University Of Technology Jamaica

The Faculty of Engineering & Computing (FENC)

School of Computing & Information Technology

Academic Year: 2020 - 2021

Semester: 1

Module: Advanced Programming (CIT3009)

Group Members:

Gregory Woolery – ID#1803376

Matthew Ruddock – ID# 1700241

Advanced Programming Group Assignment

Date of Submission: December 4, 2020.

Lecturer: Mr. Christopher Panther

Tutorial Day/Time: Wednesday 12:00pm To 1:00pm

**Table of Contents**

[Group Report](#_gjdgxs) [3](#_gjdgxs)

[Learning Trail](#_30j0zll) [4](#_30j0zll)

[Object-Oriented Analysis](#_1fob9te) [5](#_1fob9te)

[Object-Oriented Design](#_3znysh7) [7](#_3znysh7)

[Represent relationships among classes using a UML relationship diagram](#_tyjcwt) [20](#_tyjcwt)

[Design of System](#_1t3h5sf) 21

# **Group Report**

|  |  |
| --- | --- |
| **Name** | **Contributions** |
| Gregory Woolery | Advanced Graphical User Interface,  Authentication,  Database Design and Domain,  Employee Services,  Event Logging,  Object-Relational Mapping,  TCP/ IP Socket- based Networking and Object Serialization,  Threading,  Traditional Database Connectivity,  Student Services,  Written Report |
| Matthew Ruddock | Advanced Graphical User Interface,  Authentication,  Database Design and Domain,  Employee Services,  Event Logging,  Traditional Database Connectivity,  Student Services,  Written Report |

# **Learning Trail**

**GUI:**

The Official documentation for swing UI was extremely helpful when finding how to use different components as well as using the ActionListener for them. Hours upon hours were spend reading and analysis how to use the components to suit our needs. Referring:

Extending components to classes to create our own components was upon the first problem we experience when trying to use the different methods associated with the parent class.

This was needed in the Login Screen where all these were learnt:

**Use different layouts in swing:**

Including using the ‘GridBagLayout’ with its extensive learning curve that took between 4-5 days to understand and be able to implement freely. The constraints in laying out different components and resizing was an extreme bother but there were materials online that assisted in the process.

* Using insets to move components in the fields created by the column and row selections.
* Adding the components and its layout to the GridBagLayout.

Resizing the rows and columns while leaving a space to add another component made development difficult in the first couple of days. After many hours of reading the documentation and separate online resources this was hurdled after sometime.

Among the grid bag layout was using the default FlowLayout that didn’t take much time to understand, The GridLayout where each component in a grid is the similar size.

<https://docs.oracle.com/javase/tutorial/uiswing/layout/grid.html>

<https://www.geeksforgeeks.org/java-awt-gridlayout-class/?ref=lbp>

In points:

* Hiding then showing hidden messages when a condition is met using Component.isVisible(boolean)
* Test if JTextArea or JTextField is empty with the inhouse methods such as .Trim(), .isBlank(), .isEmpty(), .length()
* Finding which Radio button is selected out of a group

Resize images using: new ImageIcon.getImage().getScaledInstance(int width, int height, int hints).

* The link above provided a range of opportunities with images with its customization that could be used on the login screen and other views.

Implement external libraries found from, for example on GitHub – LGoodDatePicker.

* We initially set out with the goal to provide users with the opportunity to select a date to search from where the LGoodDatePicker would be used since Java Swing UI doesn’t provide such a calendar selection component or method. This proved to be very useful until development was hindered because of problems with extracting the date and parsing it into java.sql.Date form. As such it was removed because the deadline was close.

GitHub Repo

* <https://github.com/LGoodDatePicker/LGoodDatePicker>

Download releases

* <https://github.com/LGoodDatePicker/LGoodDatePicker/releases>

Helpful sites

* <https://docs.leponceau.org/java-examples/java-evaluation/gui.lgooddatepicker.LGoodDatePickerTest.html>
* <https://www.programcreek.com/java-api-examples/?api=com.github.lgooddatepicker.components.DatePickerSettings>
* What .setMargin does for JTextField vs other components
* How to format components using:
  + Borders
  + Backgrounds
  + Text – Font, foreground colour, size
  + How to centre frames on the screen using Toolkit and JFrame.setLocation
  + How to add icons and title to frames
* Learnt the difference between
  + Foreground vs Background
  + Preferred Size vs Maximum Size vs Size vs Minimum Size
* Label alignments with components
* How to use the java.awt.Dimension class

**Action Lister:**

* Writing ActionListener’s initially was a difficult task. Nonetheless the documentation proved to be the saviour for most of our problems. Figuring out how to close database connection in the event that the frame was forcefully closed was solved using the official documentation.
* For extending ActionListener, implementing and overriding the required method
  + public void actionPerformed(ActionEvent e) {}
* To find out which component fired and event was a found using code from Stack Overflow where the person implemented “if(e.getSource() == login\_btn)”
* where the .getSource method was compared with a component in find the component that fired the event.

**Using Action Listeners**

* Mouse Listeners using the inner class method was a great experience learning how Low-Level Events and Semantic events are handled and how performance is affected using different implementations.
* Also completing actions when the number of clicks that was predefined was also studied as well as when hovering over a button, changing the background.
* Disposing frames
* Proper communication methods with team
* Integrating GitHub with projects
* Assigning roles to different members
* How to efficiently search the web for solutions

**Creating Frames**

Creating frames was an expensive task and lowering the amount of code that was written we chose to create a skeleton Dashboard that inherits from JFrame that both the student and staff dashboards would inherit so use the variables created.

When creating the different JFrame there was a need to have the current logged in user information being displayed for a more incitive feel. We ran into many problems in creating a method that could store the current user in the system. We first created a static variable from the USER class that would store the current user but this seemed an expensive process in itself calling from every class that needed it. Logout was also a problem that the static class was unable to handle.

All this happened before the server was created so we had no knowledge that you could save the current user in the server thread and call from there. We discussed the static variable way for some time and came up with a solution to call from a class that would handle every user function: USERCONTROLLER.java. This would handle logins, getting all of the user credentials and logout at the time.

There was an issue with labels being displayed that was larger than the actual space giving. This is when we found out the HTML tags could be used with Swing UI components for example:

* JLabel name = new JLabel(“<html> hello “ + firstname + “<br>” + lastname + “</html”)
* Would allow lastname to go on a different line than firstname.
* Learnt how to use mnemonics and tooltiptext with JMenuBar, JMenu, JMenuItems.
* Learnt that without setVisible(true) components would not show

**Using JInternalFrame**

Initially using JInternalFrame was one of the hardest pieces we had to implement. The fact that JInternalFrame had the top bar was unlikely by Mr. Woolery and a method was searched for to remove this. At first the JInternalFrame was not to be minimized, maximized or moved from its position. This proved to be a difficult concept without having to do a serious reconstruction of the Frame itself. This had put us at a stop and forced to read hours of code online as well as documentations of Frame structures. After much consideration and reading about its container, JDesktopPane. The method to solve all the problems was a simple as removing the top bar from the JInternalFrame. Online resources quicky gave us an answer to remove north pane on the JInternalFrame using: (javax.swing.plaf.basic.BasicInternalFrameUI)this.getUI()).setNorthPane(null);

Considering this change the frame could not move, but it was in the wrong position and we were again at a stop. A solution was already online that we implemented to suit our needs Using Dimension class to get the JDesktopPane and the JInternalFrame sizes and setLocation from the JInternalFrame with set the with and height where the JInternalFrame should stay.

It was also a difficult process when switching between JInternalFrames. A couple of days were spent on this to find a solution to either use dispose or the JDesktopPane.removeAll() with is updateUI method. The dispose proved to be useful for some cases but the latter proved efficient is all. To keep a good vibe going we decided on using both, then a new JInternalFrame on the JDesktopPane to have a smooth transition.

* Learnt that the getContentPane() of the JFrame is the standard way to begin initializing components.
* Learnt how to use JComboBox with ArrayList and String Arrays.

**Using JTables**

* Learnt how to initialize tables with set width for rows, not making them editable or resizable and customizing changing header text
* How to make selection only be to a single row and not no column selections.
* How to update the table using a model (DefaultTableModel) that adds rows and refreshes as they are added.
* How to add ActionListener to tables to allow for actions when a double or single click is fired. This is tested against a major problem we had with the ActionListener firing an event when the table headers were clicked. This was solved by placing an if statement with the following condition that ensures the header isn’t selected, table.getSelectedRow() != -1
* How to get values from the JTable was a problem because of the first-time using Table Models.
* We quickly scanned through the list of methods offered by the class to find getModel().getValueAt(row, column) that solved our problem in retrieving the data from a specific row in a specific column.
* Learnt that a JTable couldn’t be used without a JScrollPane

**ADD – ISSUE - GUI**

A problem arose when students were entering in the details of the issue, they had but went over the specified number of characters that the database had to store it. There was not method in the JTextArea or JTextField that implemented this so we outsourced another method.

A method was found to limit the amount of characters by using a Document Filter. There was a solution online that when the max characters are entered as a parameter then the field does not go over that limit. To illustrate this, we implemented a method that could down from 140 characters when the user is typing and stops when the counter reaches 0.

This implementation can be found in the com.services package, in the class DOCUMENTSIZEFILTER.java.

