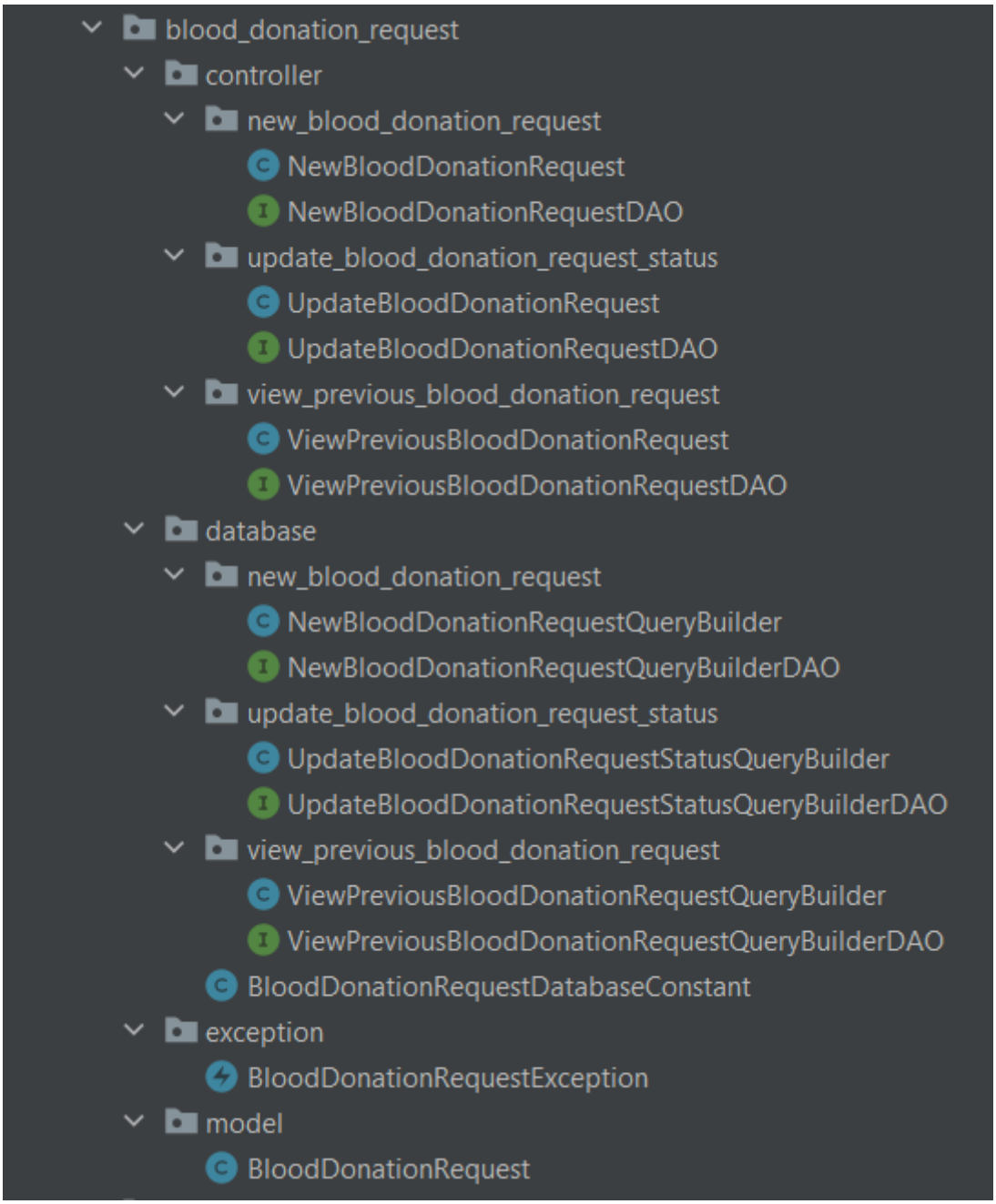
i. **Blood Donation Request by User**

The Blood Donation Request feature is a feature for the end user of the Blood Book application. In this feature there are three things that the end user is able to do

