University Registration System Design Project

Analysis Phase

Team Just Do It

Team Member Information

Team Leader	Andrew Turrentine	att0002@auburn.edu
Requirements Analyst	Mike Miller	mwm0030@auburn.edu
Architect	Jonathan Dummer	jmd0050@auburn.edu
Designer	Riley Emnace	rae0011@auburn.edu
Programmer	Ian Thomas	igt0001@auburn.edu
Programmer	Taylor Hartley	twh0008@auburn.edu

Table of Contents

*Note: Use Cases include Application Interaction Model and Interaction Design (Collaborations)

Concept Statement

Design software to support a registration system that is hosted on a domain so that the server can be accessed by a user via the internet. The registration system consists of a server and a database that stores user and course information. The registration system should be able to be used by multiple universities that will enable students to register for classes and also allow professors to access information about the classes being taught. Each university will provide its own design input for the graphics and presentation of the system. The system should retain information about previous schedules as well as a database of current students and professors that will be stored on a local computer at the university.

There are three types of users: student, professor, and admin. A student user represents a person attending the university and currently taking courses. A student user inherently has a schedule which consists of the courses the student is taking. These courses have a set of times that the class takes place, as well as a professor. The student is also assigned a grade (by the professor) for each course he/she is taking.

Additionally, reports detailing student information, past/current courses, course grades, and computed grades (i.e. GPA) may be requested by the user, which can then be generated by the server and displayed to the user.

For students, the system will provide menus for the following: selecting the “semester” they wish to view/edit, select the type of courses they wish to view, view classes being offered that fit the selected type, view the specific time and place for the selected class, add/drop the class (dependent on time of viewing) and view current schedule. The student’s information, such as registration status and current classification, will be linked to this page and possibly displayed for the student to view but will not have menu representation. Students will also have an option to view their past schedules, if necessary, to confirm eligibility for future class schedules.

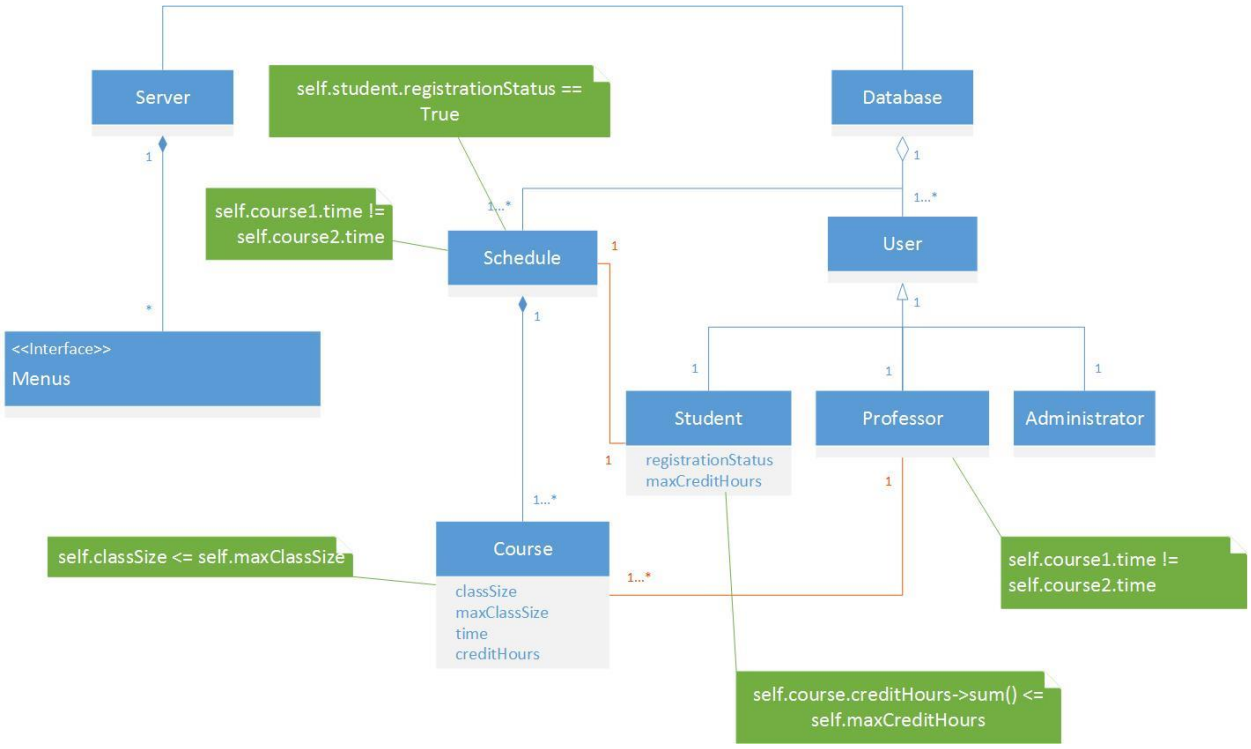
The second type of user, professor, represents a person employed by the university to teach courses to the students, as well as update the grades of each student in each course (the professor is teaching). Similar to a student, a professor also has a schedule containing courses the professor is teaching and the times at which the course takes place. Along with this, the professor also has access to the students that are taking the courses that he/she is teaching. This includes access to their grades, which can be updated by the professor. Information pertaining to past courses (similar to students) should also be able to be accessed by the professor.

The professor will log into the system through the same log in system as students, but the database will denote the professor status and allow them access to a different set of information than students. For professors, the system will provide menus for the following: view list of classes being taught, view list of students in each class, assign assistants to classes (if applicable) and grading. Most selections for professors will be "view only" since they will not need to add or drop any classes or students. The grading menu, however, will need to be fully interactive with all assigned grades connected to specific students (by name or student number).

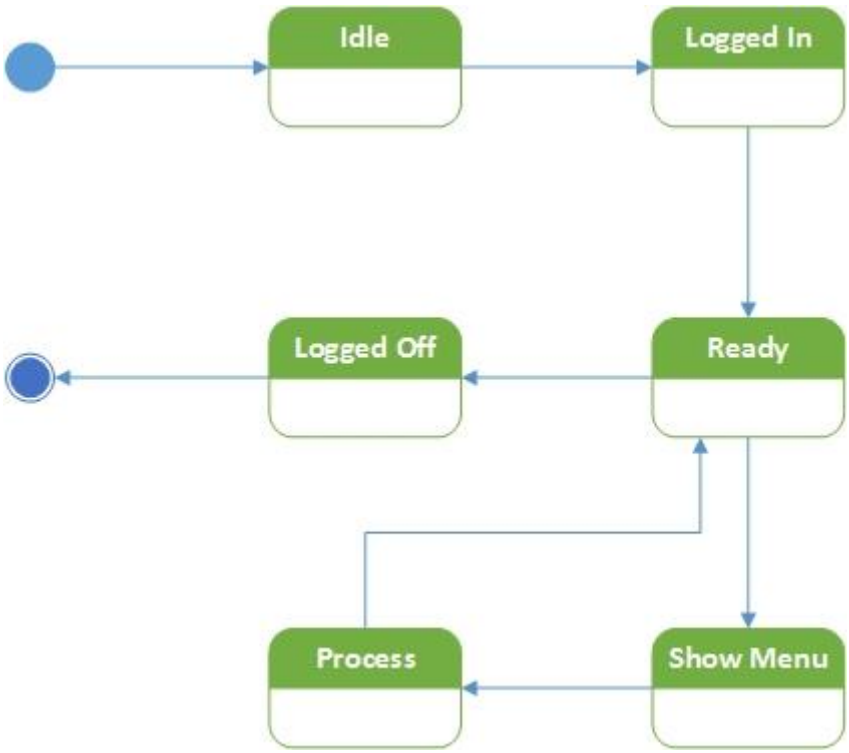
The final type of user, admin, represents a person that acts as an administrator for the registration system. This type of user has special administrative privileges in order to manage other users (of type student and professor). The administrator is able to access and edit the database freely without restrictions. This includes operations such as manually adding/removing users or courses.

The university will provide the computer to house the system database(s) and the software required to run that computer. All information from the system should be stored locally and not be available for public access. Only current students, current professors and university administrators will be able to access information within the databases (through use of a userID provided by the university and a user controlled password).

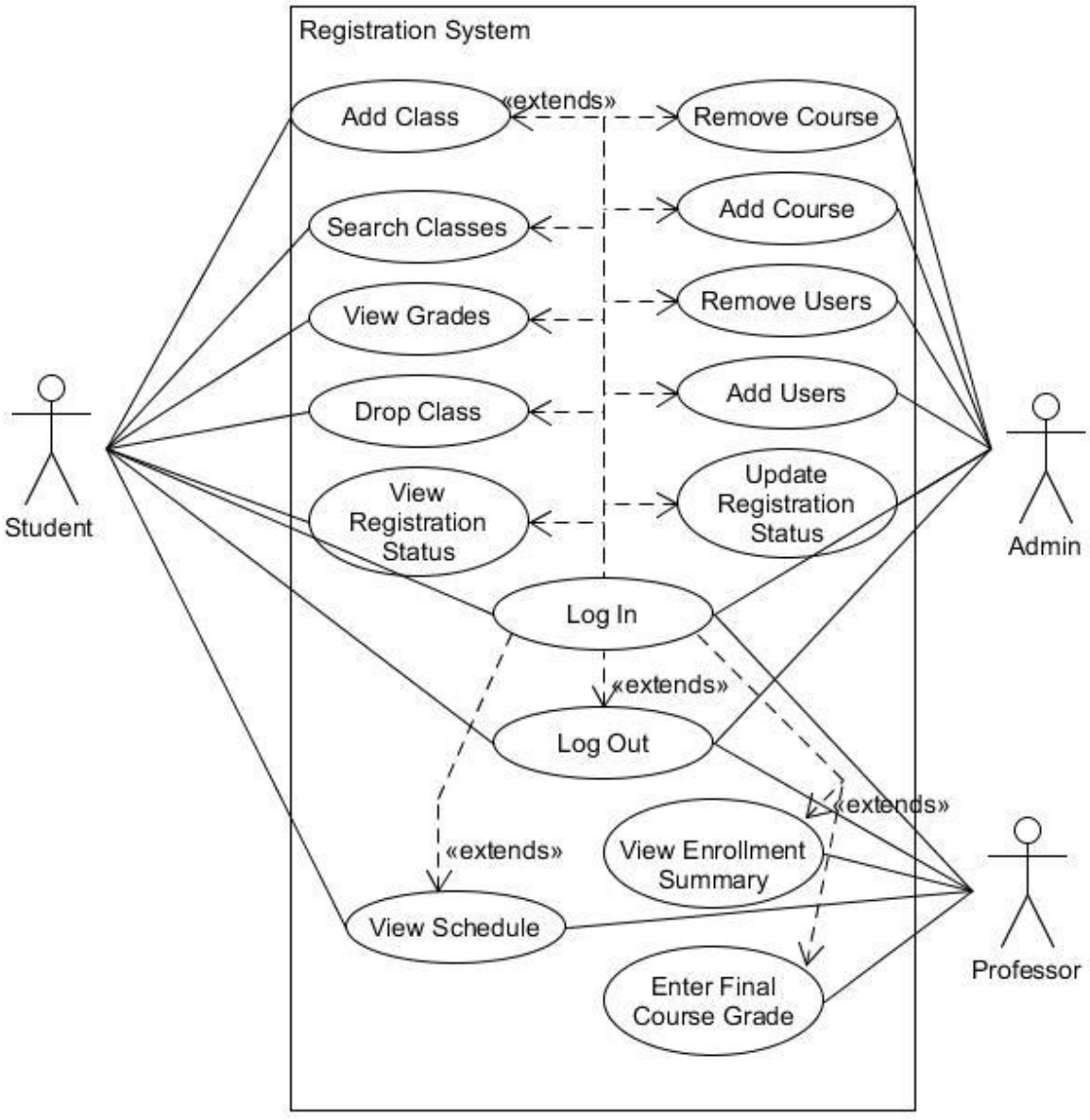
Concept Domain Model



Domain State Model



Use Case Diagram



Use Cases

Add Class Use Case (Essential)

Participants: Student, Registration System, Database

Pre-Condition: The User has logged in successfully as a student, user has navigated successfully to a class through search classes use case.