**Setting Up Database**

* **Native-Sql**
  + Learnt about port numbers and which ones are free to use by developers.
  + How to change password of mysql database
  + How to receive a connection from the mysql database or initialize a connection to create tables and add to them.
  + Table creation was a problem because of placing commas and quotations at the proper locations. This was solved after a day of starting from scratch again and reading about Native SQL Database table creation.
  + How to use foreign keys, primary keys, default values with JDBC Native SQL
  + How to use prepared statements and result sets to find values after passing in Native SQL
  + Problems with passing java.util.date arose since database only accepts java.sql.date. Online resources were found to find a way to convert one to the other and use concurrently with little to no issues. Outputting and saving the data in a required format was also a problem and had to parse using SimpleDateFortmatter to show consistent results.
* **Hibernate**
  + Hibernate took days to install into the project because of its steep learning curve. Adding annotations to the classes were researched before finding a solution that would only help for a few cases. Later a solution where annotations were inherited with the ‘javax.persistense’ library was found that worked for all cases and could be implemented in searching the MYSQL Database.
  + In the ‘SessionFactoryBuilder’ all the class the annotations were added to the hibernate.cfg.xml configuration one by one because there hasn’t been a way to include a whole package of classes.
  + Since most of the code needed was online for selecting columns based on a criterion from a database as well as update and delete it was a matter of implementing them to suit our needs.
  + However, implementing them in HQL was a difficult task because of how different it is from Native SQL. We had to consult the official documentation for assistance as well as other online resources that gave us only pieces of the solutions. Nonetheless, we ended us figuring ways to use the ‘Query.setParameter’ function that allows variables to be tested in HQL.
* **Server Side**
  + A major component of this project proved to be the hardest yet most versatile side. At first, we had a simple client server interface that created connections each time it requested data from the database. As soon as the server retrieved the data and sent it back the connection was closed causing expensive threading. It was seen through the console where threads were created by the dozens just to complete some simple operations. Action was taken to read online resources and watch videos based on the topic. A lot of results were shown that only overwhelmed.
  + There was a connection through all these resources that stood out. In order to create a multithreaded server, we had to have a main thread that delegated action to towards a handler thread that receives and controls its own connection from a user. No other thread could access another’s thread unless they were seeing the user which is connected on that thread. An ArrayList was created to manage the all the connected users with the USER model implemented to store the user connected. Now spawning thread connection operation was reduced to only the number of connected users plus the main thread delegator.
* **Client-Side**
  + With the change in the server implementation how the client side sent a request was also a difficult task to implement. A client class was implemented that handled sending and receiving request and creating connections.
  + Requests that were first send one by one were now wrapped in an Object ArrayList and send over to the server that extracted the operation and operand that were sent. This provided a smooth means come communication between client and server while drastically reducing lines of code.
  + Connection creation was also taken out and put inside the Client class so only a single connection would be created per client.

**LOG IN VS LOG OUT**

The logout operation was now even more complicated since the server now stored the user currently on the connection. In the event that a user selects to logout or stop the program the connection, input and output streams and the current user would all have to be reset. This looked a very detailed procedure at first glance where hours were spend reading and watching videos on removing the current user from a client server multi-threaded application.

The solution somehow was as simple as sending a disconnect message to the server to remove the current client from the ArrayList storing connected users, closing the connection and input output streams on the server side. The client side would then close the connection on that side along with it input output streams and log out the current user. Another user was now free to login without an exception being thrown.

**LOGGING**

Since it wouldn’t be efficient to store logs on the client side a method was sought out to send logs over the network to the server side. The methods that were decided between were a database that stored logs or the Socket Appender from log4j2. Since log4j2 was studied in a good detail the Socket Appender was decided upon. The appenders were first set up to log on each side initially using Console and Rolling File Appenders. After that was set up there was a problem using the Socket Appender since the Console and Rolling File Appenders were set up on the client side. After hours of research both appenders were taken out and allowed the Socket Appender to stay on the client side alone.

Logs were being sent to from client to server but it was not being recognised by the server’s input stream. This demanded further research to come to a conclusion of testing if the received Object is an instance of org.apache.logging.log4j.core.LogEvent. If this was true then delegate and log events based on respective log levels on the server side with client information also being logged.

**LIVE CHAT**

Live chat functionality was not implemented because students couldn’t find a way to send data over the server from one client to the other and keeping the functionality of performing different operations send and receiving requests. The client and server side were rewritten many times with errors rising in GUI freezing because of waiting for a response, not being able to send an ONLINE notification to another Client and not being able to send messages to clients without GUI freezing.

# **Object-Oriented Analysis**

The University of Technology (UTECH), Jamaica, is the premiere University in Jamaica. One of the outstanding features of UTECH is the efficiency with which *students*’ Services are dealt with. However, the COVID-19 pandemic has drastically impacted the harmonious relationship the university’s *administrative staff* has enjoyed with its *students.* There are now gaps between *students*’ Services queries and their resolution. The university must now contend with numerous *complaints* from *students*. To maintain the efficiency in treating with *student* Services issues for which the *university* is known, the *Students Services manager* wants a query logging system that will be able to **capture** all *student* *complaints* and *queries*. Upon receipt of a *complaint* or *query*, a *Student Services Agent* will **log** the *complaint* or *query* and **schedule** a *Student Services Representative* to contact the student to rectify the issue **remotely**.

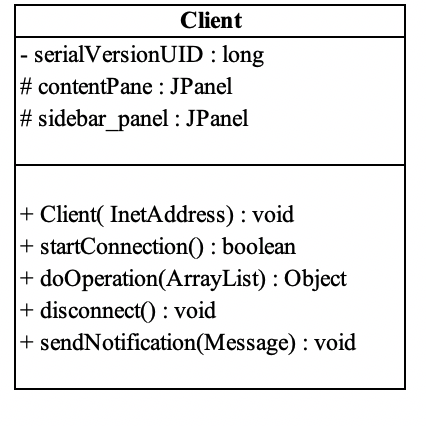
You are to **build** a TCP/IP socket-based Client / Server architecture application that will facilitate the above scenario. Your Client application must **send** requests to the server application as only the server application can establish connections to the database. The client application must consist of a GUI. The server application need not have a GUI. However, it may include graphical components to enhance user experience.

Your system must **capture** all data about *Students*, *Student Services Representatives*, *Student Services Agents*, *Complaints* and *Queries*.

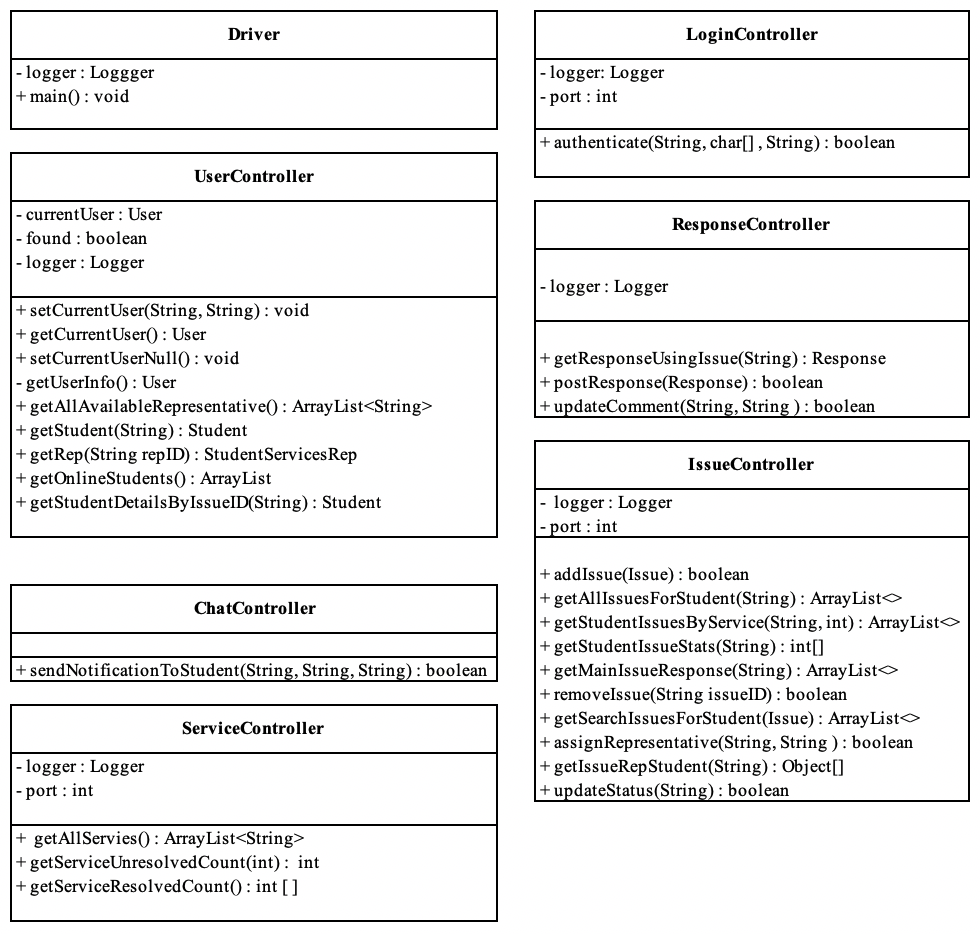
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Potential  Class | Retained Information | | | Needed Services | Multiple Attributes | | Common Attributes | | Common Operations | | Essential Requirements | | | Accepted As Class |
| University of Technology | | N | N | | | N | | N | | N | | N | N | | |
| Jamaica | | N | N | | | N | | N | | N | | N | N | | |
| Students | | Y | Y | | | Y | | Y | | Y | | Y | Y | | |
| Student Services Representatives | | Y | Y | | | Y | | Y | | Y | | Y | Y | | |
| Student Services Agents | | Y | Y | | | Y | | Y | | Y | | Y | Y | | |
| Complaints | | Y | Y | | | Y | | N | | Y | | Y | Y | | |
| administrative staff | | Y | Y | | | Y | | Y | | Y | | N | N | | |
| Students Services manager | | Y | Y | | | Y | | Y | | Y | | N | N | | |
| query logging system | | Y | Y | | | Y | | Y | | Y | | Y | Y | | |
| Query | | Y | Y | | | Y | | Y | | Y | | Y | Y | | |
| Client | | Y | Y | | | Y | | Y | | Y | | Y | Y | | |
| Server | | Y | Y | | | Y | | Y | | Y | | Y | Y | | |
| Database | | Y | Y | | | Y | | Y | | Y | | Y | Y | | |