- Create a new blood donation request
- View all previous blood donation requests of that user
- Request for blood donation fulfilment

The above tasks in the blood donation request feature gives the user an end to end option to create a new blood donation request to viewing all the previous requests and also requesting for fulfilment.

ii. The below image shows how separate interfaces are created of each class file implementing a single functionality.



iii. Each interface ends with a **DAO**, focusing specifically on having abstract methods for each functionality class that implements that interface.

iv. The controller classes and the database classes all have their separate interfaces implementing a single functionality which avoids a class from implementing any unwanted methods.

b. <u>Description of why Interface Segregation Principle was applied to this feature:</u>

i. The main reason for implementing Interface Segregation principle for the **Blood Donation Request** feature is that, it being a single feature contains three different tasks, so by implementing the Single Responsibility principle all the classes are separated implementing methods and functionalities specific to that class only. So, here Interface Segregation principle implementation makes interfaces contain only those methods that a class should implement without any unnecessary methods not related to the class's functionalities. Implementing Interface Segregation also makes the code more clean, readable, maintainable, easy to modification and easy to understand. If, this principle wouldn't had been implemented then all the classes would have to implement those unwanted and unrelated methods even though they don't need them or use them.

c. <u>Description of what problems would be triggered in the project if Interface Segregation Principle wouldn't have been applied:</u>

i. Avoiding to implement this principle would result in a lot of coupling in the code and also increases unnecessary dependency which in turn causes the code to become difficult to change.
ii. Another problem would be that unwanted cycles and loops of dependencies gets created.
iii. Major problem caused is that making modifications in the code would be very difficult if this principle is avoided and would also make extending the code troublesome.
iv. The code readability and maintainability would be affected greatly if we avoid or violate this principle.
v. Another problem being triggered would be violation of **Single Responsibility Principle** as if interfaces are not segregated then the classes would have to implement unwanted methods.
vi. Another violation caused is of Liskov Substitution Principle if we avoid using this principle.

3. <u>Dependency Inversion Principle</u>

The third principle applied in the Group 13 – Blood Book project is the Interface Segregation principle.

a. <u>Description of the functionality where Dependency Inversion Principle is used:</u>

i. **Blood Donation Request by User**

The Blood Donation Request feature is a feature for the end user of the Blood Book application. In this feature there are three things that the end user is able to do

- Create a new blood donation request

- View all previous blood donation requests of that user
- Request for blood donation fulfilment

The above tasks in the blood donation request feature gives the user an end to end option to create a new blood donation request to viewing all the previous requests and also requesting for fulfilment.

