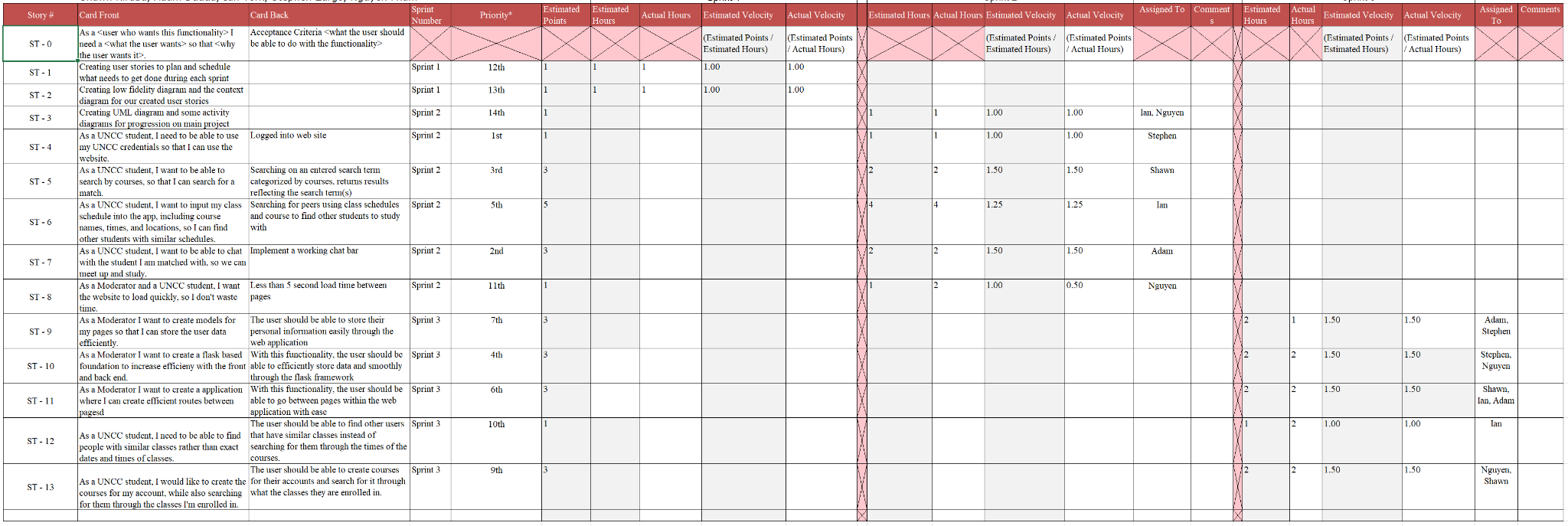
**SD2: Software Design Document**

**1. Project Overview**

Niner Connect is an application software that gives students the ability to share and view schedules with other fellow students. They can post their schedules, connect, and communicate with other students who have similar classes. Our stakeholders include professors and students who have desired this in the past. Our software will address the problem by making a site where the students can share their schedules in a social media format and they can also communicate with other students who have the same classes. There will be ways to find other students within the site to connect with and view their schedules by providing the students with profiles.

<https://github.com/skiruba/ITSC4155_MDSp24_Group4>



**User Stories:**

As a UNCC student, I need to be able to use my UNCC credentials so that I can use the website.

As a UNCC student, I want to be able to search by courses to search for a match.

As a UNCC student, I want to input my class schedule into the app, including course names, times, and locations, so I can find other students with similar schedules.

As a UNCC student, I want to be able to chat with the students I am matched with, so we can meet up and study.

As a Moderator and a UNCC student, I want the website to load quickly, so I don't waste time.

As a Moderator, I want to create models for my pages so that I can store the user data efficiently.

As a Moderator, I want to create a flask-based foundation to increase efficiency with the front and back end.

As a Moderator, I want to create an application where I can create efficient routes between pages

As a UNCC student, I need to find people with similar classes rather than exact dates and times of classes.