Typical Course of Events

Actor Intentions	System Responsibility
1. Student decides to add the course	2. If students schedule is free at that time, add course to schedule
	3. If course enrollment not full, add 1 to enrollment
	4. Display completed action
5. Student confirms course has been added to schedule	

Alternative Courses

- Step 2: Student schedule is not free at the time. Notify student of conflict. Display the conflicting course and course ID number to the student. Abort add request.
- Step 3: Course enrollment is full. Notify student that the course is full and that they have been "waitlisted". Add the student to the wait list and increment waitlist counter.

Post Condition: The student has 0 or 1 additional courses

Add Class Scenarios

Normal “Add Course” Scenario

1. Student request the course be added to his or her schedule
2. System adds course to schedule
3. A course addition confirmation for the student to view is presented

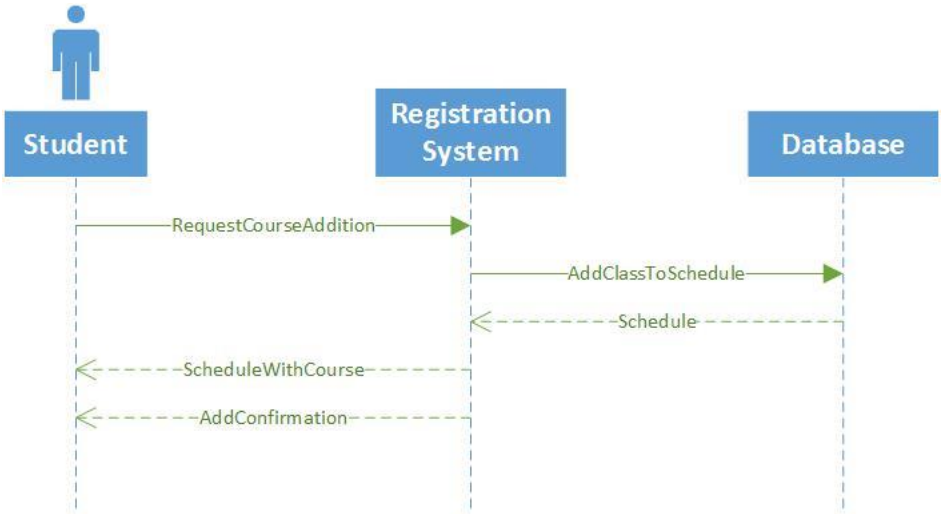
“Add class” Conflicting Time Scenario

1. Student request the course be added to his or her schedule
2. System detects and displays a time conflict to user
3. Class is not added

“Add class” Course Enrollment Full Scenario

1. Student request the course be added to his or her schedule
2. System detects and displays that the class is full
3. Class is added to schedule and marked as waitlisted

Add Class High Level SSD



Add Class Use Case (Concrete)

Participants: Student, Registration System, Database

Pre-Condition: The User has logged in successfully as a student, user has navigated successfully to a class through search classes use case.

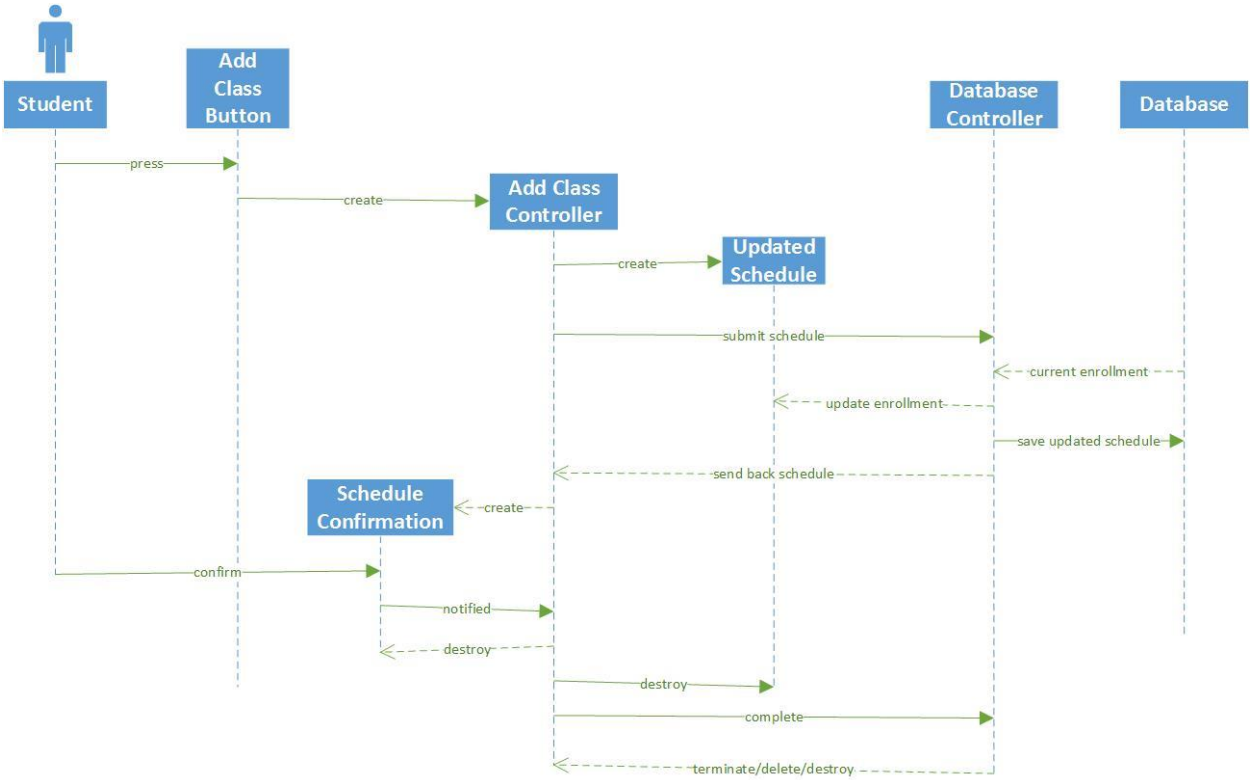
Typical Course of Events

Actor Intentions	System Responsibility
1. Student presses button on viewed course to add the course	2. Create a schedule with new course
	3. Get current course enrollment from the database
	4. Update course enrollment on new course schedule and add one to enrollment
	5. Save this new schedule to the database
	6. Create a confirmation page with new schedule for the student to see
7. Student confirms course has been added to schedule	

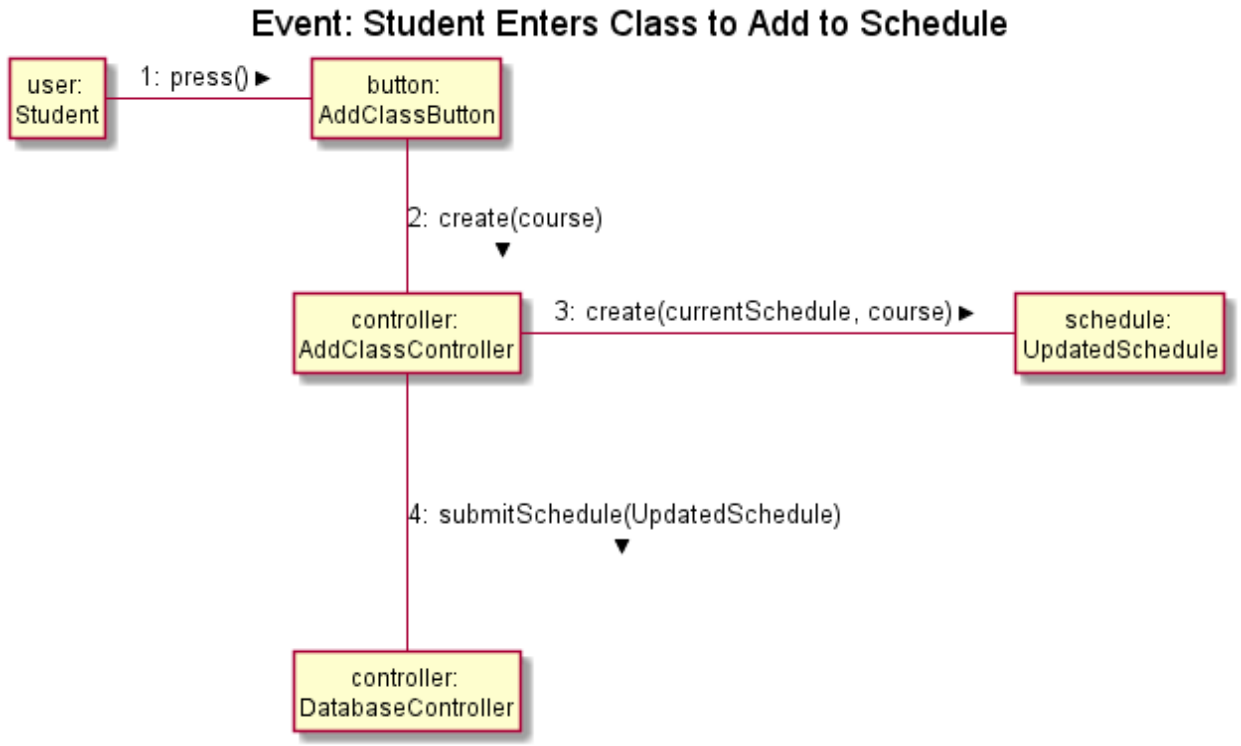
Alternative Courses

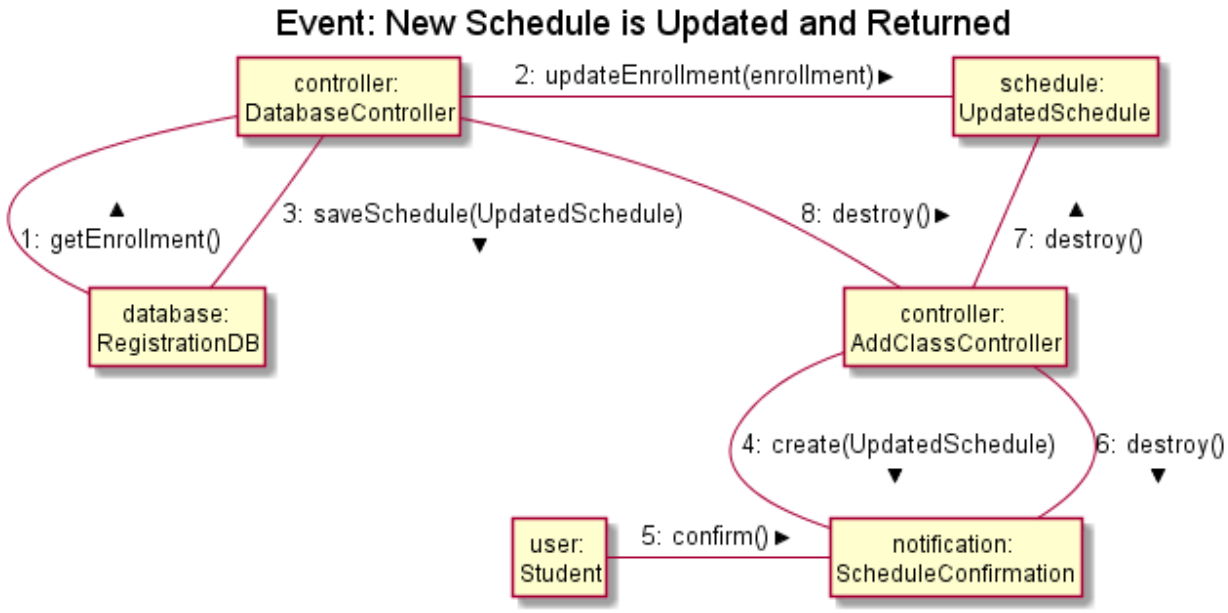
- Step 2: Student schedule is not free at the time. Notify student of conflict. Display the conflicting course and course ID number to the student. Abort add request.
- Step 4: Course enrollment is full. Notify student that the course is full and that they have been "waitlisted". Add the student to the wait list and increment waitlist counter.