ii. The below images of code snippets show how **Dependency Inversion** principle is followed in the **Blood Donation Request** feature.

```
private final DatabaseConnectionDAO databaseConnectionDAO;
private final NewBloodDonationRequestQueryBuilderDAO newBloodDonationRequestQueryBuilderDAO;
private final ViewPreviousBloodDonationRequestQueryBuilderDAO viewPreviousBloodDonationRequestQueryBuilderDAO;

public NewBloodDonationRequest(
    DatabaseConnectionDAO databaseConnectionDAO,
    NewBloodDonationRequestQueryBuilderDAO newBloodDonationRequestQueryBuilderDAO,
    ViewPreviousBloodDonationRequestQueryBuilderDAO viewPreviousBloodDonationRequestQueryBuilderDAO) {
  this.databaseConnectionDAO = databaseConnectionDAO;
  this.newBloodDonationRequestQueryBuilderDAO = newBloodDonationRequestQueryBuilderDAO;
  this.viewPreviousBloodDonationRequestQueryBuilderDAO = viewPreviousBloodDonationRequestQueryBuilderDAO;
}
```

```
private final DatabaseConnectionDAO databaseConnectionDAO;
private final UpdateBloodDonationRequestStatusQueryBuilderDAO updateBloodDonationRequestStatusQueryBuilderDAO;

public UpdateBloodDonationRequest(
    DatabaseConnectionDAO databaseConnectionDAO,
    UpdateBloodDonationRequestStatusQueryBuilderDAO updateBloodDonationRequestStatusQueryBuilderDAO) {
  this.databaseConnectionDAO = databaseConnectionDAO;
  this.updateBloodDonationRequestStatusQueryBuilderDAO = updateBloodDonationRequestStatusQueryBuilderDAO;
}
```

```
private final DatabaseConnectionDAO databaseConnectionDAO;
private final ViewPreviousBloodDonationRequestQueryBuilderDAO viewPreviousBloodDonationRequestQueryBuilderDAO;

public ViewPreviousBloodDonationRequest(
    DatabaseConnectionDAO databaseConnectionDAO,
    ViewPreviousBloodDonationRequestQueryBuilderDAO viewPreviousBloodDonationRequestQueryBuilderDAO) {
  this.databaseConnectionDAO = databaseConnectionDAO;
  this.viewPreviousBloodDonationRequestQueryBuilderDAO = viewPreviousBloodDonationRequestQueryBuilderDAO;
}
```

iii. As we can see in the above figures, the **DatabaseConnection** class is not directly accessed by the **NewBloodDonationRequest**, **UpdateBloodDonationRequest** and **ViewPreviousBloodDonationRequest** classes.

iv. An object of the DatabaseConnection interface and respective query builder interfaces are created to access their methods.

b. Description of why Dependency Inversion Principle was applied to this feature:

i. The justification for implementing **Dependency Inversion** principle for the **Blood Donation** feature are as follows:

1. Modules should depend on abstractions rather than on concrete class implementations.
2. And in turn abstractions should not depend on concrete objects.
3. By implementing this principle, I was able to achieve flexibility in my code giving me more room for modifications.
4. Coupling is reduced drastically thus making the code more readable and easier to understand.
5. But the most important reason is adaptable to change. The code won't need to be completely changed to incorporate any changes/modifications.

c. Description of what problems would be triggered in the project if Dependency Inversion Principle wouldn't have been applied:

i. Avoiding to implement this principle will lead to increased coupling and less cohesion in the code which is a sign of bad code.
ii. Another problem that would arise is that making any modifications will be very difficult as the code is not at all flexible.
iii. The code maintainability, readability, understandability is severely affected.
iv. Avoiding this principle will lead to all modules interacting through concrete classes and their objects.
v. Avoiding this principle violates Interface Segregation principle to some extent.
vi. We won't be able to achieve higher level of orchestration code, giving us the control to pass which object to satisfy the dependency.

4. Open/Closed Principle

The fourth principle applied in the Group 13 – Blood Book project is the Open/Closed principle.

a. Description of the functionality where Open/Closed Principle is used:

i. **Blood Donation Request by User**

The Blood Donation Request feature is a feature for the end user of the Blood Book application. In this feature there are three things that the end user is able to do

- Create a new blood donation request
- View all previous blood donation requests of that user
- Request for blood donation fulfilment

The above tasks in the blood donation request feature gives the user an end to end option to create a new blood donation request to viewing all the previous requests and also requesting for fulfilment.

ii. The below images of code snippets show how **Open/Closed** principle is followed in the **Blood Donation Request** feature.

```
private final DatabaseConnectionDAO databaseConnectionDAO;
private final ViewPreviousBloodDonationRequestQueryBuilderDAO viewPreviousBloodDonationRequestQueryBuilderDAO;
```

```
private static final String STATUS_REQUEST = "request";
private static final String STATUS_ACTIVE = "active";
private static final String STATUS_FULFILLED = "fulfilled";

private final DatabaseConnectionDAO databaseConnectionDAO;
private final NewBloodDonationRequestQueryBuilderDAO newBloodDonationRequestQueryBuilderDAO;
private final ViewPreviousBloodDonationRequestQueryBuilderDAO viewPreviousBloodDonationRequestQueryBuilderDAO;
```

iii. As we can see in the above figures, all the member variables are made private so they cannot be accessed nor modified by some other class that extends the above classes.