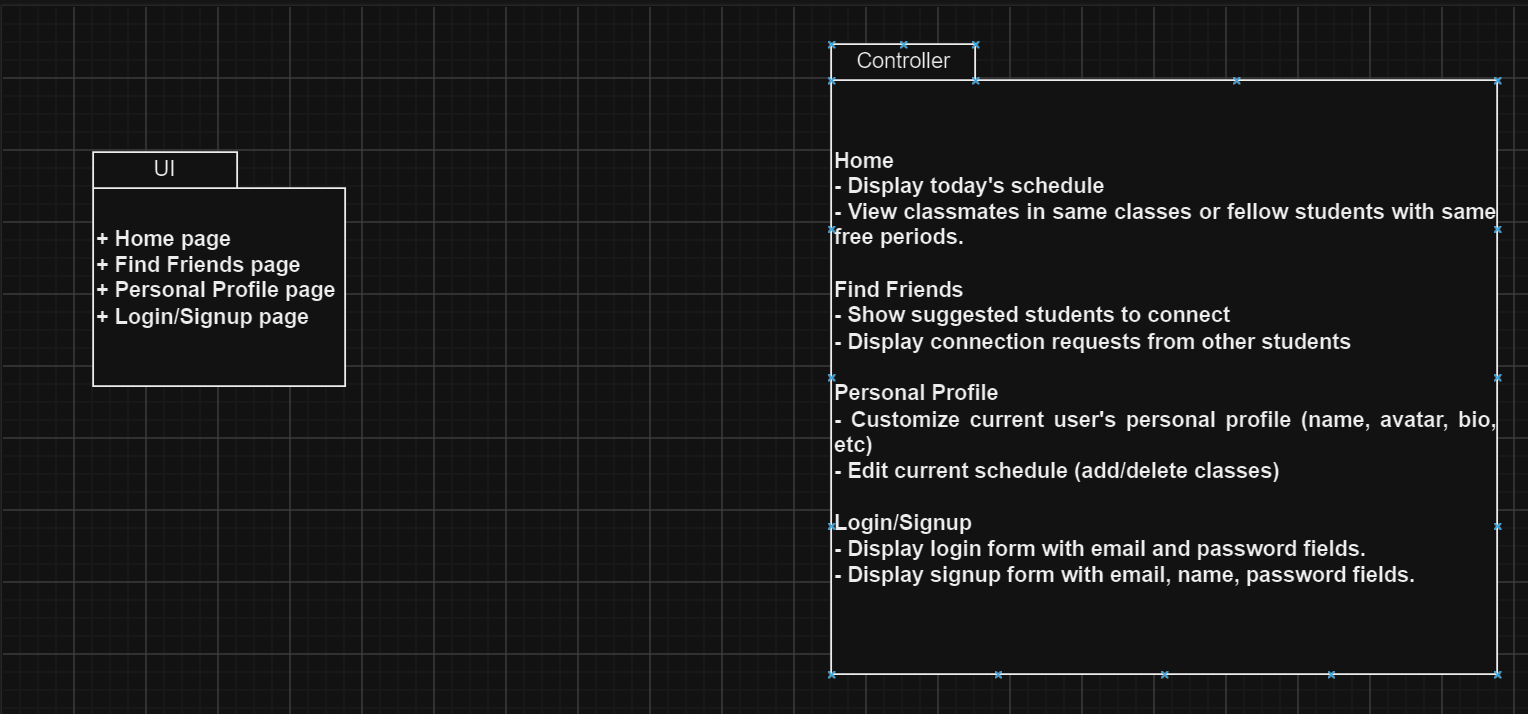
As a UNCC student, I would like to create the courses for my account, while also searching for them through the classes I'm enrolled in.