# **Object-Oriented Design**

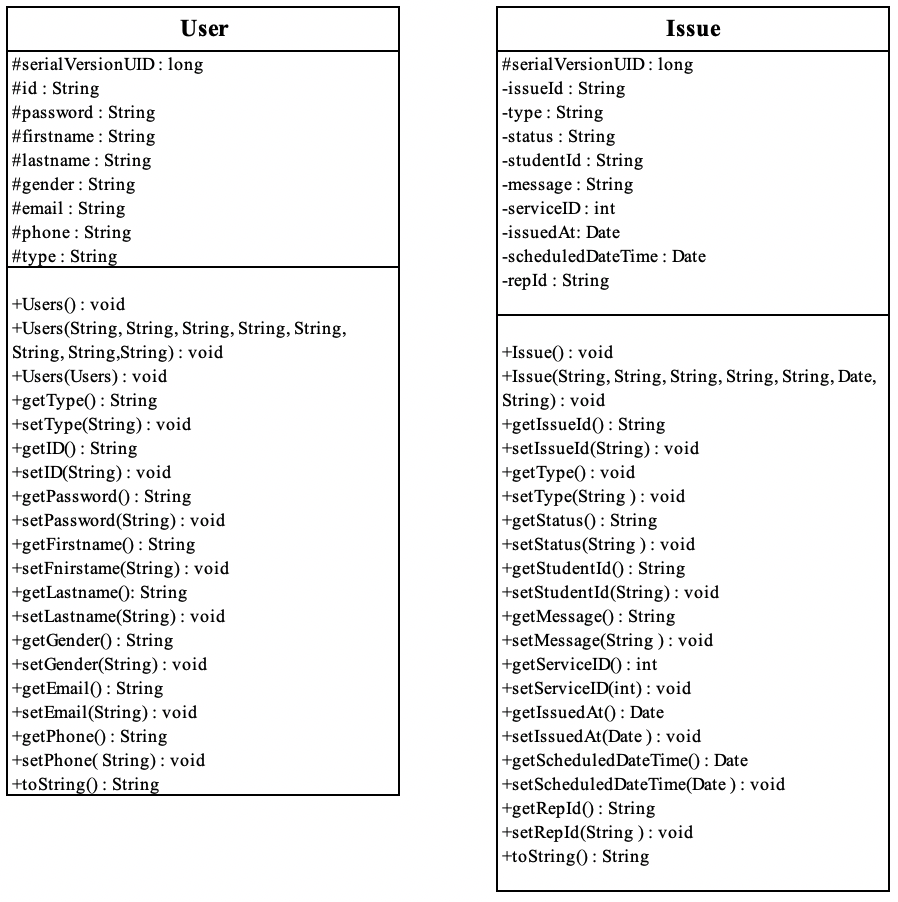
com.Client

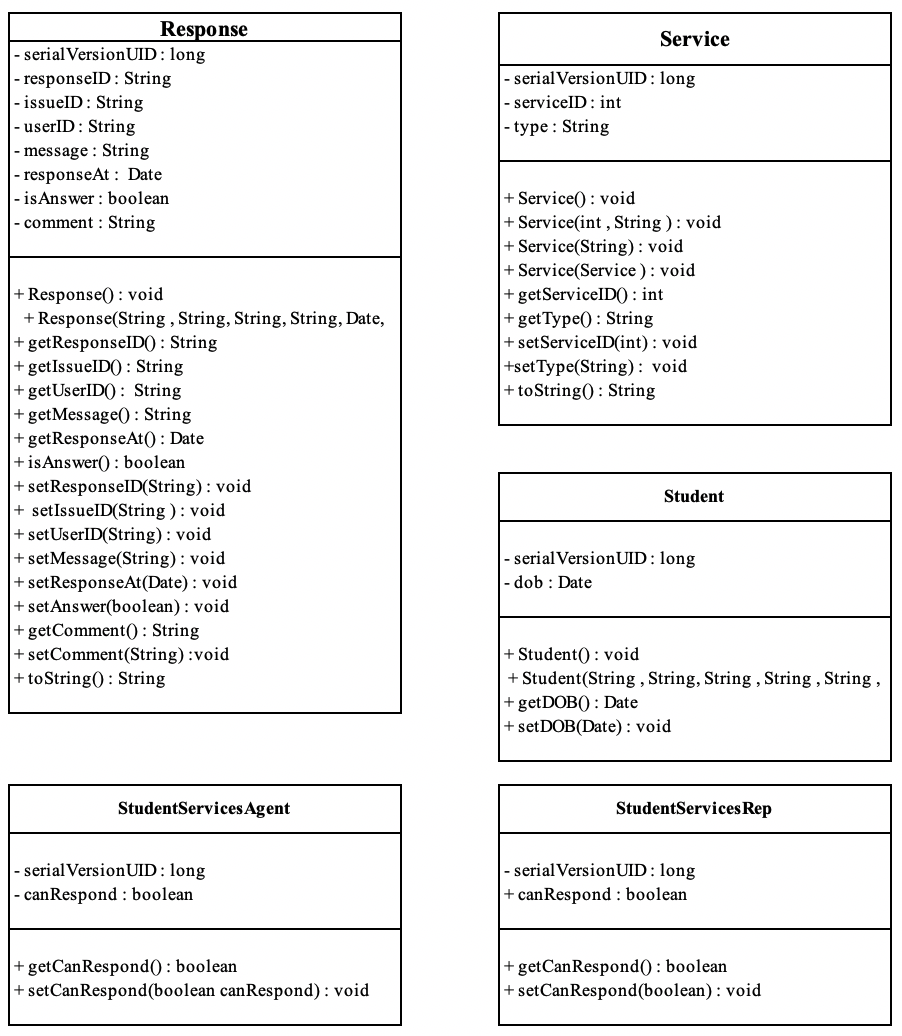


com. Controller

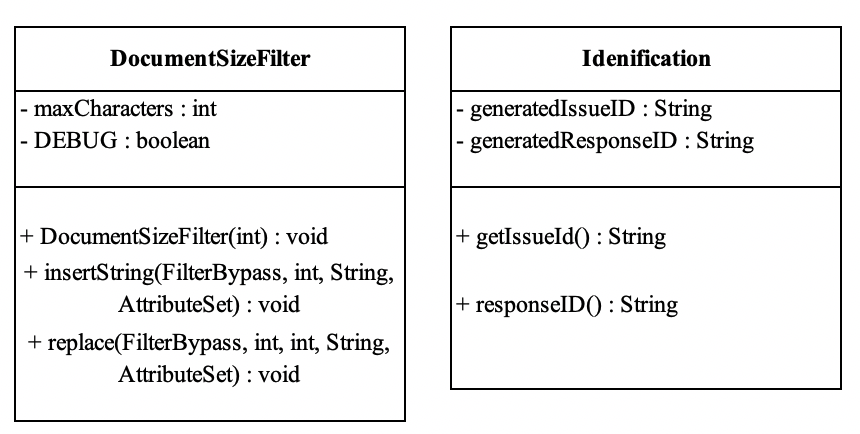


com. model

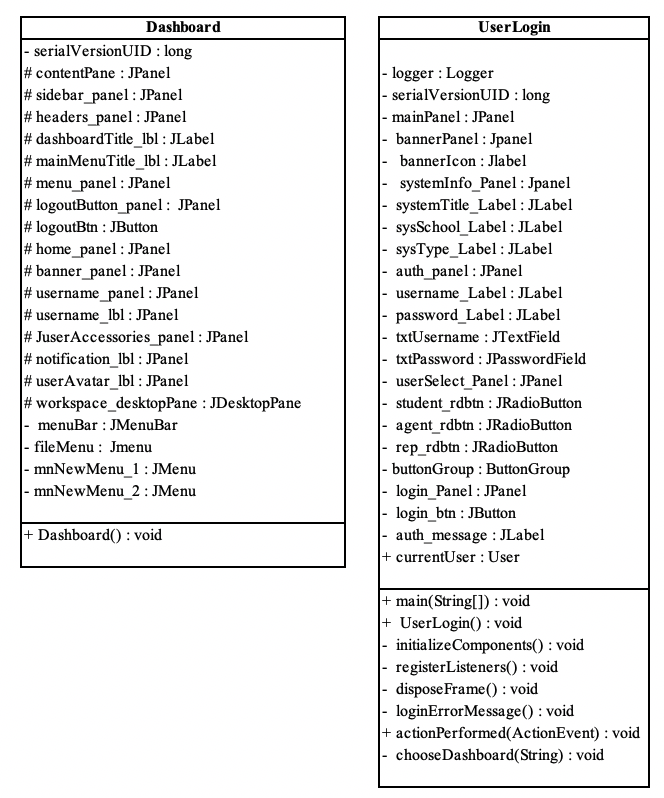




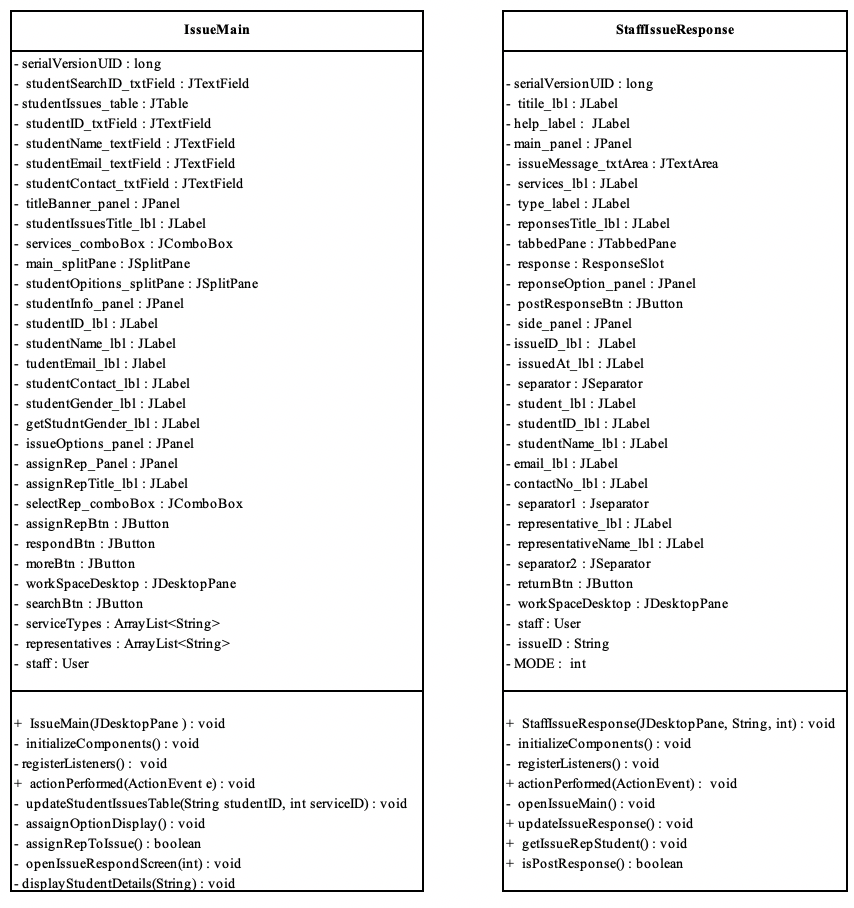
com. services

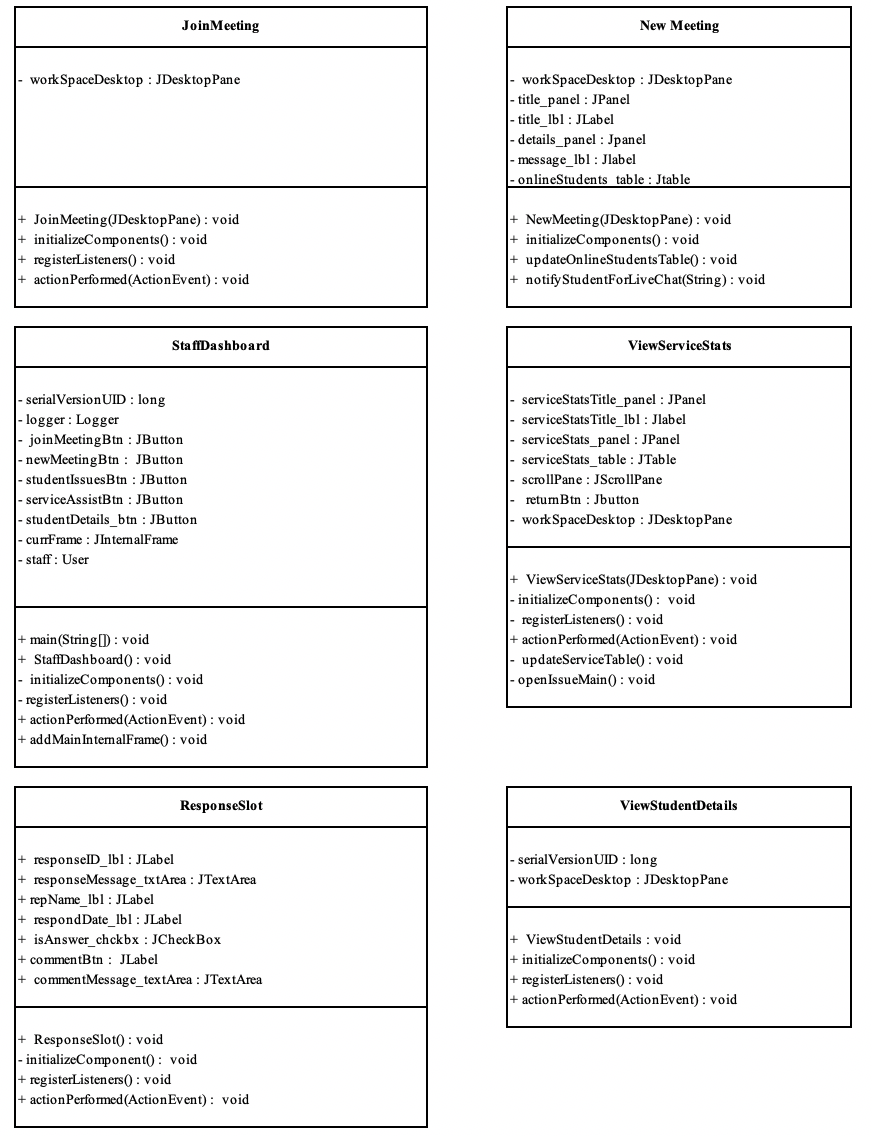