b. <u>Description of why Open/Closed Principle was applied to this feature:</u>

i. The justification for implementing **Open/Closed** principle for the **Blood Donation** feature are as follows:
   1. Reduces coupling, making the code more manageable and robust.
   2. Adding any new code or making any changes becomes quite easy as there is less coupling and more cohesion in the code.
   3. The idea of making the member variables private shields the variables from being accessed from outside the class.
   4. As, if the member variables of the class change all the methods that use those variables are affected causing the application to fail.

c. <u>Description of what problems would be triggered in the project if Open/Closed Principle wouldn't have been applied:</u>

i. If this principle is avoided then the code becomes fragile, rigid, unpredictable, un-reusable and less flexible.
ii. Compromising the internal secrets of the class by exposing the member variables can cause some serious issue to the code.
iii. Avoiding this principle leads to code design that will always tend to change thus affecting the overall working of the application.
iv. The main purpose is defeated of not changing the behaviour of old code by adding the new code.

<u>NOTE:</u>

These were the SOLID principles that are implemented by the **Blood Donation Request** module.

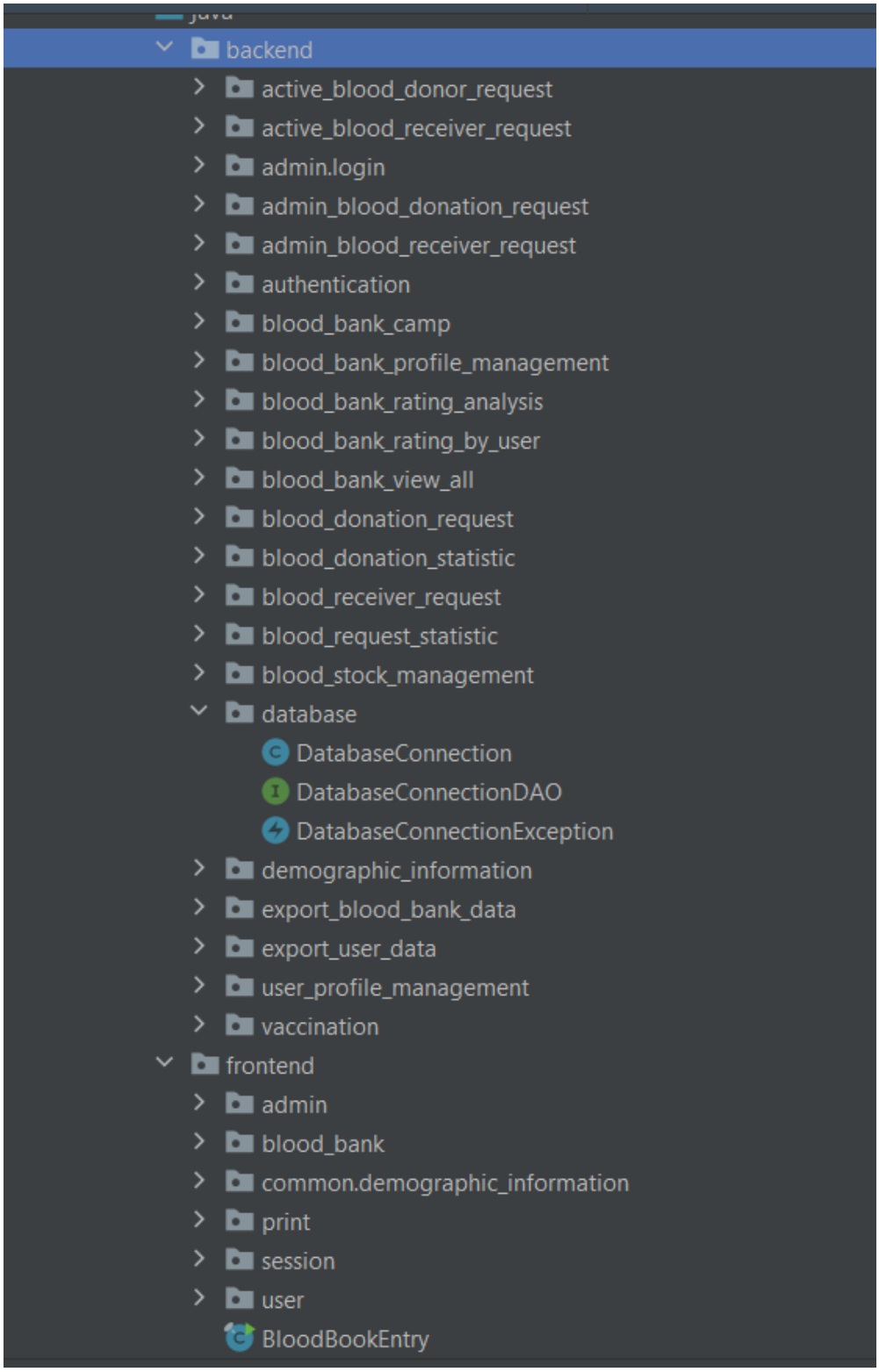
Similarly, all other modules that I have developed along with the above one follow the same SOLID principles:

- All Blood Donor Requests

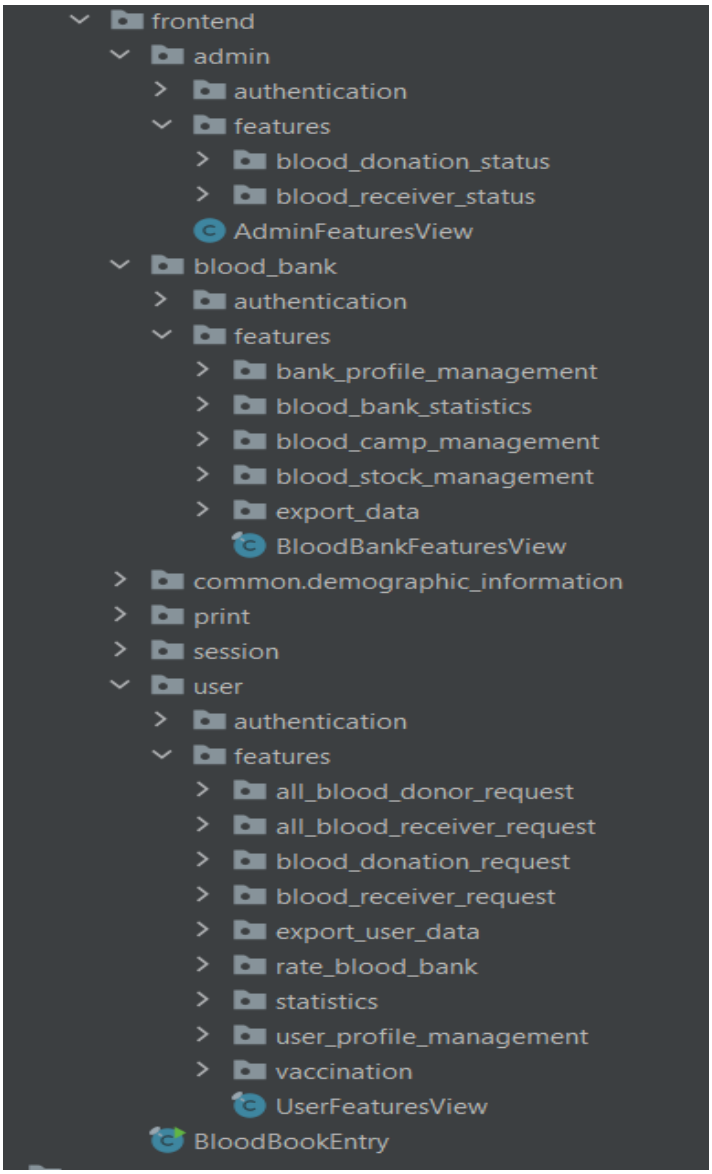
- Blood Bank Profile Management
- User Statistics (relating to blood donation)
- Export User Data
- Export Blood Bank Data