**2. Architectural Overview**

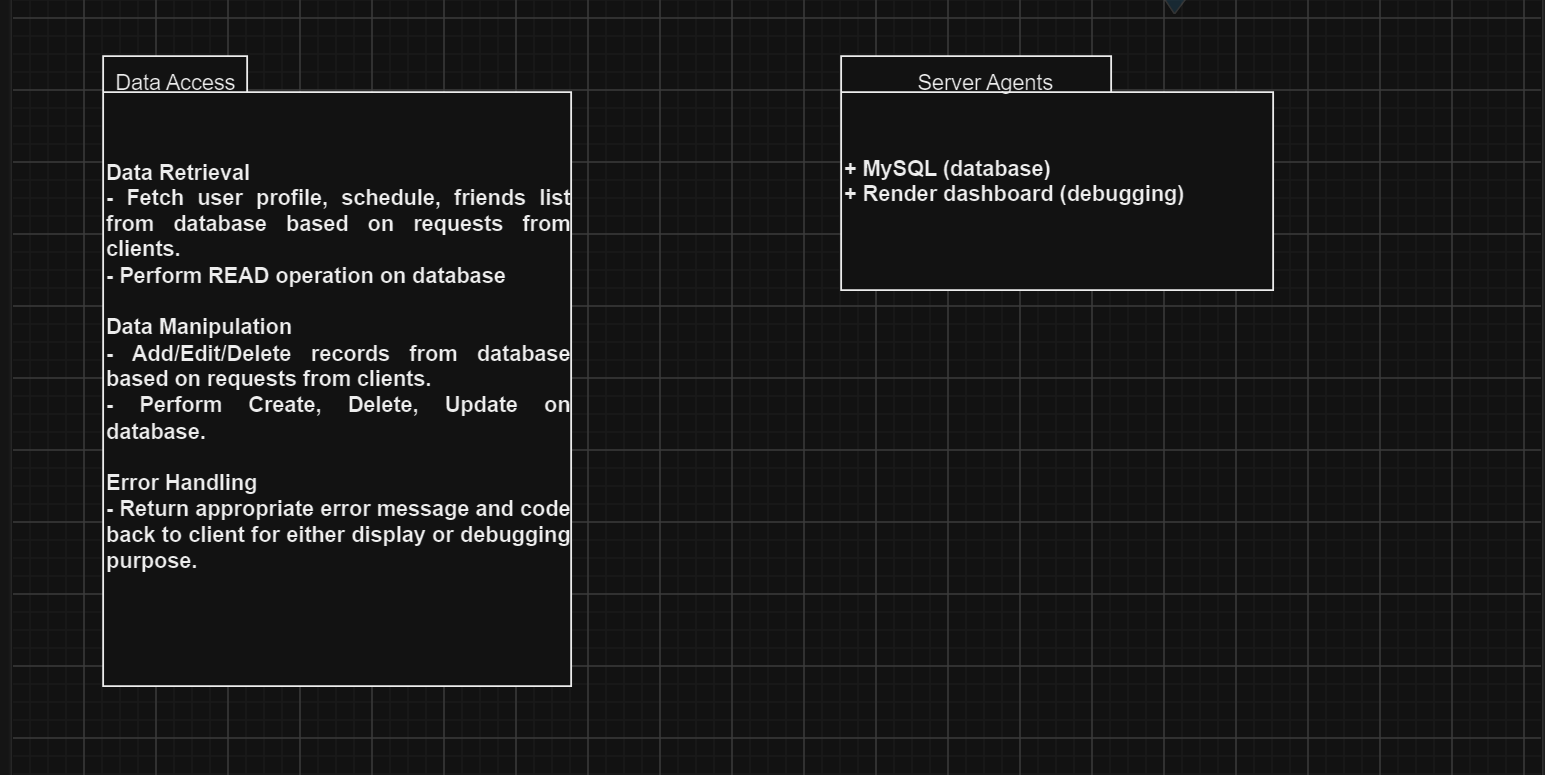
We went with the client-server architecture because it is straightforward and provides great scalability, security, and reliability that is beneficial to the development process in both short-term and long-term aspects. Basically having a client and server communicating back and forth helps establish the workflow nicely that is easily scalable and debuggable. Other architectures that we considered included microservices and event driven architecture. We decided that client-server architecture made the most sense for this application because having all resources in a centralized location is typical for applications of this nature. Client-server architecture will allow for easy maintenance, scalability, and security. These alternative architectures would've been uncommon for an application of this nature due to the services provided. Our system will include the ability for students to upload their schedule information. This information from all users will be stored on the server, which will allow users anywhere to access the information at the same time. By using a client-server architecture, we are able to take in information from all users, process their information to provide schedule matches and friend recommendations, and then return this output to the client without storing this information locally for each user. Overall, it makes the most sense for the data and processing to take place on a server and return the output to the client. This further allows us to secure important user information, scale the service as it grows, upgrade the service, and simplifying the service for the user.

**2.1 Subsystem Architecture**

* Client:
* User interface for students to interact with the application, such as viewing schedules, posting their schedules, and communicating with peers.
* Handling user input and displaying information received from the server in a user-friendly manner.
* Managing user authentication and session handling (e.g., logging in, logging out).
* The client communicates with the server to request data, submit user actions (such as posting schedules), and receive updates.