com.view



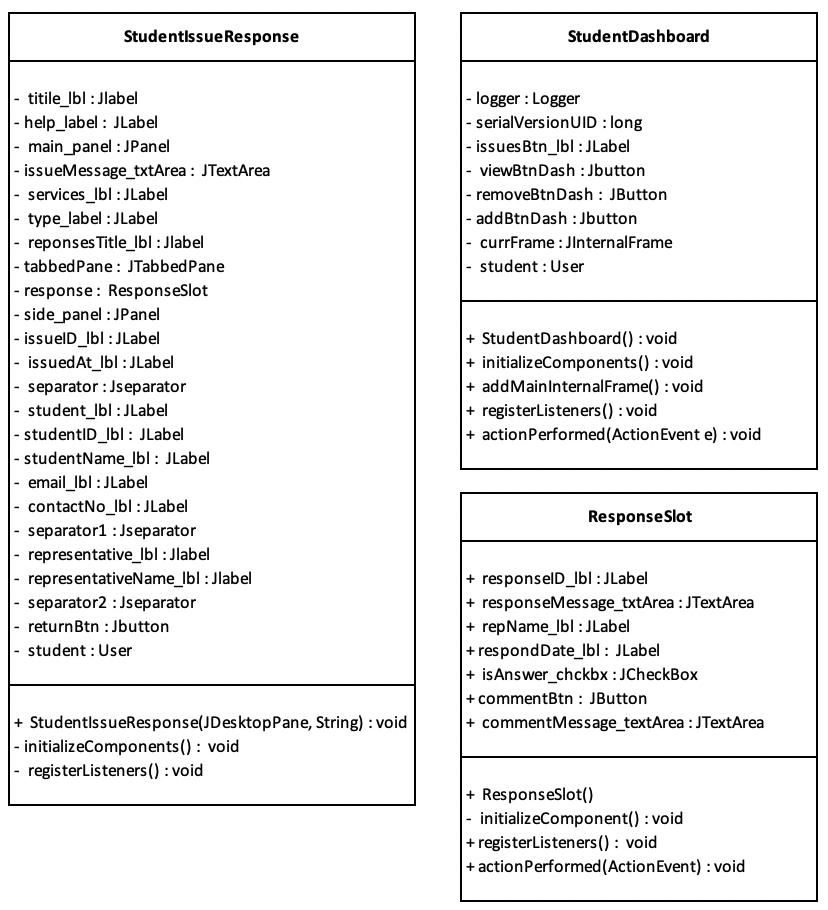
com.view.staff



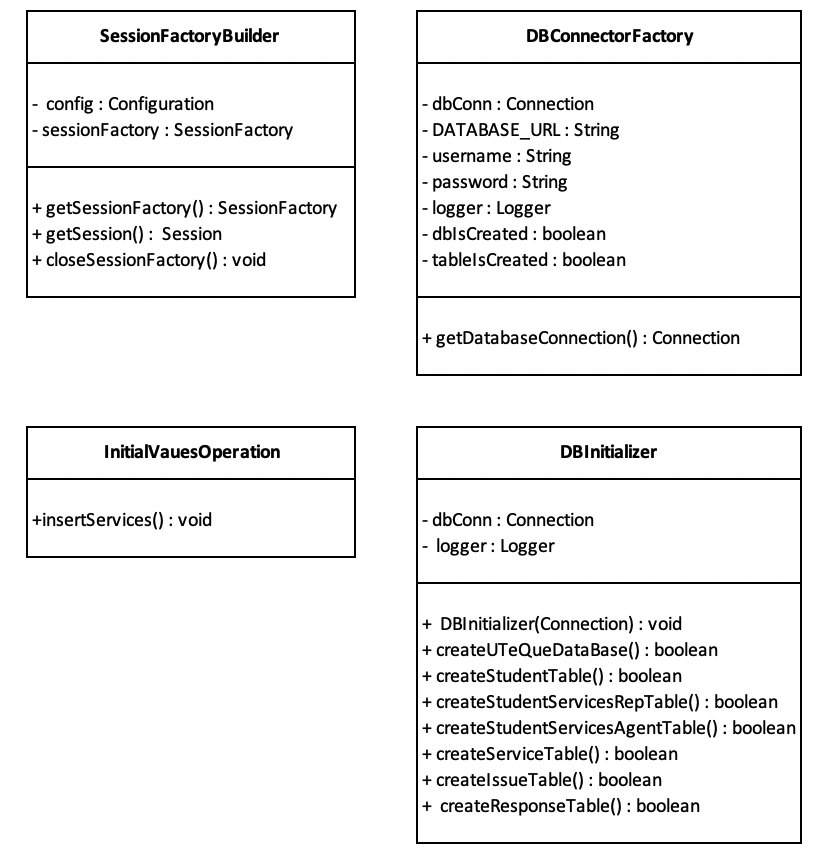


com.view.student

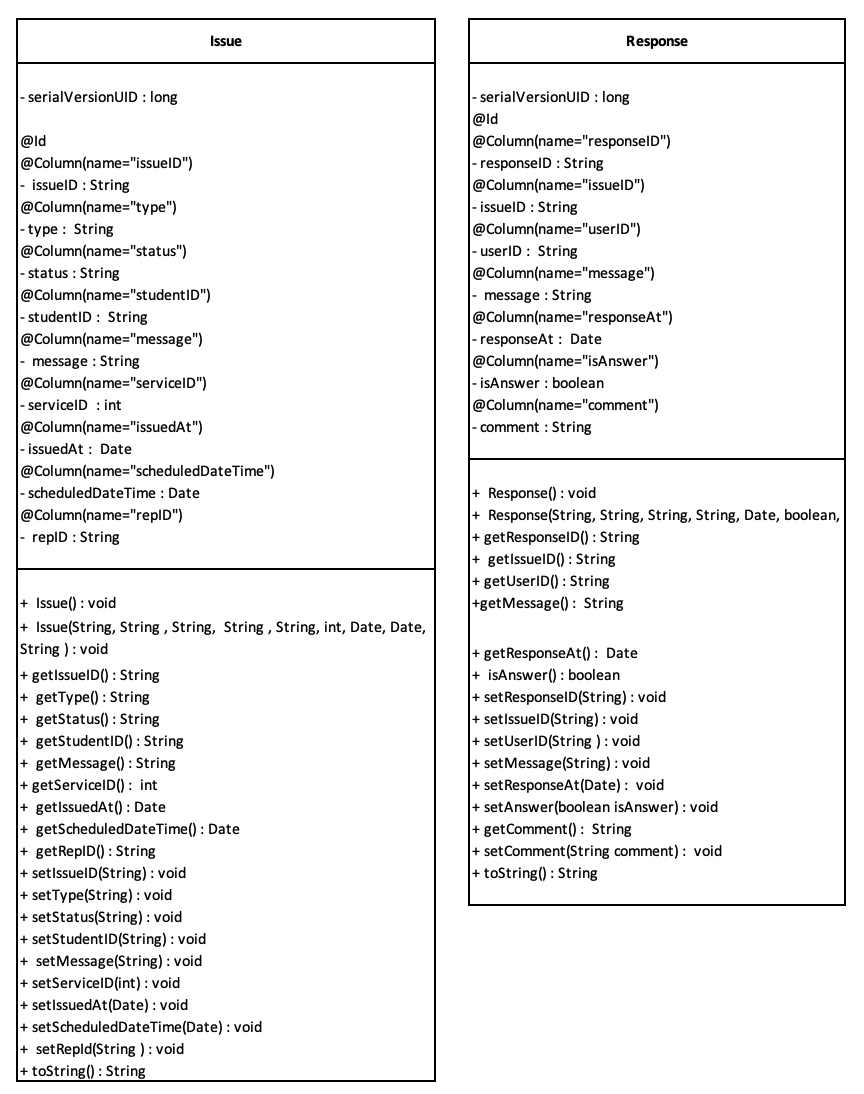


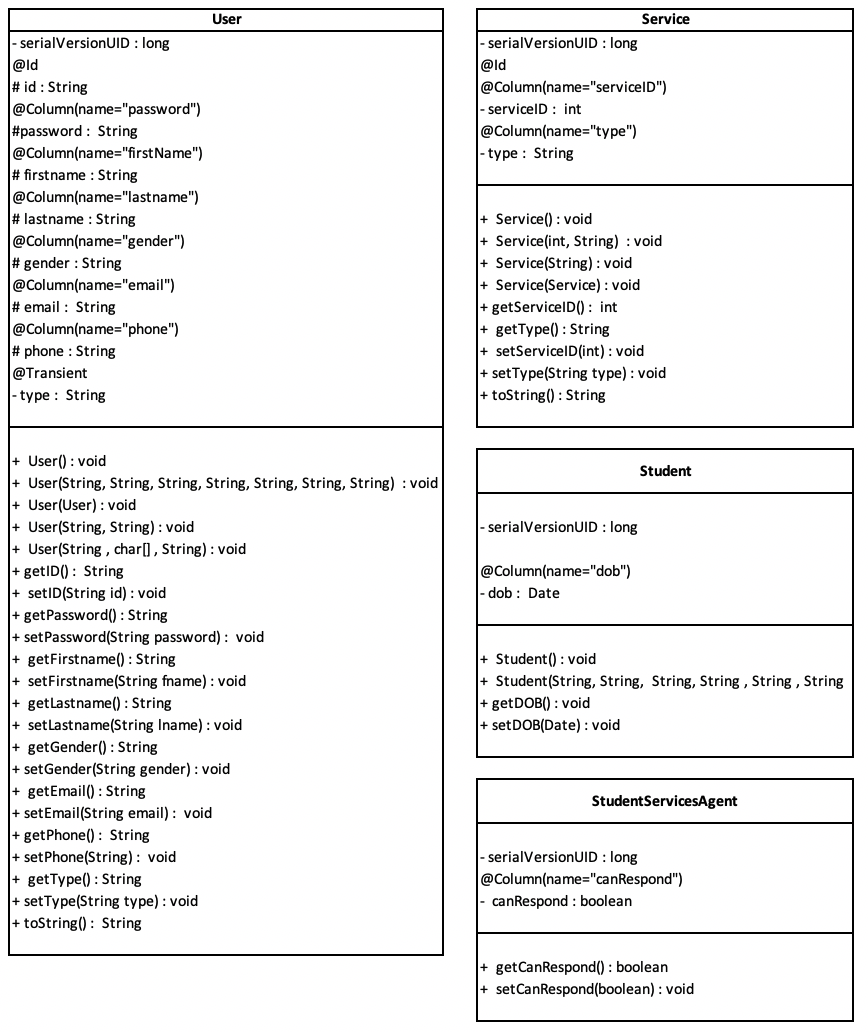


SEVER/ com.connectionFactories

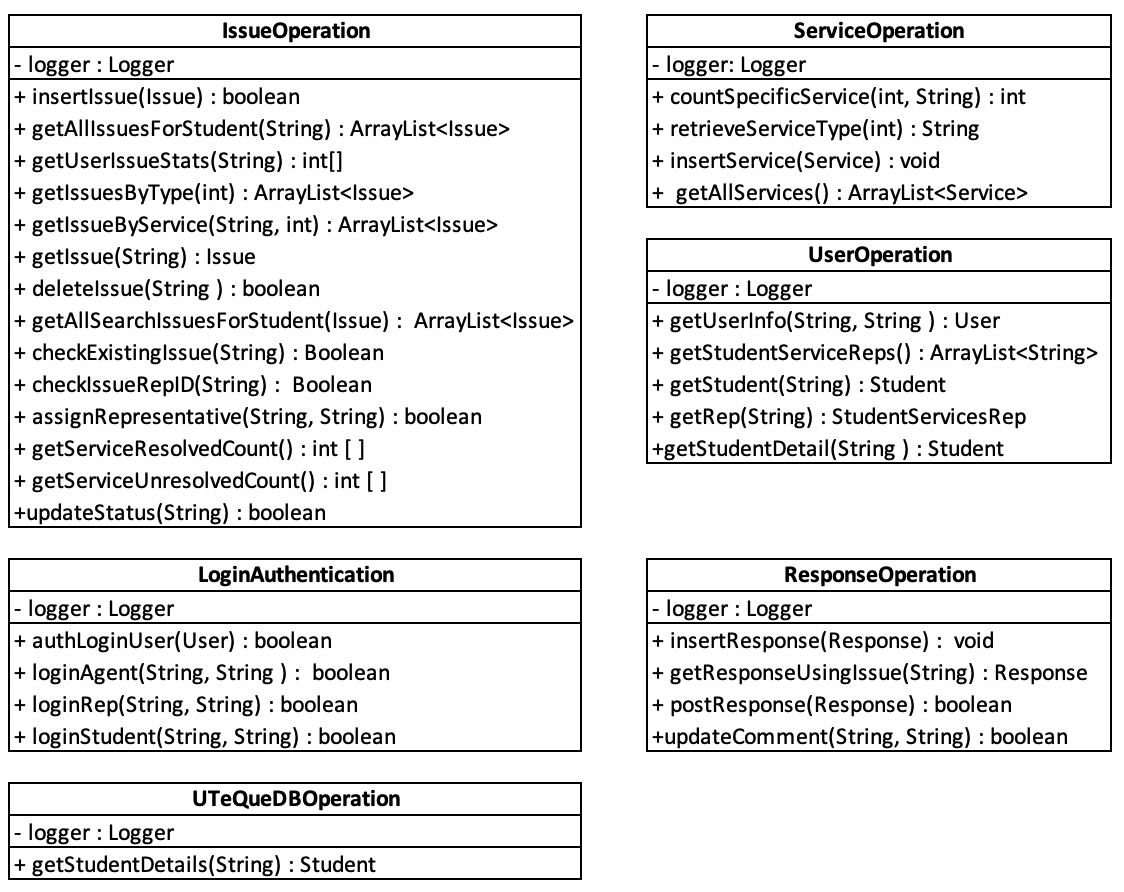


SERVER/ com.model