Add Class Detailed SSD



Add Class Collaboration Diagram





Add Class GRASP Patterns

The first GRASP pattern used in the first collaboration diagram is creator. First, the add class button creates the add class controller which in turn creates the schedule and schedule confirmation notification. Perhaps the biggest GRASP pattern used is the controller, there is an add class controller that facilitates the UI and functionality specific to this use case and there is a database controller than manages interactions with the database.

In the second diagram, the controllers help reduce coupling through indirection. While the controller themselves may not be highly cohesive they delegate the work to objects that are highly cohesive like the Schedule Confirmation object or the Updated Schedule object. This helps achieve the low coupling GRASP pattern for this use case, since most of the information is flowing through the controllers as opposed to objects directly communicating and relying on each other.

Search Classes Use Case (Essential)

Participants: Student, Registration System, Database

Pre-Condition: The User has logged in successfully as a student

Typical Course of Events

Actor Intentions	System Responsibility
1. Student chooses to look for a course for further details about the course	
2. Student request for a list of courses	3. Present list of courses available during term that student is looking at
4. Students decides on a possible course and requests additional information about that course	5. Present information about course including the time, professor, and course ID number.
6. Student considers information about course and considers further actions	

Alternative Courses

- Step 3: No courses available in that term. Notify student of no courses not available.
- Step 5: Missing information. Mark all missing information fields as TBD (to be determined) before displaying information to the student.

Search Classes Scenarios

Normal “Search Classes” Scenario

1. Student request for a list of courses
2. System displays all available courses in that term
3. Student chooses a course to get additional information about
4. System displays additional information about the course

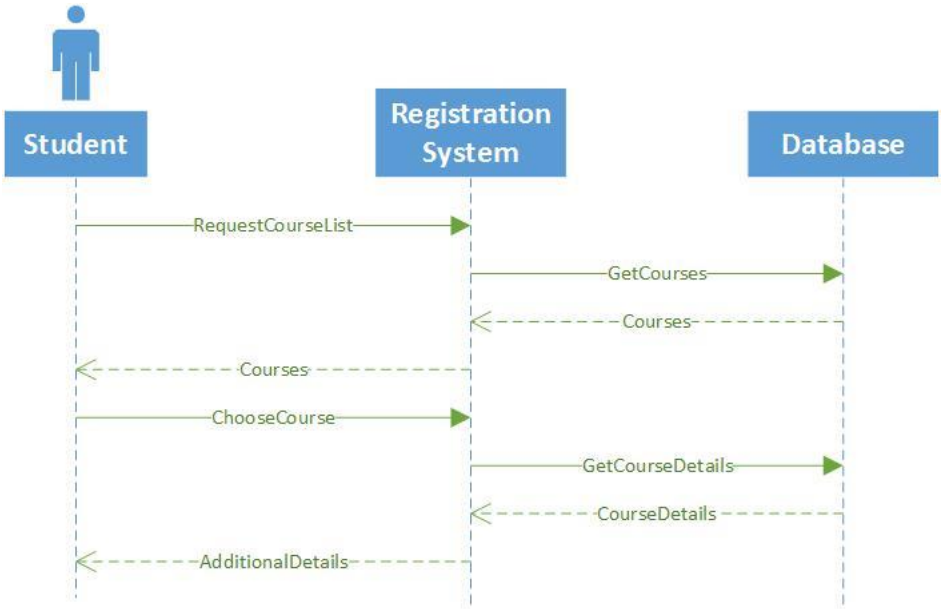
“Search Classes” No Courses Available Scenario

1. Student request for a list of courses
2. System displays that no courses are currently available for that term

“Search Classes” Missing Information Scenario

1. Student request for a list of courses
2. System displays all available courses in that term
3. Student chooses a course to get additional information about
4. System fills in missing information about course with TBD.
5. System displays additional information about the course

Search Classes High Level SSD



Search Classes Use Case (Concrete)

Participants: Student, Registration System, Database

Pre-Condition: The User has logged in successfully as a student and selected the search classes option

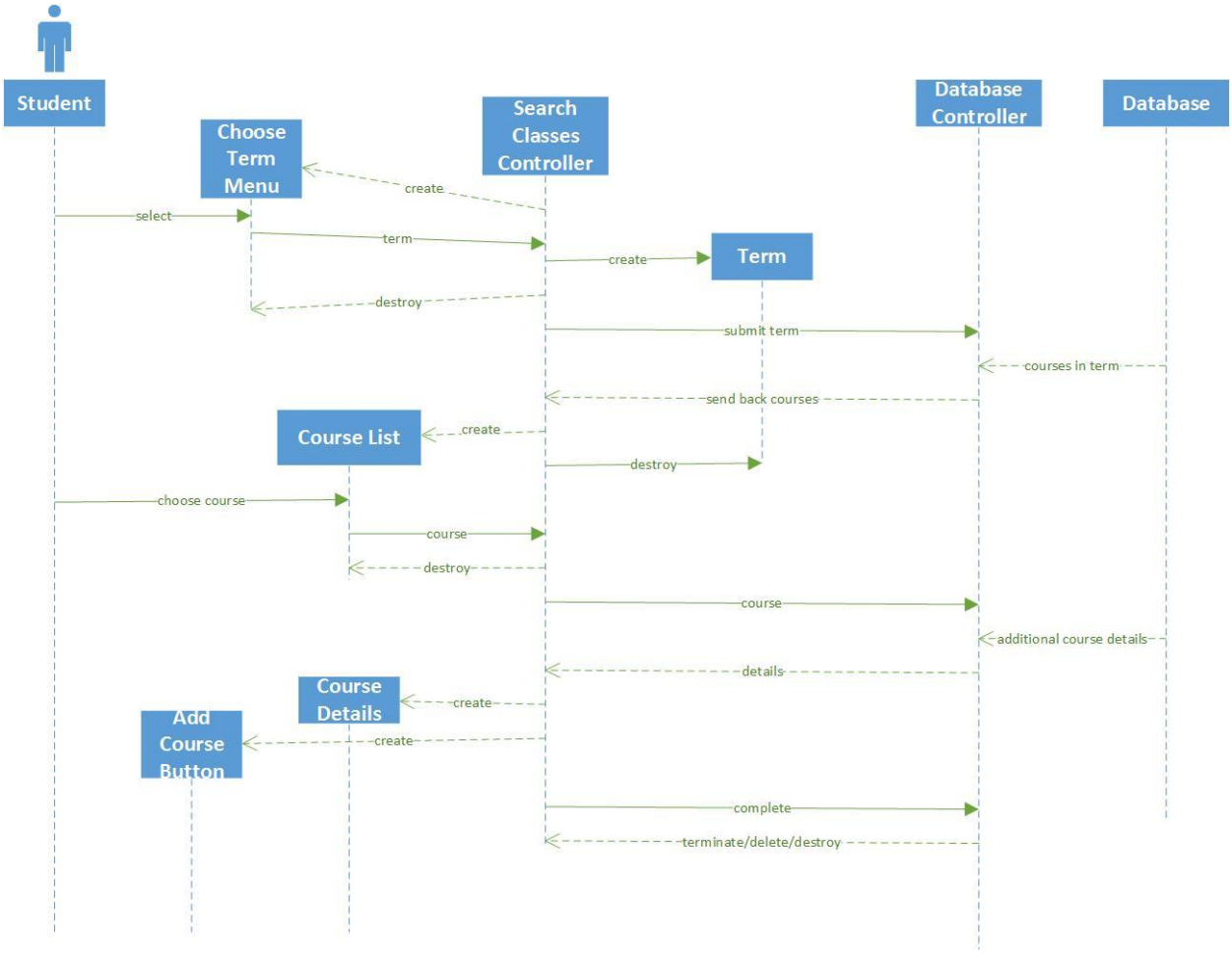
Typical Course of Events

Actor Intentions	System Responsibility
1. Student request for a list of courses for a term through drop down menu	2. Get list of courses available during term from the database
	3. Present list of courses available during term that student is looking at
4. Students decides on a possible course and requests additional information about that course by hitting a button	5. Send course to database.
	6. Get additional course information from database
	7. Generate course information page and display to the user
8. Student considers information about course and considers further actions	