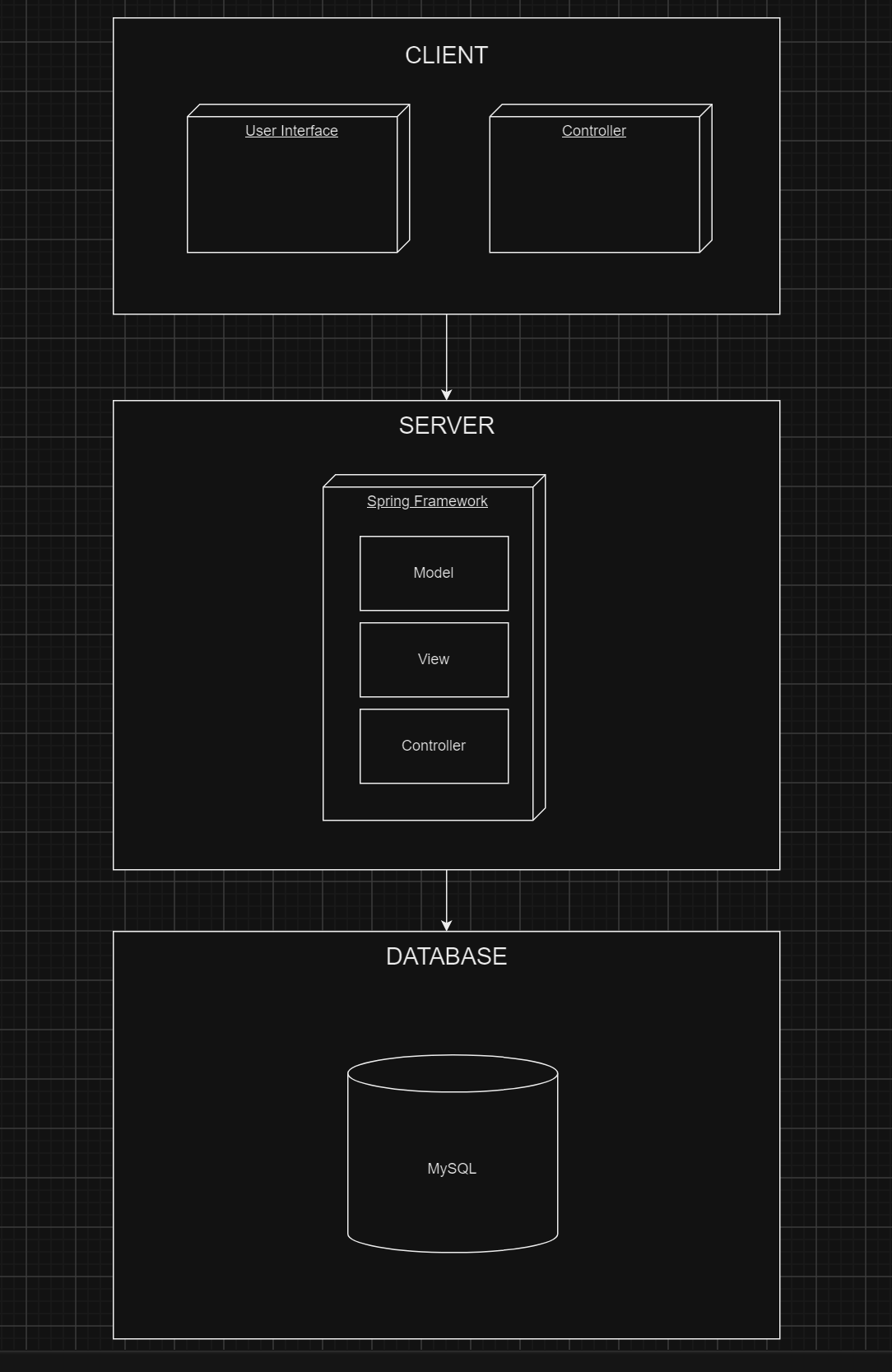


* Server:
* User authentication: Verifying the identity of users and managing access control to ensure only authorized users can access the system.
* Schedule management: Storing and organizing schedules posted by students, as well as facilitating schedule sharing between users.
* Communication management: Handling messages and notifications exchanged between users, such as chat messages or notifications about schedule changes.
* Data storage: Managing the database where user profiles, schedules, and other relevant data are stored securely.
* The server processes requests from clients, retrieves or modifies data as needed, and sends responses back to clients.



**2.2 Deployment Architecture**

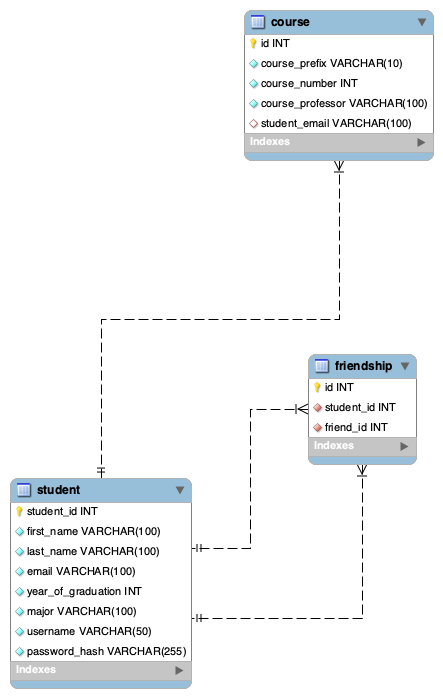
The client will handle data from user requests and communicate with the server to establish appropriate actions (displaying data, sending form data). The server will handle user authentication, persist user sessions, and manipulate data based on user requests, utilizing Spring Framework and MVC (Model, View, Controller). The database will be used to store relevant information and communicate with the server via JDBC Driver.