SERVER/ com.services

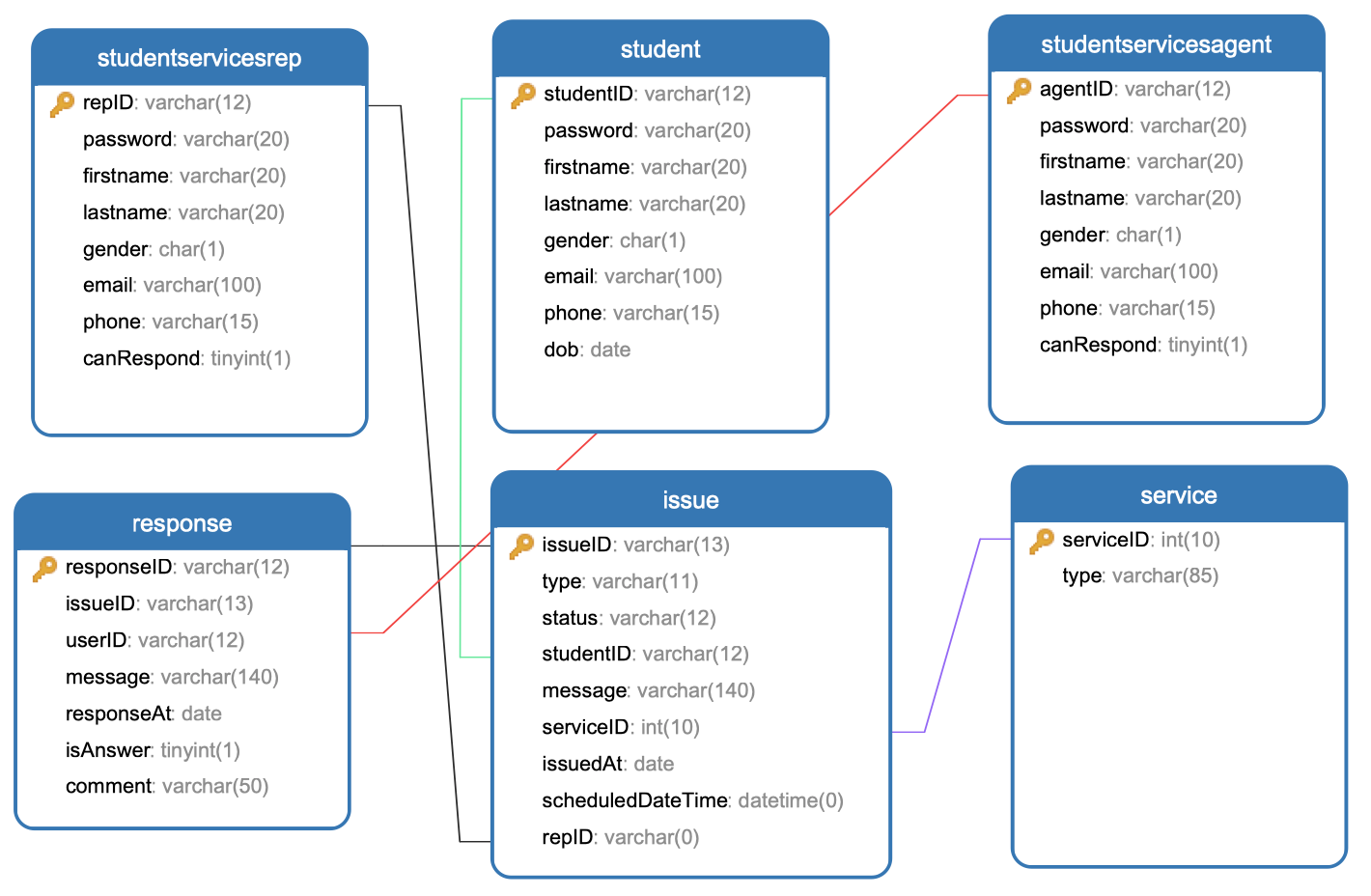


# 

# SEVER / server

# 

# **Represent relationships among classes using a UML relationship diagram**



The diagram above shows the relationship between the “studentservicesrep”, “student”, “response”, “studentservicesagent”, “response”, “issue” and “service” tables.