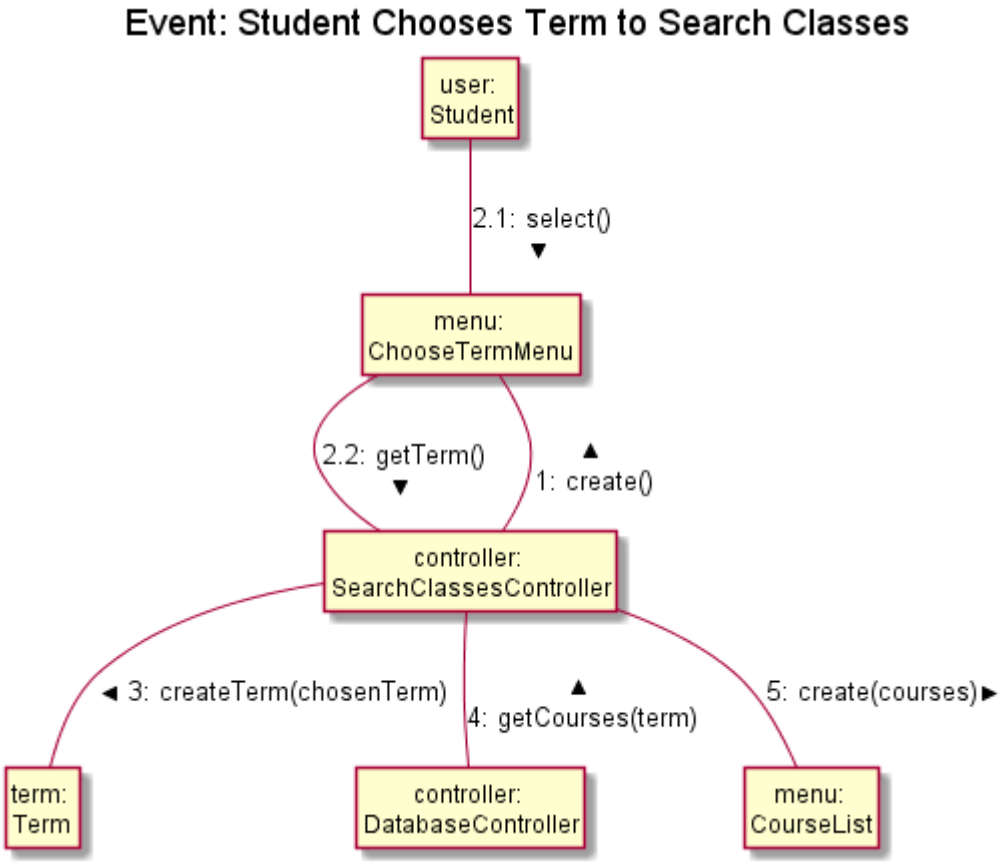
Alternative Courses

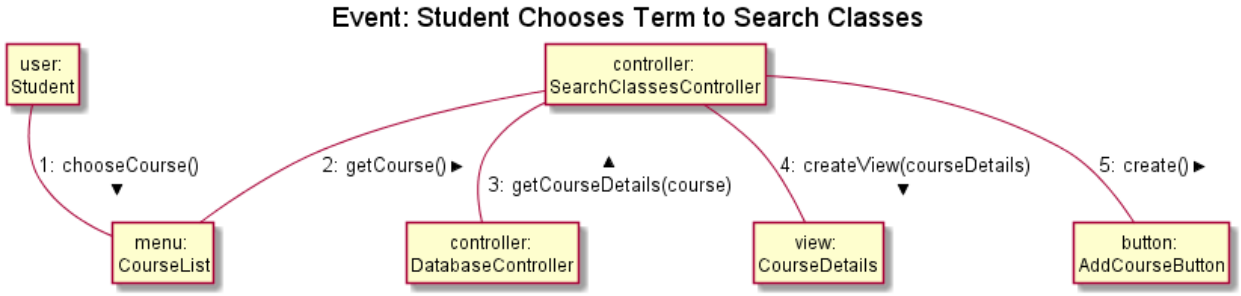
- Step 3: No courses available in that term. Notify student of no courses not available.
- Step 8: Missing information. Mark all missing information fields as TBD (to be determined) before displaying information to the student.

Search Classes Detailed SSD



Search Class Collaboration Diagrams





Search Class GRASP Patterns

The big two GRASP pattern used in these collaboration diagrams are creator and controller. First, the search class controller acts mainly as a creator in the first diagram, since its primary goal in the first diagram is to create the course list, term menu, and term. This helps keep the other objects high cohesive and reduce the coupling by reducing the communication between UI objects.

In the second diagram, the controllers help reduce coupling through indirection. The search class controller manages the operations and communication of boundary objects such as course list and also manages the flow of information between these objects (such as course details) so that these non-controller objects can be highly cohesive and focus on one thing. This helps achieve the low coupling GRASP pattern for this use case, since most of the information is flowing through the controllers as opposed to objects directly communicating and relying on each other.

Remove Course Use Case (Essential)

Participants: Administrator, Registration System, Database

Pre-Condition: The User has logged in successfully as an administrator

Typical Course of Events

Actor Intentions	System Responsibility
1. Administrator chooses to look for a course to remove from the system	
2. Administrator request for a list of all available courses	3. Present list of courses available
4. Administrator chooses a course and requests additional information about that course	5. Present information about course including the time, professor, and course ID number.
6. Administrator decides to remove the course	7. If no one is registered for course and no one is assigned to teach the course, remove course
	8. Display completed action
9. Administrator confirms course has been deleted from the system	

Alternative Courses

- Step 7: Students are registered for a course. Notify students who are signed up for the course. Drop students from course. Remove course.
- Step 7: Professor is assigned to course. Notify professor for that course. Remove that course from professor's class schedule. Remove course

Post Condition: There is 1 fewer course in the system.

Remove Course Scenarios

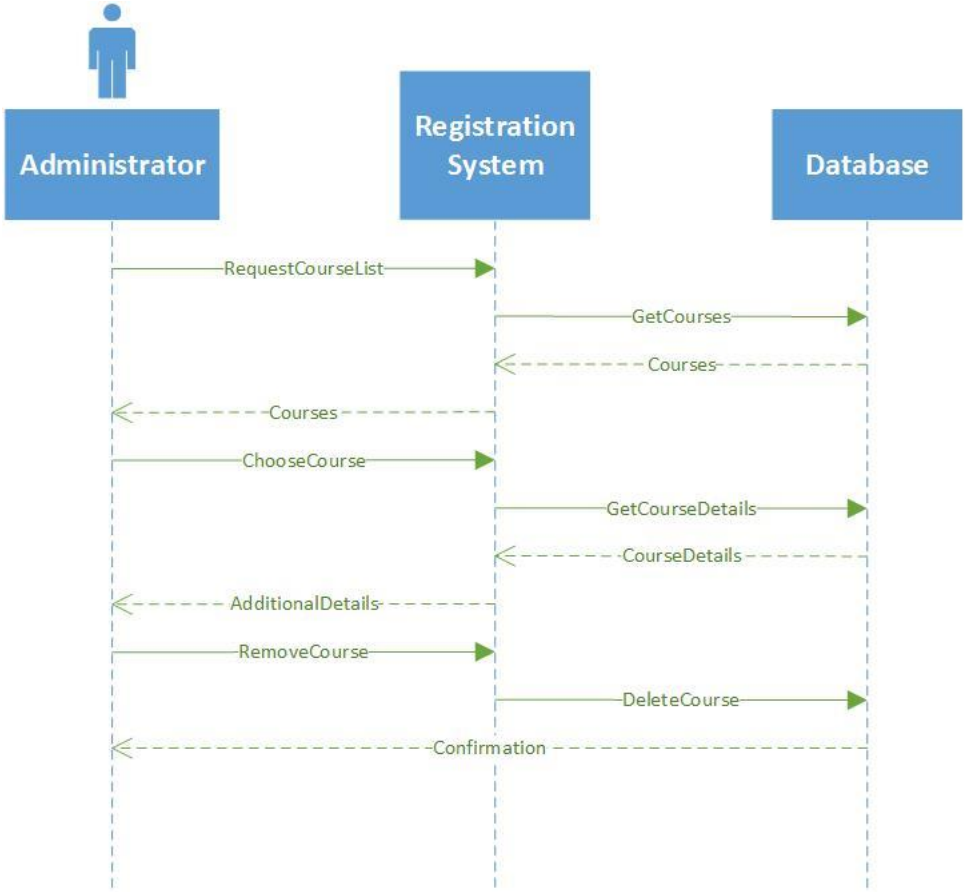
Normal “Remove Course” Scenario

1. Administrator request for a list of courses
2. System displays all available courses in that term
3. Administrator chooses a course to get additional information about
4. System displays additional information about the course
5. Administrator decides to remove the course
6. System deletes course
7. System displays completed action

“Remove Course” Nonempty Scenario

1. Administrator request for a list of courses
2. System displays all available courses in that term
3. Administrator chooses a course to get additional information about
4. System displays additional information about the course
5. Administrator decides to remove the course
6. System notifies those involved and updates their information.
7. System deletes course
8. System displays completed action

Remove Course High Level SSD



Remove Course Use Case (Concrete)

Participants: Administrator, Registration System, Database

Pre-Condition: The User has logged in successfully as an administrator and has chosen remove course option.

Typical Course of Events

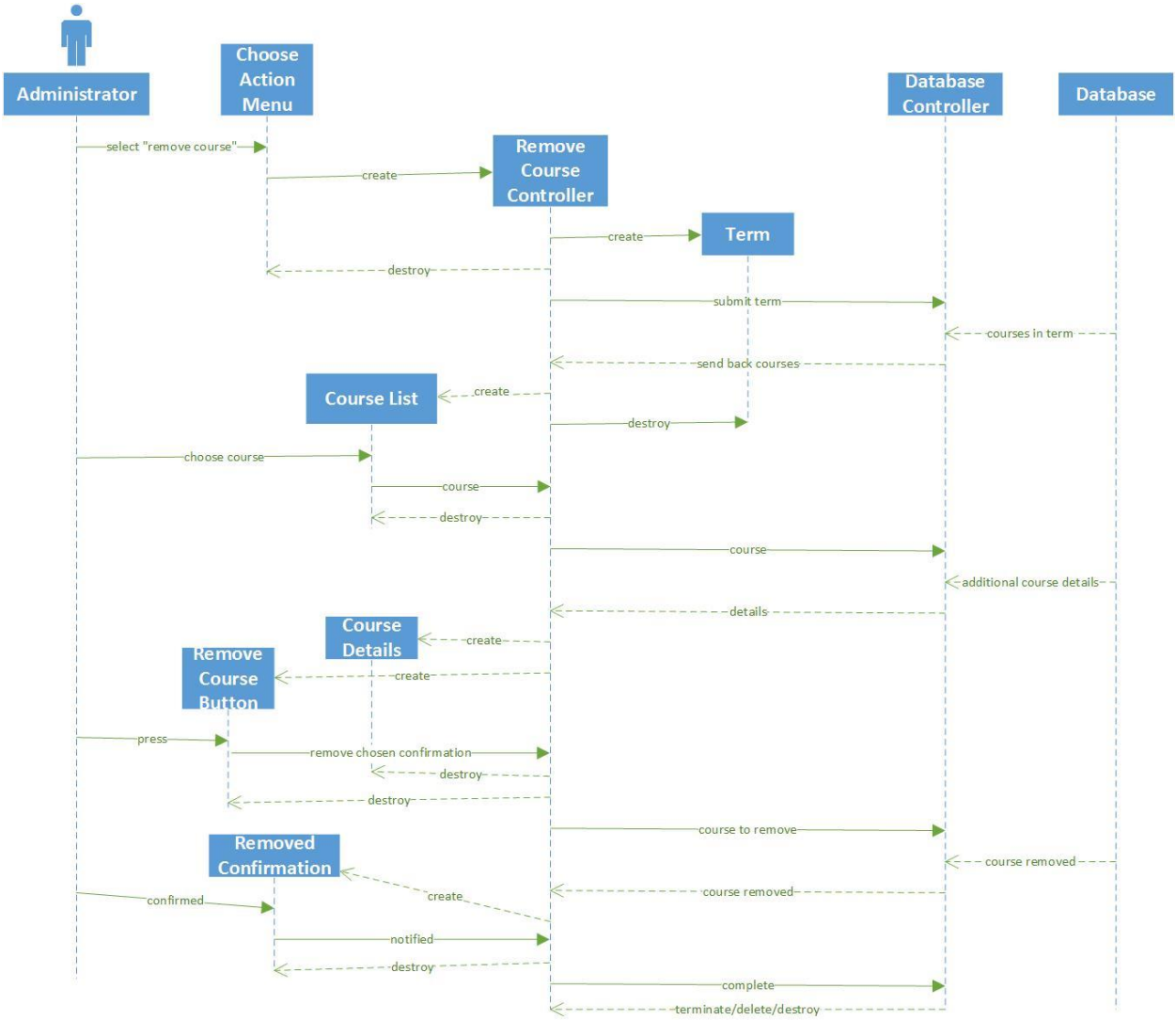
Actor Intentions	System Responsibility
1. Administrator request for a list of courses for a term through drop down menu	2. Get list of courses available during term from the database
	3. Present list of courses available during term that the administrator is looking at
4. Administrator decides on a possible course and requests additional information about that course by hitting a button	5. Send course to database.
	6. Get additional course information from database
	7. Generate course information page and display to the administrator
8. Administrator considers information about course and chooses to remove the course	9. Remove the course from the database