Task 2:

The Group 13 – Blood Book project follows a layered approach of the MVC architecture that is the Model View and Controller. The below image shows how different packages are made for the backend, frontend and database. This segregation makes the application more robust and easily reusable. Each part of the code has a specific purpose and those purposes are different making the architecture clean and layered.

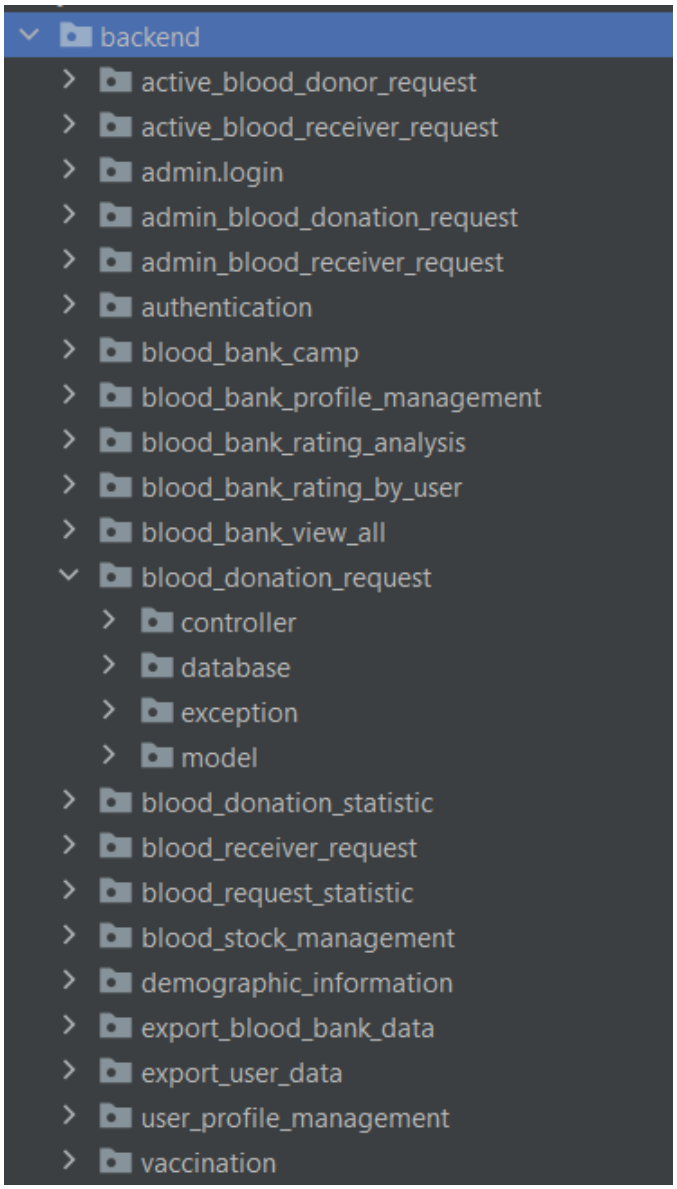


1. **Description of the Presentation Layer**
    a. The presentation layer of the Blood Book project is a clean and structured package hierarchy that makes it very easy to understand, use and modify as well.
    b. The idea is that, inside the frontend package, there are three main packages **admin**, **blood_bank** and **user** focusing on the three end users that will use the application.
    c. Other packages like the session, and print are the helper packages containing specific and generic classes that would be used by all other front-end classes.
    d. Separating the presentation layer from other layers leads to isolating what user interacts with from the business logic and the database as well.
    e. In the frontend not even a single database connection is made, only instances of the database interface are created and passed as parameters.
    f. Except the **print** package not even a single print statement is written anywhere in the frontend.
    g. Each package contains a feature specific view file containing the logic that solely interact with the user and passes parameters to all other methods containing the business logic.
    h. The frontend has a **BloodBookEntry** file which is the start file for the application. The application initiates from that file and ends on that file displaying the response.