# 

# **Design of System**

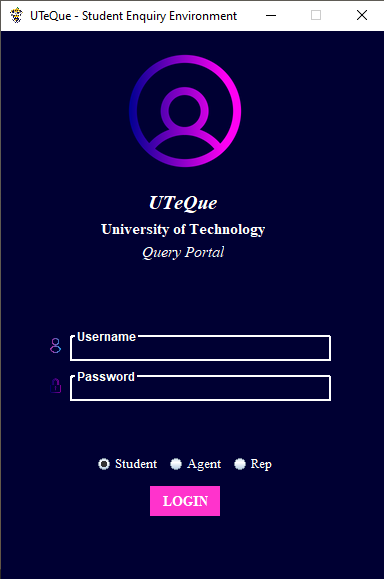


Figure 1.0 Login Screen

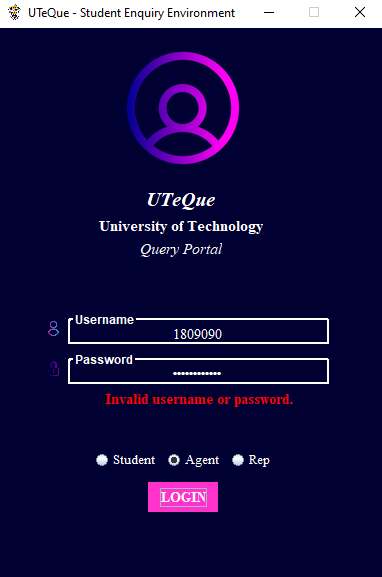
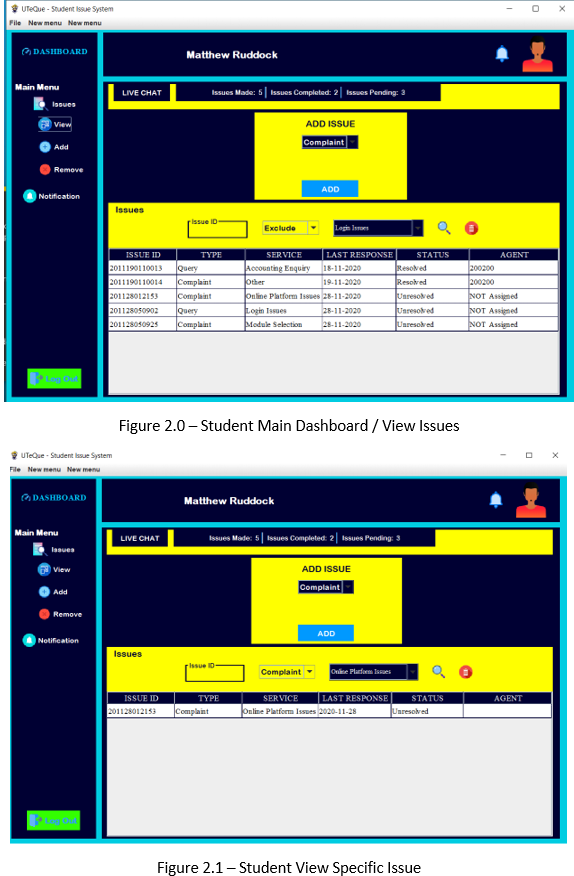
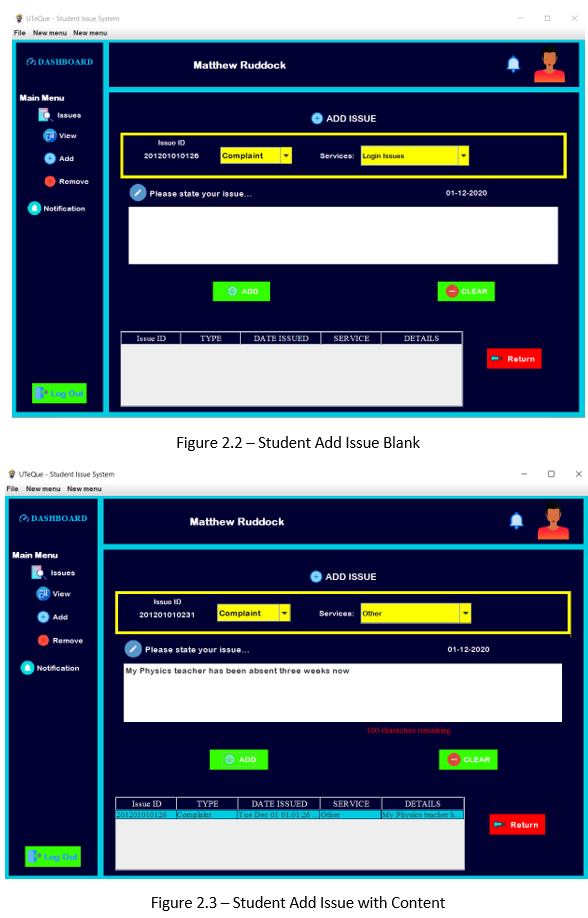
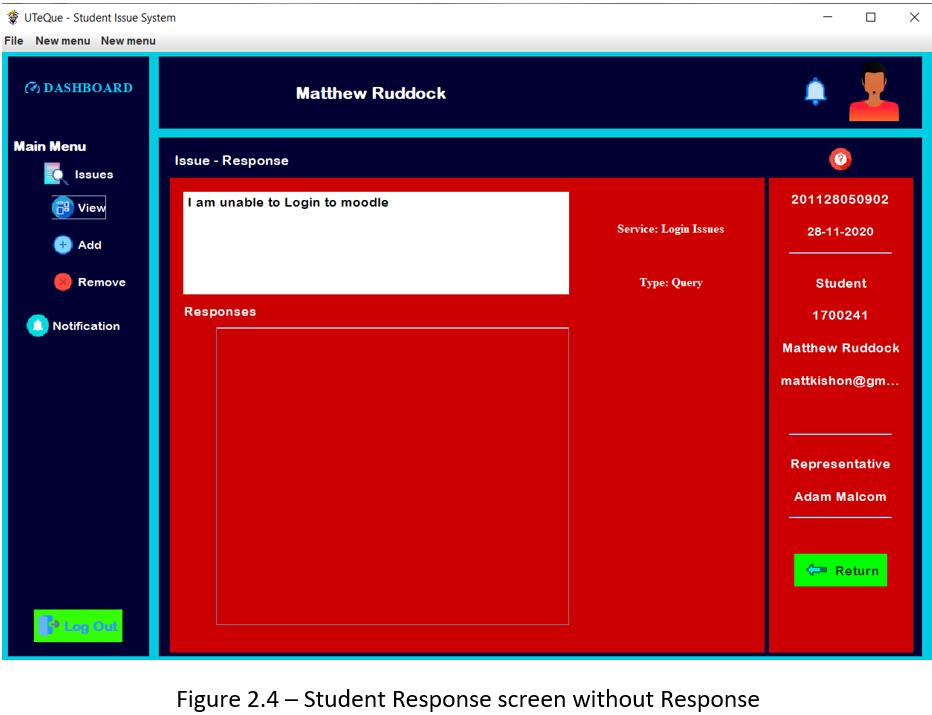
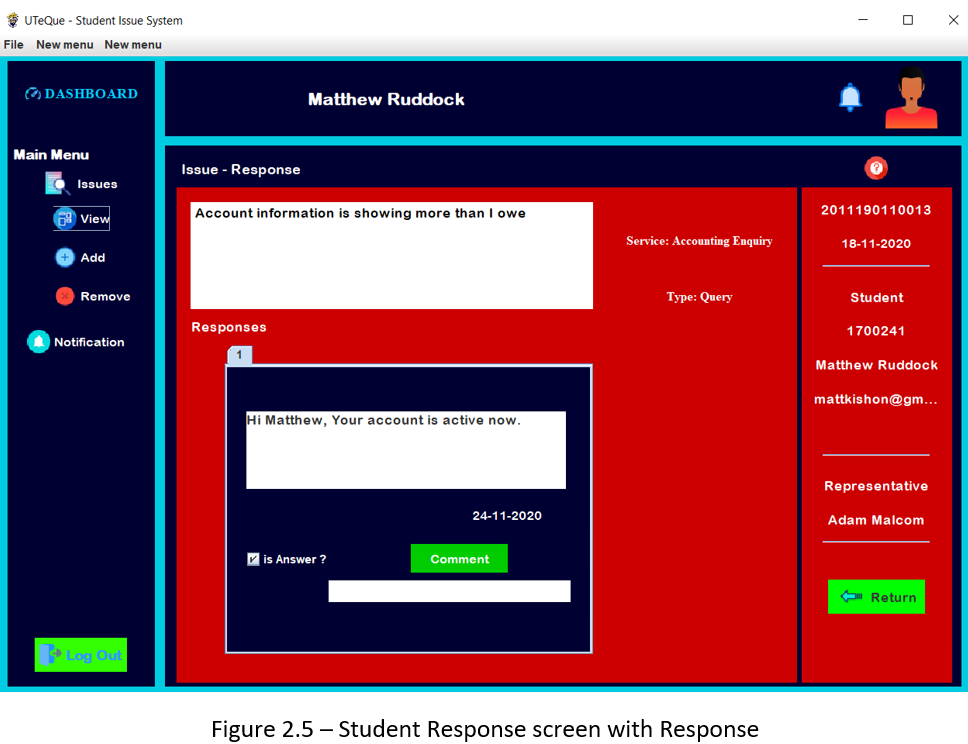