Alternative Courses

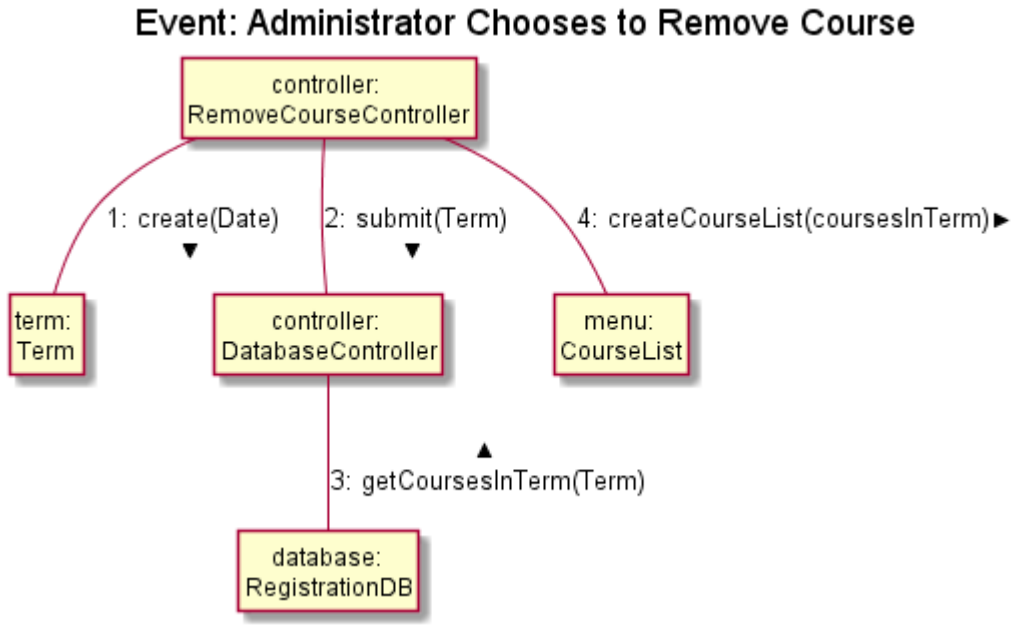
- Step 9: Students are registered for a course. Notify students who are signed up for the course. Drop students from course. Remove course.
- Step 9: Professor is assigned to course. Notify professor for that course. Remove that course from professor's class schedule. Remove course

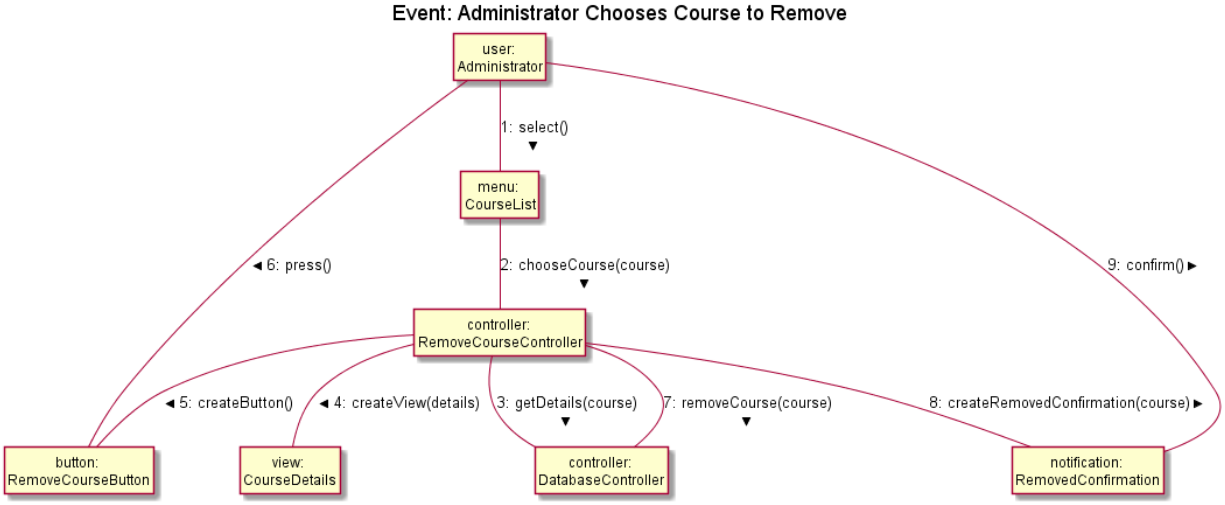
Post Condition: There is 1 fewer course in the system.

Remove Course Detailed SSD



Remove Course Collaboration Diagrams





Remove Course GRASP Patterns

The two biggest GRASP pattern used in these collaboration diagrams are once again creator and controller, as the administrator's remove course use case shares many similarities with the student's search classes use case. First, the remove course controller acts as a creator in the first diagram, since its primary goal in the first diagram is to create the course list and term. After creating the term object, it functions as controller by managing how the term object is passed as data. This helps keep the other objects high cohesive and reduce the coupling by reducing the communication between UI objects.

In the second diagram, the remove course controller creates many UI objects and views such as the course details view and the removed course confirmed notification, but the main goals of the controllers is to help reduce coupling through indirection. The remove course controller manages the operations and communication of boundary objects such as the remove course button and also manages the flow of information between these objects (such as course details) so that these non-controller objects can be highly cohesive and focus on one thing. Similarly, the database controller manages interactions with the data base to simplify the flow of information prevent direct communication with the database by other objects. This helps achieve the low coupling GRASP pattern for this use case, since most of the information is flowing through the controllers as opposed to objects directly communicating and relying on each other.

View Grades Use Case (Essential)

Participants: Student, Registration System, Database

Pre-Condition: The User has logged in successfully as a student

Typical Course of Events

Actor Intentions	System Responsibility
1. Student wants to view his/her grades	
2. Students request for the grades of all his/her courses	3. Retrieve the student's courses and corresponding grades
	4. Display semester report (classes + grades)
5. Student reviews grades	

Alternative Course

- Step 4: Students has no grades. Display error that tells student that no grades have been assigned yet and to check back at the end of the semester.

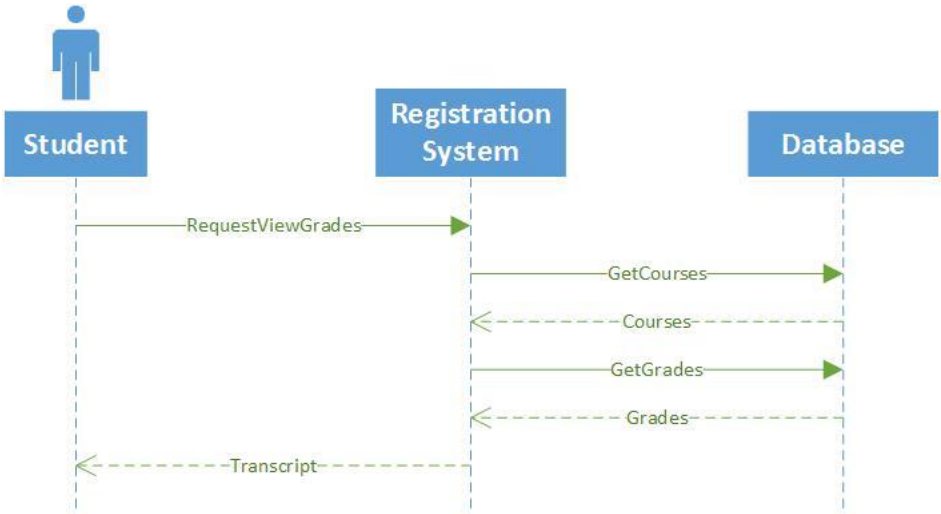
Normal "View Grades" Scenario

1. Student request to view grades
2. System retrieves all classes and corresponding grades
3. System displays the student's transcript

"View Grades" No Grades Scenario

1. Student request to view grades
2. System retrieves all classes and corresponding grades
3. System finds no final grades.
4. System displays error about no grades assigned

View Grades High Level SSD



View Grades Use Case (Concrete)

Participants: Student, Registration System, Database

Pre-Condition: The User has logged in successfully as a student

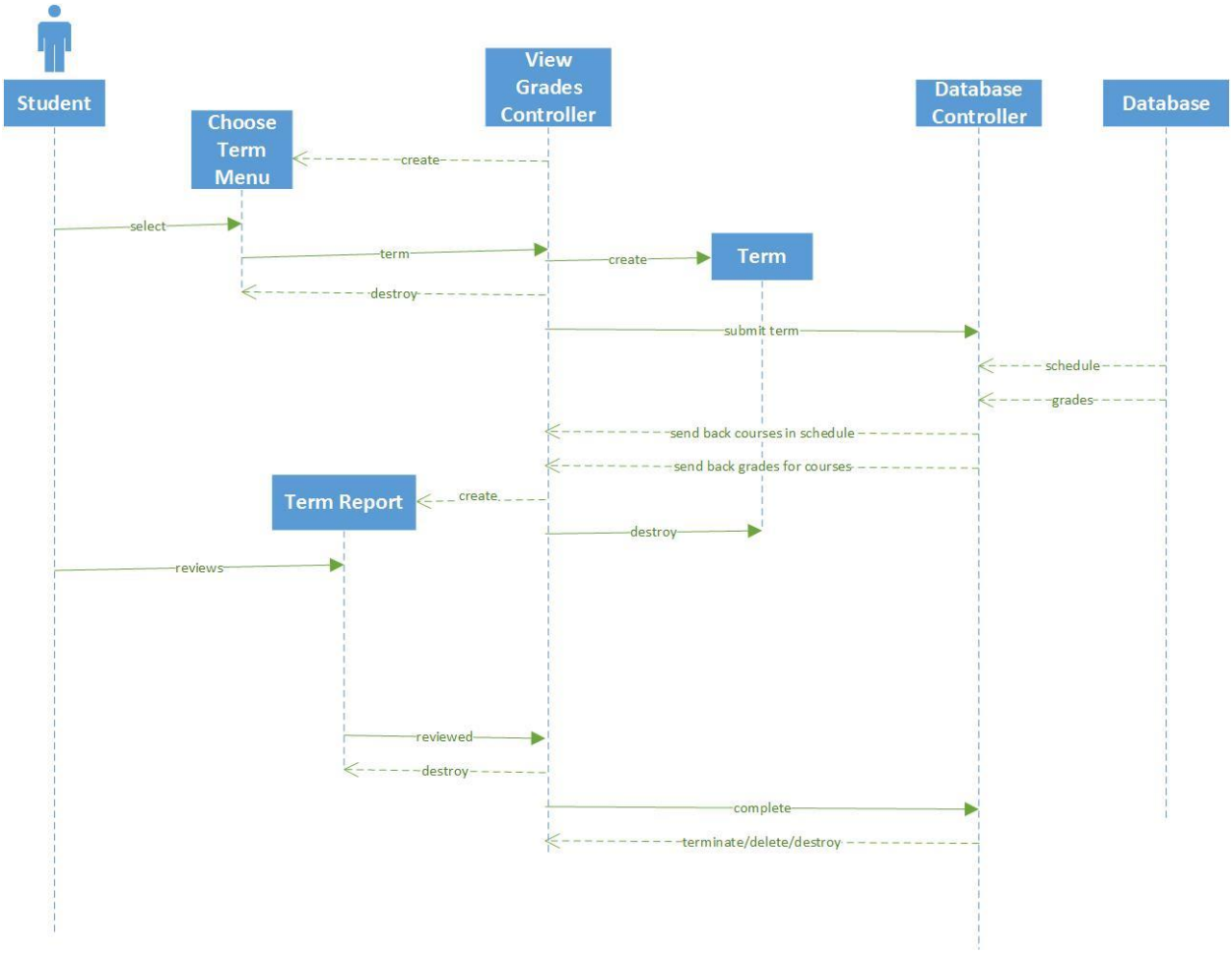
Typical Course of Events

Actor Intentions	System Responsibility
1. Student wants to view his/her grades	
2. Students request for the grades of his recent term	3. Retrieve students schedule from database
	4. Retrieve grades for each class from database
	5. Generate a semester report (classes + grades)
	6. Display report to the student
7. Student reviews grades	

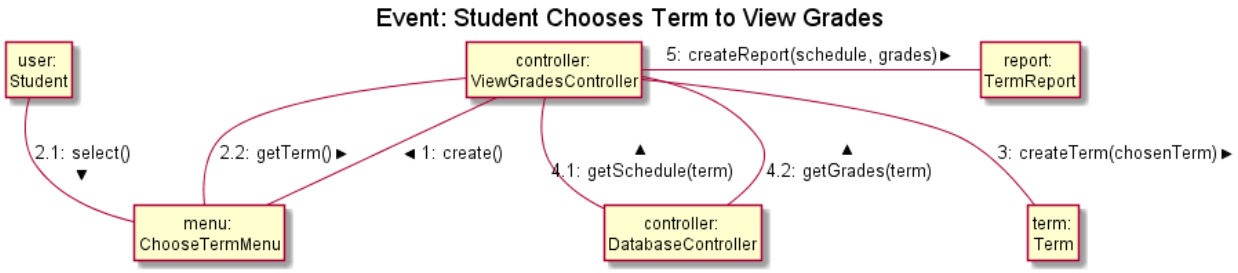
Alternative Course

- Step 4: Students has no grades. Display error that tells student that no grades have been assigned yet and to check back at the end of the semester.