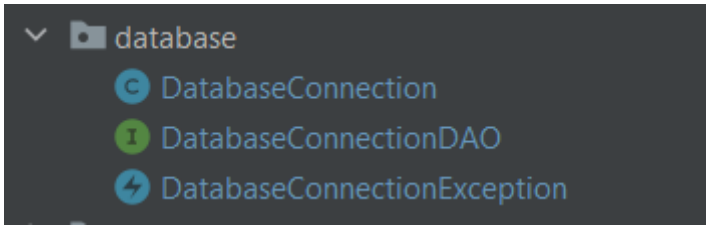
2. Description of the Business Logic Layer

a. The business logic layer of the Blood Book project is similar to the presentation layer of the project – clean, simple, easy to understand and use.
b. Each individual feature of the application implementing business logic is stored in a separate package with that features name.
c. Now, within each feature package the logic is further divided into controller, database, exception and model packages.
d. These packages club all the business logic for a particular feature in a single package, with multiple separate packages implementing specific functionalities.
e. The model package contains the java bean class.
f. The database package contains the database queries being used to interact with the database to fetch, create or update data.
g. The controller is the main package that contains the actual business logic that does all the processing and filtering and conditioning.
h. The segregation and interaction among the classes follow the SOLID principles.
i. The exception package is the custom exception for each feature being implemented.



3. Description of the Database Layer

a. The below image shows the database package that actually contains the logic to establish connection to database following the SOLID principles.
b. It also has its own custom exception.
c. This **DatabaseConnectionDAO** is the one that is instantiated everywhere to access the DatabaseConnection class and create a connection to the database.
d. This is the layer that persists the data to the MySQL database.
e. There is a separate **databaseConfig.properties** file that stores the different database credentials and connection details which are accessed by the **DatabaseConnection** class to connect to desired database.



Task 3:

1. Singleton Creational Pattern

a. Description of the functionality where Singleton Creational Pattern is used:

i. This pattern is being used throughout the project wherever it could be used.
ii. In the below figure it can be seen that mainly this pattern is beneficial in the Database connection class, as we only need one instance of the database to be created avoiding the error of Too many connections.
iii. This pattern restricts the class to be instantiated only once, throughout its life.

```
private static final String DATABASE_CONFIGURATION_FILE = "./databaseConfig.properties";
private Connection connection = null;
private static DatabaseConnection databaseConnection;

private DatabaseConnection() {
    //Required private constructor
}

public static DatabaseConnection getInstance() {
    if (databaseConnection == null) {
        databaseConnection = new DatabaseConnection();
    }
    return databaseConnection;
}
```

b. Description of why Singleton Creational Pattern was applied to this feature:

i. The main reason for applying this principle to the database connection class is because we only need to connect to the database once.
ii. Once connected to the database, same instance can be used to perform all the operations throughout the programs application.
iii. Creating multiple connections to the database is not feasible and a good practice.

c. A description of what problems are avoided in your code by applying such pattern:

i. If we wouldn't apply this pattern, then multiple connections would be made to the database leading to causing an error of Too many connections, hampering the applications operations,
ii. This also wastes a lot of resources and increases the processing time of the application, sometimes leading to connection timeouts as well.