Figure 1.1 Login Screen - Failed Attempt









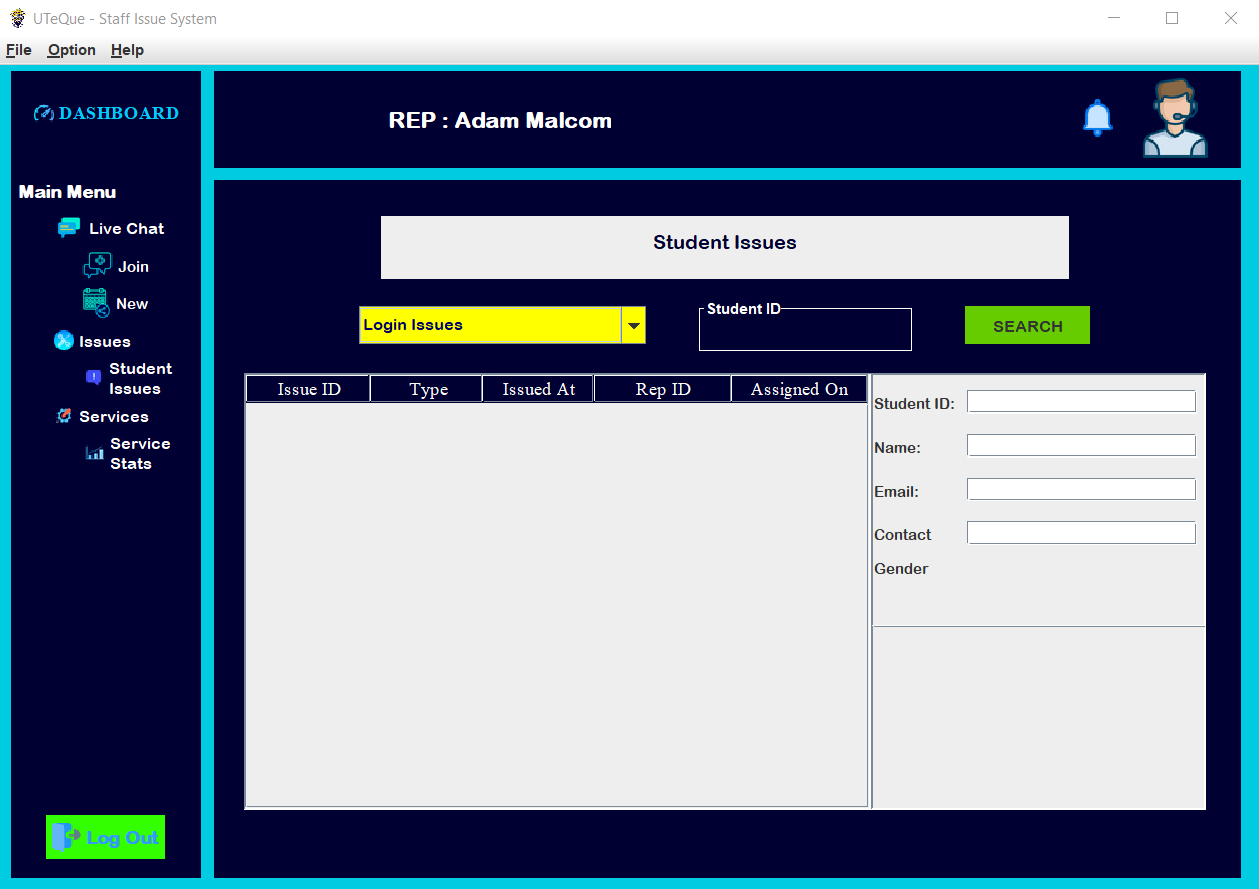


Figure 3.0 – Staff Main Dashboard

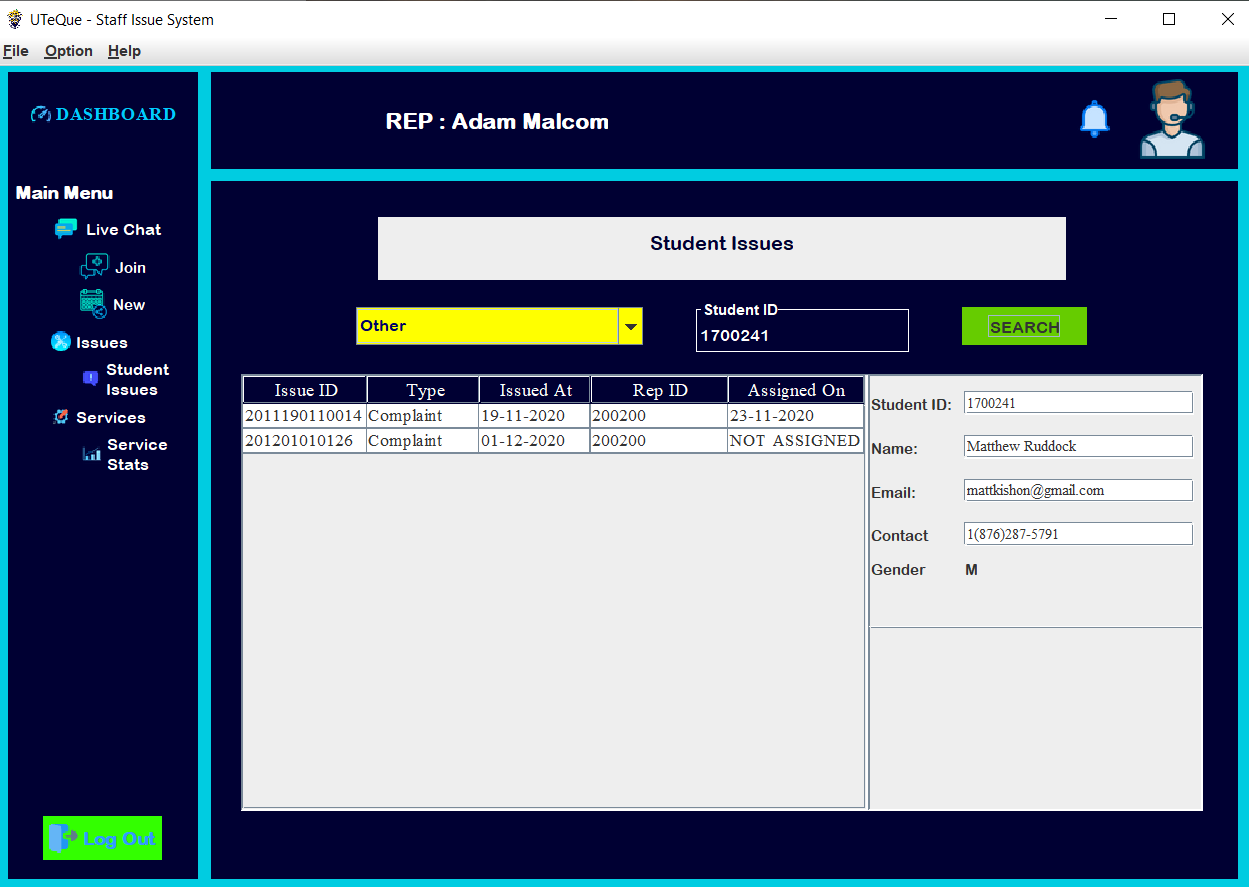