View Grades Detailed SSD



View Grades Collaboration Diagram



View Grades GRASP Patterns

This use case uses controller objects to create UI elements and to reduce coupling through indirection. First, the view grades controller acts as a creator by creating the choose term menu. It then uses the student's result of interacting with the choose term menu to create a term object. After creating the term object, it functions as controller by managing how the term object is passed as data. The term object is used to retrieve a schedule and grades from the database. These requests go through a database controller in order to avoid direct interaction with the database. After obtaining the results from the database controller, the view grades controller uses the schedule and grades to create a report object. This helps keep the other objects (the term, report, and menu objects) high cohesive and reduce the coupling by reducing the communication between these objects.

Drop Class Use Case (essential):

Actor:	**System:**
1. Student decides to drop class.	2. Schedule is checked to ensure class can be dropped. 3. If class, can be dropped, system displays confirmation message for student to verify dropping class.
4. Student verifies class is to be dropped.	5. Newly adjusted schedule is sent to database for storage.

Alternative Courses:

1. Student is unable to log in to system because of invalid credentials.
2. Student is unable to drop class due to passing of deadline for ability to drop classes.

Drop Class Use Case (concrete):

Actor (student):	**System:**
Student attempts to log in to system	
	System verifies identity of actor
	System presents actor with appropriate menu
Student selects the option to “drop class”	
	System accesses “drop class” screen
	System displays list of available classes to drop
Student selects which class they wish to drop	
	System displays the confirmation screen for the actor to verify drop status
Student confirms drop status	
	System removes the class from the actor’s schedule
	System displays confirmation message for actor
Student confirms that action has been taken	
	System moves the newly adjusted schedule to the database to be updated
Student logs out of system	
	System sends information to database for final updating

Drop Class Scenarios

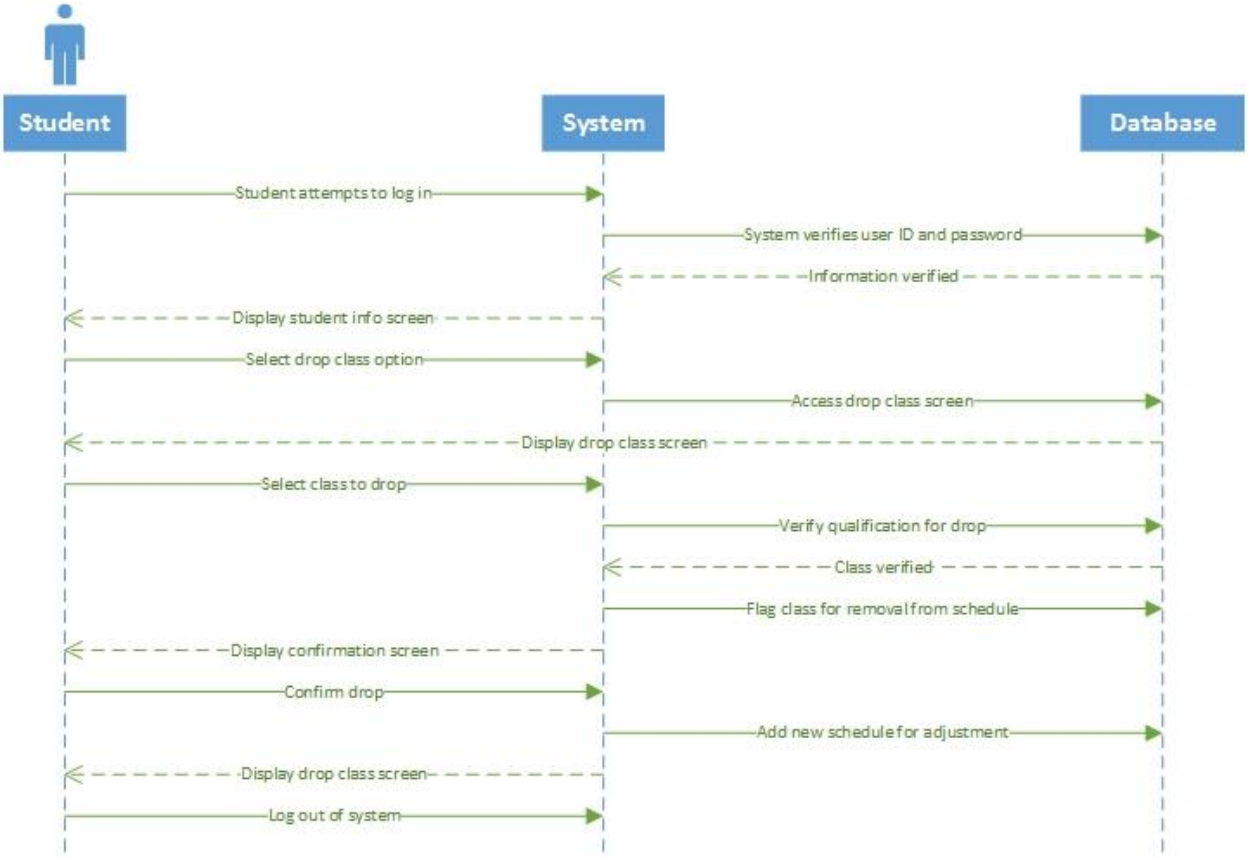
Successful log in and dropping of class:

Student attempts to log in to the system with proper user ID and password.
System verifies user ID and password.
System displays screen with student information.
Student selects the option to drop a class.
System accesses and displays the “drop class” screen with a list of classes available for dropping.
Student selects which class they wish to drop.
System verifies that class is available on schedule and qualifies for drop status.
System flags the class for removal.
System displays a confirmation screen for the actor to verify that the class is to be dropped.
Student verifies the action is to be taken.
System removes the class from the student’s schedule.
System displays a confirmation message for the actor.
Student confirms that the action has been taken.
System moves the newly adjusted schedule to the database to be updated by administrator.
System returns to the “drop class” screen.
Student logs out of system.

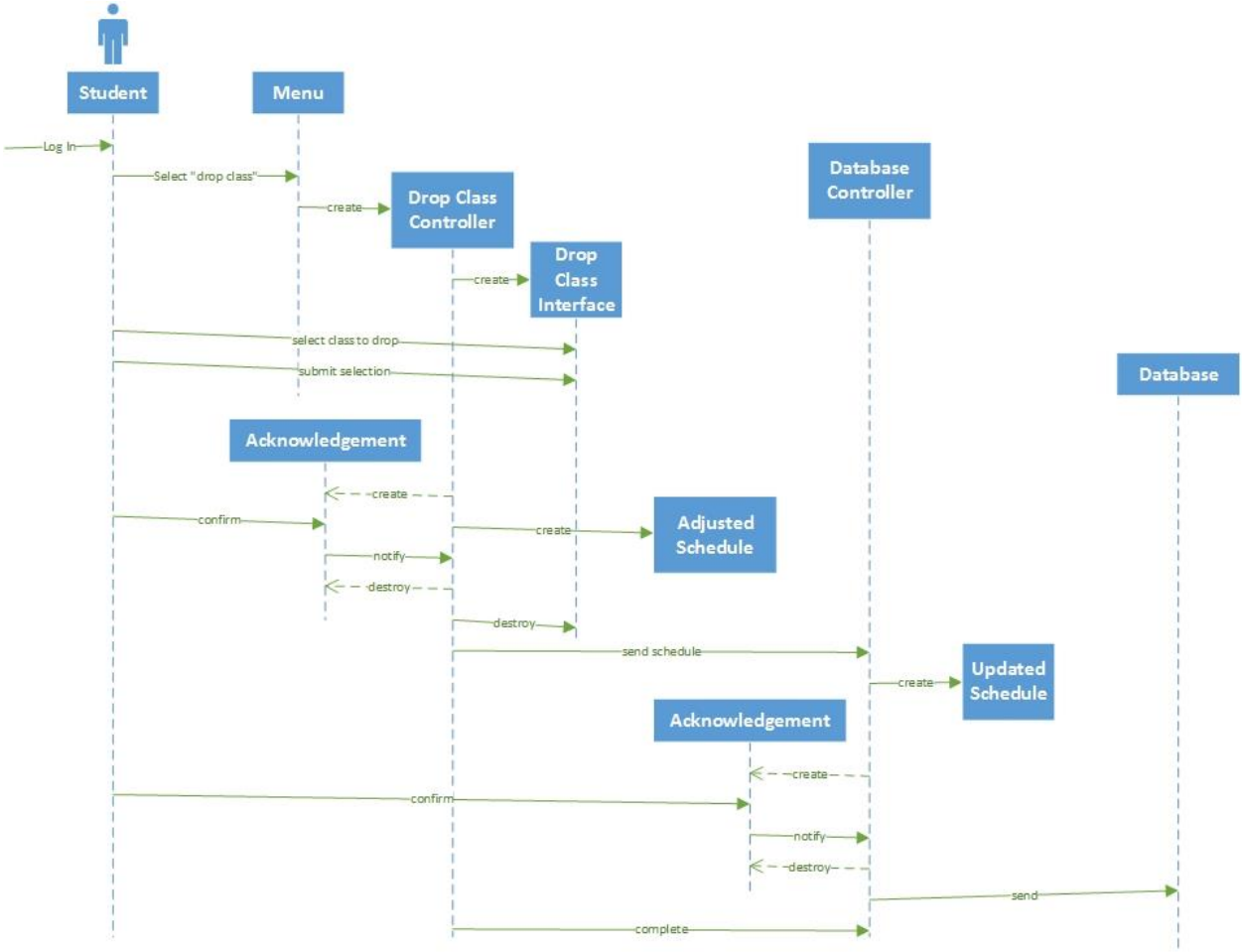
Successful log in with inability to drop class:

Student attempts to log in to the system with proper user ID and password.
System verifies user ID and password.
System displays screen with student information.
Student selects the option to drop a class.
System accesses and displays the “drop class” screen with a list of classes available for dropping.
Student selects which class they wish to drop.
System verifies that the class is available on schedule and qualifies for drop status.
System notes that class either is not on schedule or does not qualify for drop status.
System displays an error message to actor informing them of the problem with the action.
If class is not on schedule, system returns to “drop class” screen and allows student to reenter class information.
If class is not eligible for drop, the system returns to the “drop class” screen.
Student logs out of system.