2. Abstract Factory Creational Pattern

a. Description of the functionality where Abstract Factory Creational Pattern is used:

i. This pattern has been used throughout the project as each class following the SOLID principles has an interface for it.

ii. This makes the class unreachable from an external method implementation, making the code more secure and hiding the important details.
iii. Using this pattern also makes the code more readable, manageable, easily modifiable and more secure.

b. Description of why Abstract Factory Creational Pattern was applied to this feature:

i. There are several reasons for applying this pattern like, this pattern isolates the concrete classes and even their names are not shown in the client code.
ii. The process of exchanging product families becomes very easy by applying this design pattern.
iii. Another major reason for applying this pattern promotes consistency among products.

c. A description of what problems are avoided in your code by applying such pattern:

i. The problems that could have been caused by not using this pattern would be that the SOLID principles would be violated.
ii. There would be high coupling and low cohesion causing the program to become difficult to handle and maintain.

3. Decorator Structural Pattern:

a. Description of the functionality where Decorator Structural Pattern is used:

i. This pattern can be sued in the application. As of now it is not being implemented.
ii. This pattern focuses on the structural pattern, as it provides a way to add additional responsibilities to an object dynamically.
iii. They provide a flexible and easy alternative for extending functionality by sub classing.

b. Description of why Decorator Structural Pattern can be applied to this feature:

i. Using this pattern adds more flexibility to the code as compared to the static inheritance.
ii. The most interesting feature is that they can be added and removed at runtime.

c. A description of what problems are avoided in your code by applying such pattern:

i. The problems that could arise by not applying this pattern is affecting the flexibility of the code, making it more rigid and prone to failures on adding new features.
ii. The decorator and its components are not the same, so if we implement the pattern in a wrong manner then it could cause the code to fail.

4. Mediator Behavioural Pattern:

a. Description of the functionality where Mediator Behavioural Pattern is used:

i. This pattern implements the set of objects that communicate in a previously determined form and state.

ii. This pattern is implemented using he model package containing the model classes containing instantiation of the class and having getter setter methods as mediators.

b. Description of why Mediator Behavioural Pattern was applied to this feature:

i. This pattern mainly abstracts how the objects cooperate with each other.
ii. Another benefit is that it centralizes the control of the objects and the application, moving complexity of the interaction into mediator.

c. A description of what problems are avoided in your code by applying such pattern:

i. The problems that could arise by avoiding this pattern is high coupling and low cohesion.
ii. This makes the code less modifiable and maintainable.

5. Chain of Responsibility Behavioural Patter:

a. Description of the functionality where Chain of Responsibility Behavioural Pattern is used:

i. This pattern is implemented in the project, as many features and functionalities are dependent on each other although by loose coupling.
ii. The main part is that the client has no idea which object handles the request, as client is concerned with whether the request has been handled or not.

b. Description of why Chain of Responsibility Behavioural Pattern was applied to this feature:

i. This pattern makes the flow of the application more robust, like there is frontend, backend and the database modules being chained to one another.
ii. Changing the behaviour of the application can be achieve by adding or removing objects at runtime from the chain.
iii. The only drawback is that there are chances that the request may not be handled at times.

c. A description of what problems are avoided in your code by applying such pattern:

i. The problem of unhandled request can be resolved by using this pattern, mainly for the MVC project structure.

6. Template Method Behavioural Pattern:

a. Description of the functionality where Template Method Behavioural Pattern is used:

i. The project can use this pattern where a template methods created which can be reused.
ii. They factor out the common things in the library being used, leading to better code maintainability and extensibility.
iii. Interesting thing is that this is known as dependency injection.

b. Description of why Template Method Behavioural Pattern was applied to this feature:

i. The reason for using this pattern in the application is that, it leads to an inverted control structure called the dependency inversion.

ii. Majority of high-level objects will pick and choose from a set of subclasses that are accessible.

c. A description of what problems are avoided in your code by applying such pattern:

i. If this pattern is not used it could cause the application to crash if the implementors of the subclass forget to call the super class.

ii. The major effect would be that it could break the implementation leading to errors and bugs in the code.