Figure 3.1 – Staff Main Dashboard

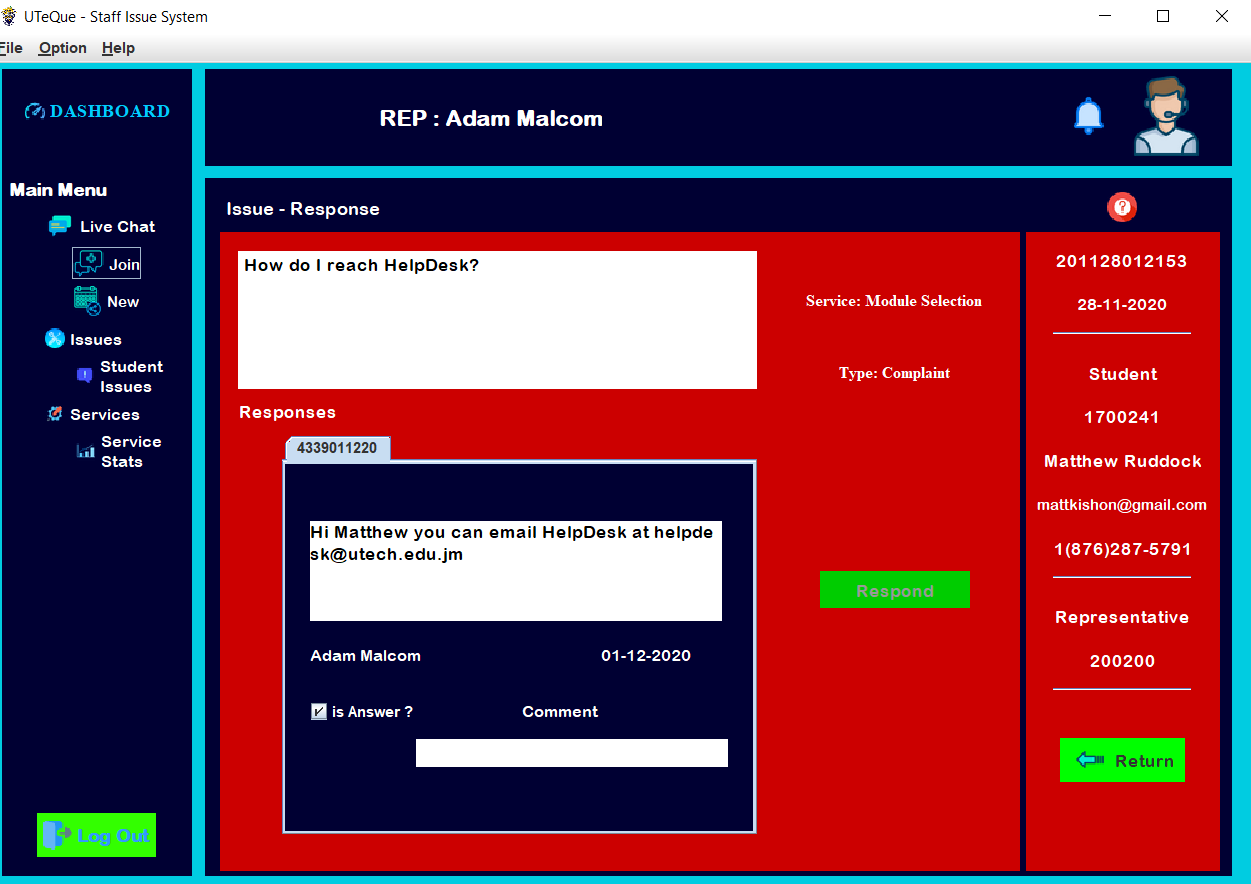


Figure 3.2 – Staff Issue – Response

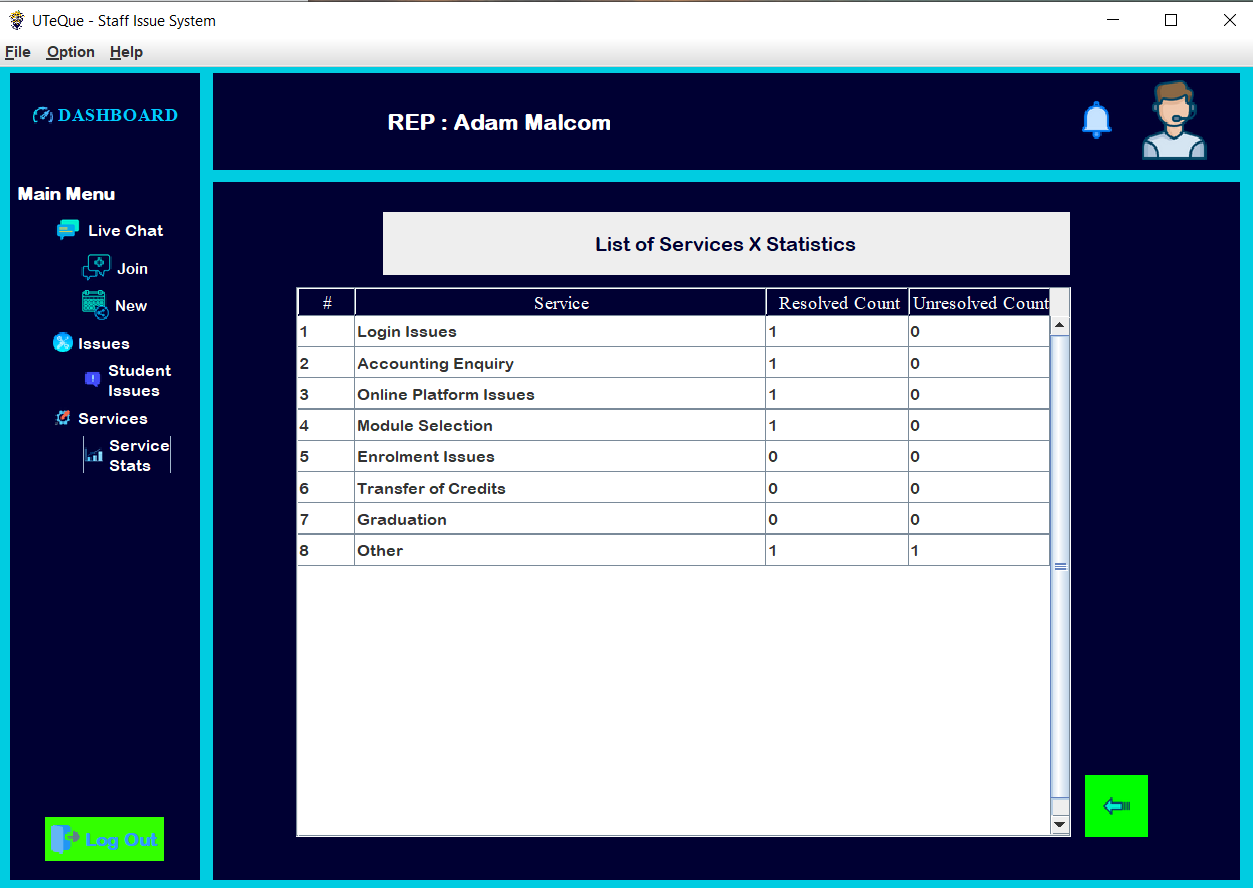


Figure 3.3 – Staff List of Services