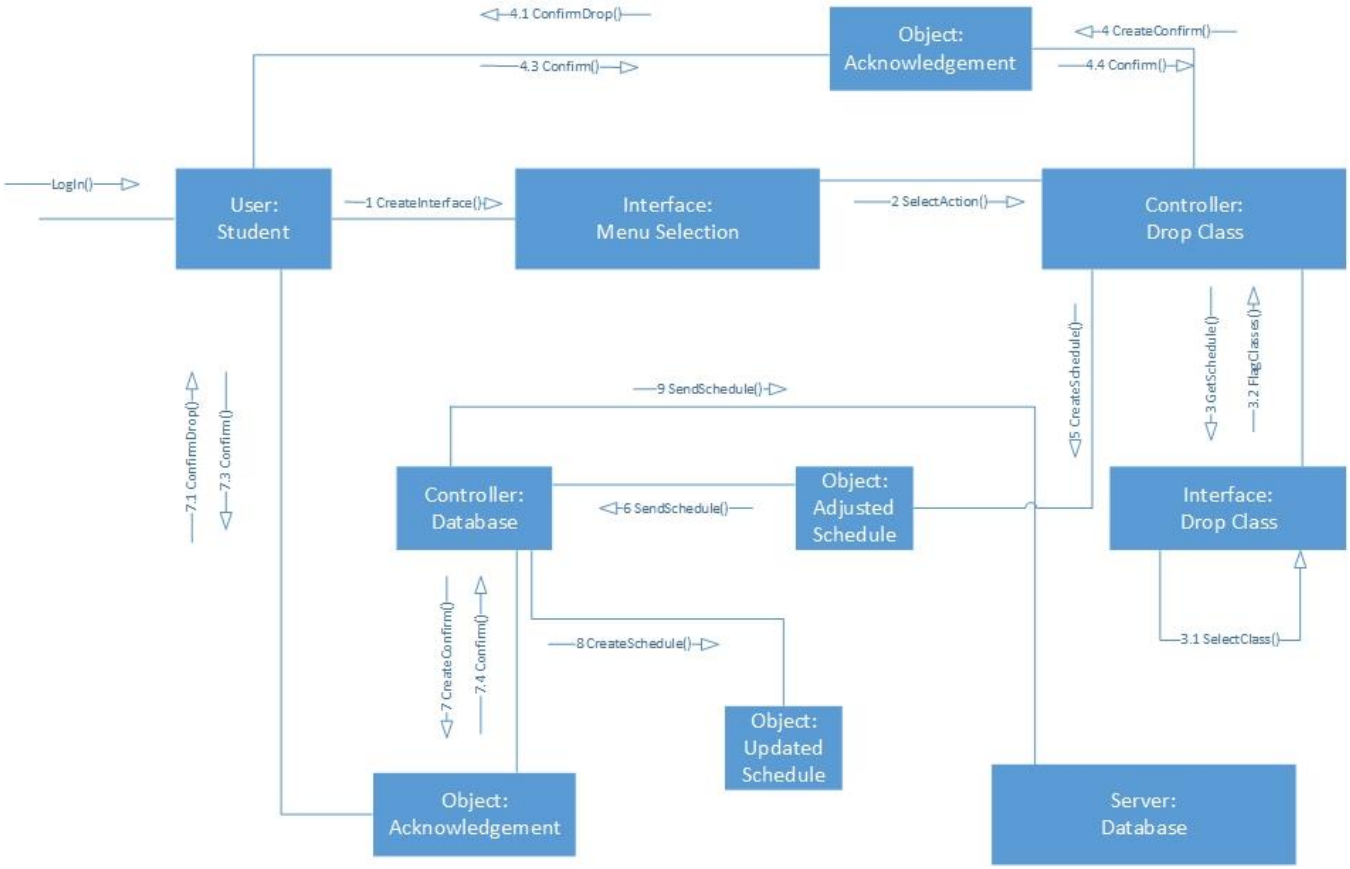
Drop Class High Level SSD



Drop Class Detailed SSD



Drop Class Collaboration Diagram



Drop Class Grasp Patterns

The GRASP patterns used in this collaboration diagram include creator, cohesion, coupling and controller. The button being pressed creates the controller for the system that will be used to drive the functionality of the program. The controllers used in this system include the Drop Class controller that facilitates the interaction with the system and the Database controller which controls all interaction with the database itself. These controllers are separate entities which guide the system to separate, highly cohesive objects and methods which helps maintain the high cohesion pattern of GRASP. Finally, since each object and method are doing their own work, the coupling of the system will remain low.

Update Registration Status Use Case (essential):

Actor:	**System:**
1. Administrator needs to change registration status of a student.	2. System pulls record of student to be changed.
3. Administrator changes information for student.	
4. Administrator logs out of system.	5. System sends information to database for storage.

Alternative Courses:

1. Registration status is unable to be changed because student has not been approved for registration by another entity.
2. Registration status is unchangeable because the student record is not found.

Update Registration Status Use Case (concrete):

Actor (Administrator):	**System:**
Administrator attempts to log in to system	
	System verifies log in
	System displays “adjustment” screen for administrator use
Administrator selects “registration” from list displayed by system	
	System accesses and displays registration screen
	System displays records available for adjustment
Administrator selects record to adjust	
	System accesses and displays entry associated with selected record
Administrator makes adjustments to record	
Administrator submits new record information to system	
	System accesses database to apply adjustments
	System displays confirmation screen once adjustments have been made to record
Administrator verifies adjustments	
Administrator logs out of system	
	System sends all information to database for final update.

Update Registration Status Scenarios

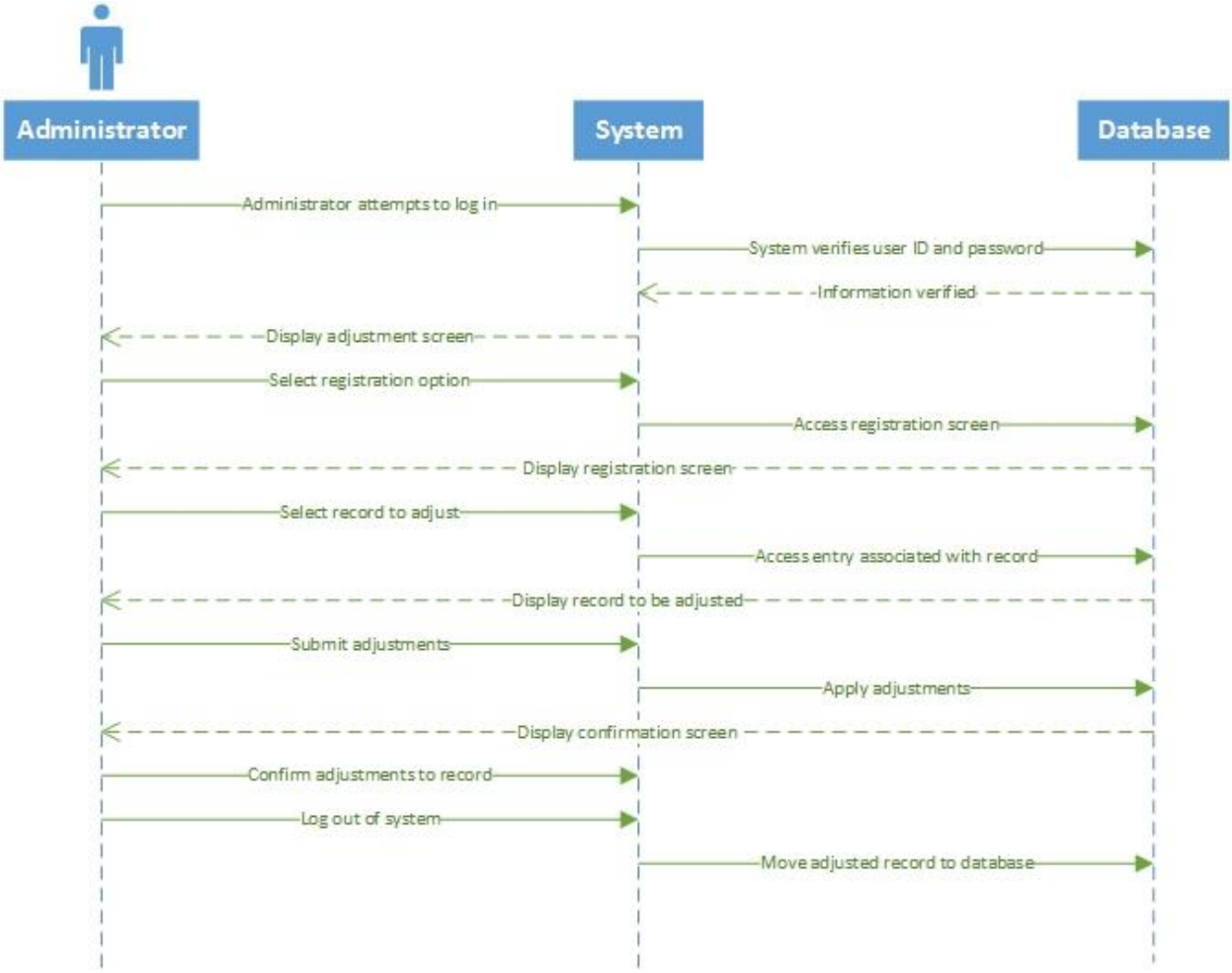
Successful log in and update of record:

Administrator attempts to log in to system with user ID and password.
System verifies user ID and password.
System displays “adjustment” screen for administrator use.
Administrator selects “registration” from list displayed by system.
System accesses and displays registration screen.
System displays records available for adjustment.
Administrator selects record to be adjusted.
System accesses entry associated with selected record.
System displays record and all adjustments needed for that record.
Administrator makes adjustments to record.
Administrator submits new record information to system.
System accesses database to apply adjustments.
System displays confirmation screen once adjustments have been made to record.
Administrator verifies that adjustments are correct.
Administrator logs out of the system.
System sends all information to database for final update.