**2.3 Data Model**

Our system utilizes a SQL database with the following schema, as depicted below. The main table is 'Student', where each student is uniquely identified by their 'student\_id'. The 'Student' table stores essential information about each student, including their 'first\_name', 'last\_name', 'email', 'year\_of\_graduation', 'major', 'username', and 'password\_hash'. Additionally, each student can have multiple friends, facilitated by the 'Friendship' table. This table tracks friend relationships with 'student\_id' as the initiator of the friend request and 'friend\_id' as the recipient. Moreover, students can add their class schedules, which are stored in the 'Course' table. Each course is identified by a unique 'id' (CRN) and includes details such as 'course\_prefix', 'course\_number', 'course\_professor', and 'student\_email', which serves as a foreign key referencing the student who added the course. This updated schema reflects the current structure of our application.

Current Schema:



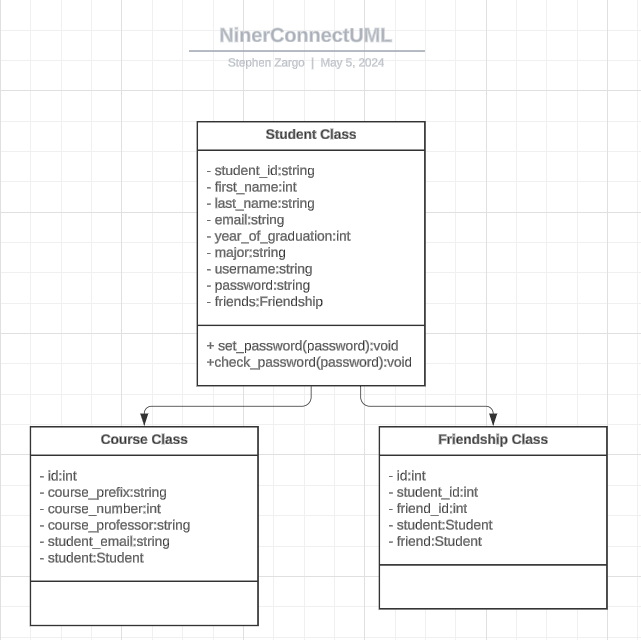
**2.4 Global Control Flow**

Our system’s execution is event-driven. The system will operate based on events prompted by the user. For example, the user may want to add a class. At this point, the system will prompt the user for class information. The user will also be able to choose to view their schedule, view other students, or change their profile information. Each of these scenarios represents events that the system would respond to. Most of these events are caused by user actions, such as button clicks or key presses. There are no time-controlled actions in our application, although there are many event responses inside of the application that will search and find classes and students that are in similar classes. In terms of concurrency, there are a few different components such as the calendar feature which the user can not only use to view their calendar and their classes for the week but they are also able to view the students that they are friends with that are in those same classes.

**3. Detailed System Design**

The design that we went with for the web application is very straightforward and has pages that are immediately accessible to the user as well as effective for the application. We have a navbar that the students can access where they can visit pages such as home, today’s schedule, add classes, find friends, and my friends. These were all made for our website where students can share schedules with their friends while also finding more friends that have similar classes in case they need to reach out to other students.

**3.1 Static view**

****

So, as you can see we really only have 3 main classes in our Web Application: the Student Class, Course Class, and Friendship Class. Each one acts as an entity that contains data about itself and the other 2 are all centered around the Student Class. This is pretty intuitive as it makes sense for the Student Class to contain fields such as the student ID, first and last names, year, major, and all other important information relating to the student. The student class has a relationship with the Course Class in which a Student can have a Course, and the course contains information about itself such as the course id, prefix, professor, and course number. The 3rd class we have is a Friendship class that contains information about Friendship relationships between 2 mutual friends including the unique friendship id, student one’s id, the friend’s id, and the 2 student objects that are friends. The Student Class’s relationship between Student and Friendship is that a student has friendships. These are the Classes in our system, and most of the methods and logic happens within the routing of our system so that’s why there’s not many contained in the classes.

**3.2 Dynamic view**

Here in our sequence diagram we have modeled the different actions the user can take in our system, and the interactions between the systems in it when that happens. We have the login and registration page where the user enters their information to create an account in the system. Then we also have represented what happens when a user adds a course in the system, and all the checks it goes through. Next we have what happens when a user adds a friend in our system, and the potential checks to make sure that the student they are trying to friend is in fact in the system.