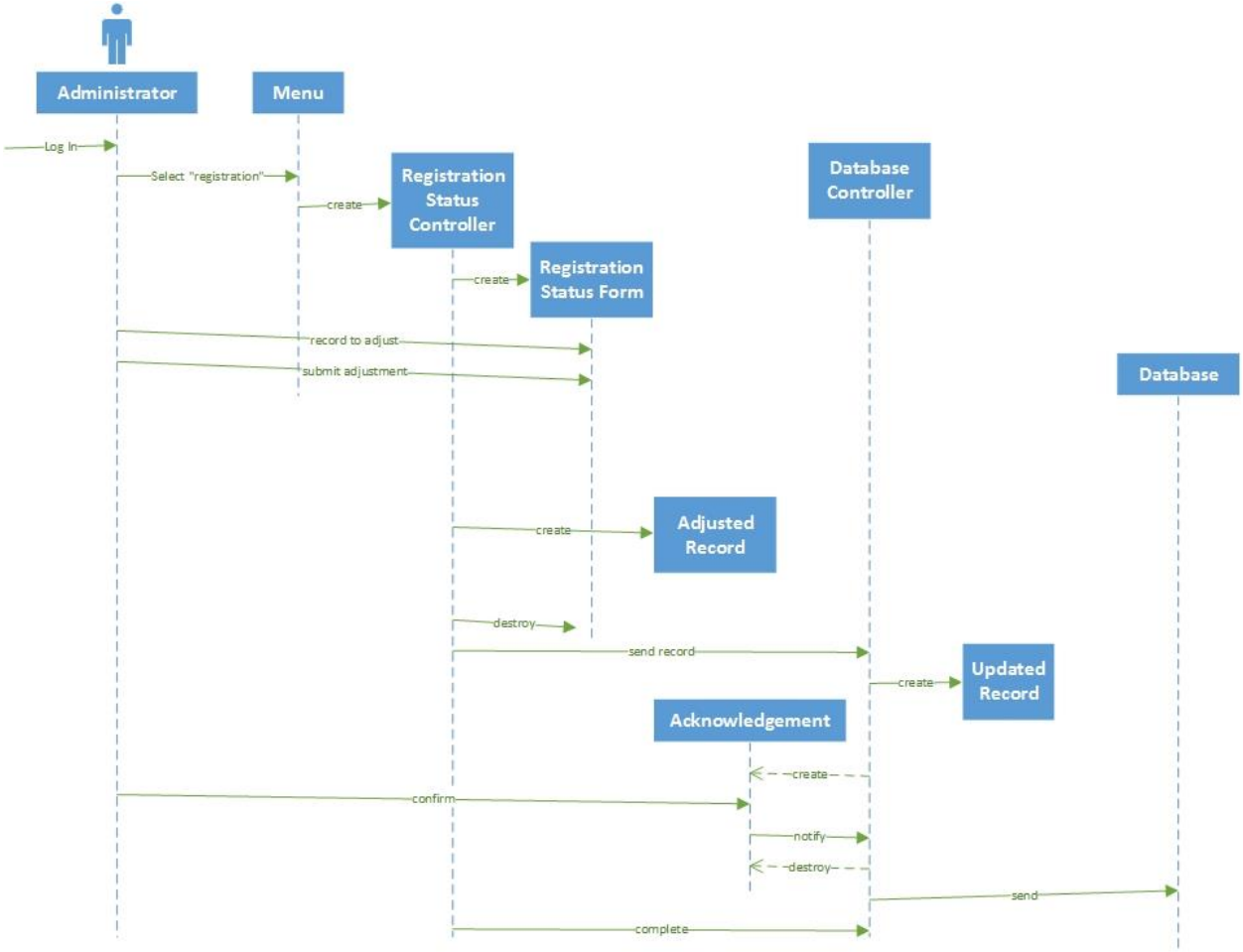
Unsuccessful log in:

Administrator attempts to log in with user ID and password.
System verifies user ID and password.
System notices that information is incorrect.
System displays a message stating that the user ID and/or password are incorrect.
System returns the user to the login screen.
Administrator attempts secondary login.

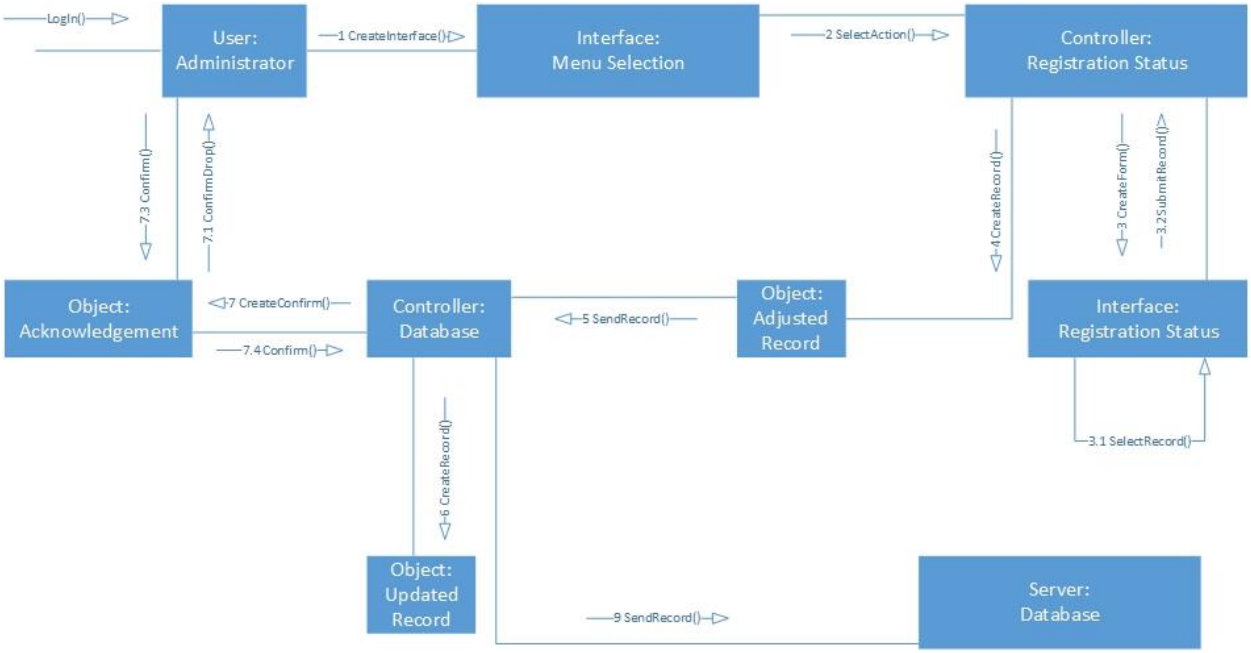
Update Registration Status High Level SSD



Update Registration Status Detailed SSD



Update Registration Status Collaboration Diagram



Update Registration Status Grasp Patterns

The GRASP patterns used in this collaboration diagram include creator, cohesion, coupling and controller. The button being pressed creates the controller for the system that will be used to drive the functionality of the program. The controllers used in this system include the Registration Status controller that facilitates the interaction with the system and the Database controller which controls all interaction with the database itself. These controllers are separate entities which guide the system to separate, highly cohesive objects and methods which helps maintain the high cohesion pattern of GRASP. Finally, since each object and method are doing their own work, the coupling of the system will remain low.

Remove Users Use Case (essential):

Actor:	**System:**
1. Administrator decides to update user list.	2. User list is pulled for editing.
3. Administrator updates list, as needed.	
4. Administrator logs out of system.	5. System sends information to database for storage.

Alternative Courses:

1. System is unable to pull list for editing due to corrupted data on the database.
2. Administrator is unable to log into system due to invalid credentials.

Remove Users Use Case (concrete):

Actor (Administrator):	**System:**
Administrator attempts to log in to system	
	System verifies user ID and password
	System displays “adjustment” screen for administrator use
Administrator selects “student list” from list displayed by system	
	System accesses database to retrieve list of available students
	System displays student list with options for adjusting student status
Administrator changes status of all students not currently enrolled in classes to “remove”	
Administrator submits new list to system for adjustment	
	System changes status of all adjusted students
	System displays a confirmation screen with list of all students to be adjusted
Administrator verifies list is correct and confirms list as displayed	
	System returns user to the “adjustment” screen
Administrator logs out of system	
	System sends information to database for final updating

Remove Users Scenarios

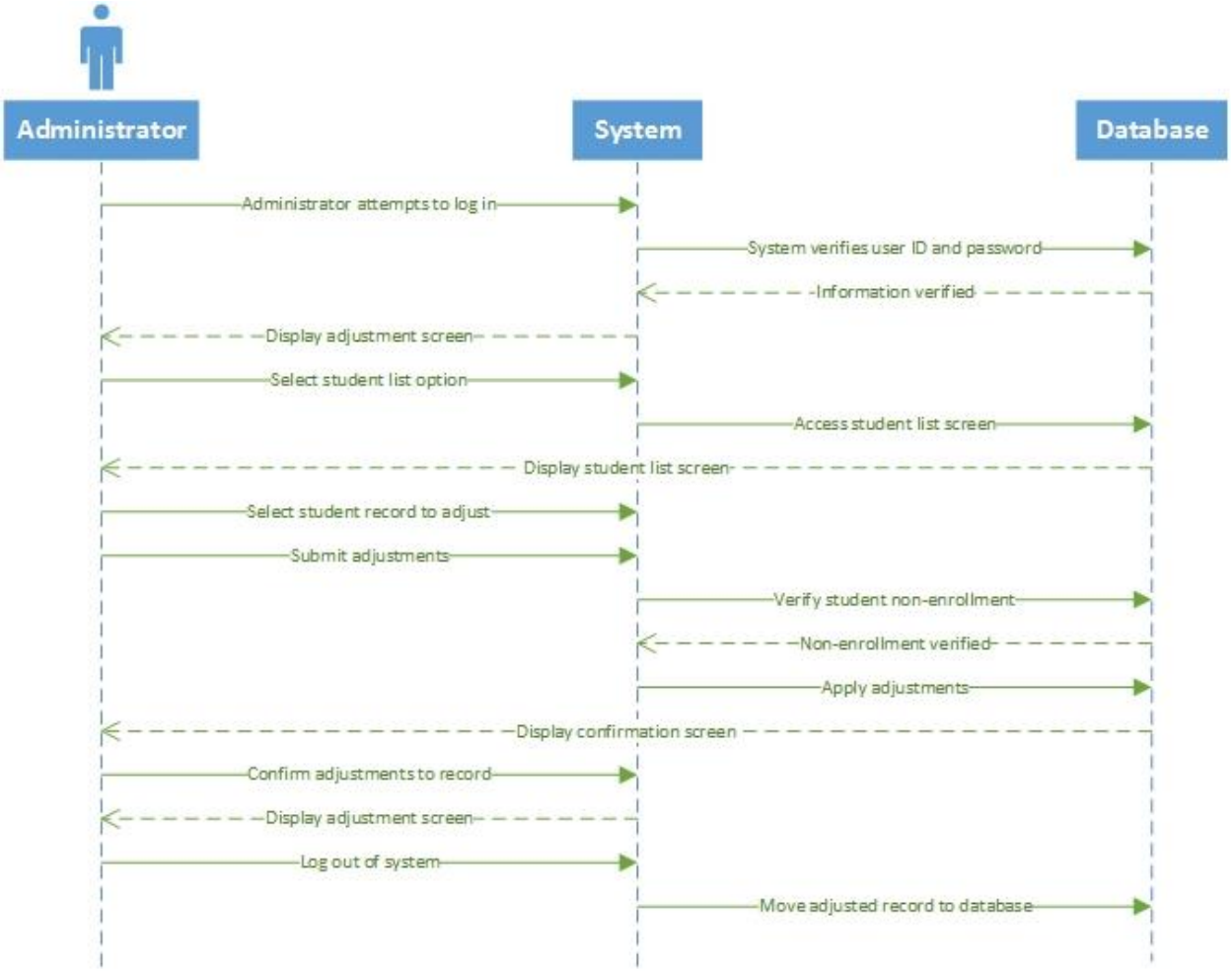
Successful log in and removal of students:

Administrator attempts to log in to system with user ID and password.
System verifies user ID and password.
System displays “adjustment” screen.
Administrator selects “student list” from list displayed by system.
System accesses database to retrieve active student list.
System displays student list with options for adjusting student status.
Administrator changes status for all students not currently enrolled in classes to “remove”.
Administrator submits new list to system for adjustment.
System verifies all students do not have active schedules.
System adjusts status of all adjusted students.
System displays a confirmation screen with list of all adjusted students.
Administrator verifies list is correct and confirms list displayed.
System returns user to “adjustment” screen.
Administrator logs out of system.
System sends information to database for final updating.

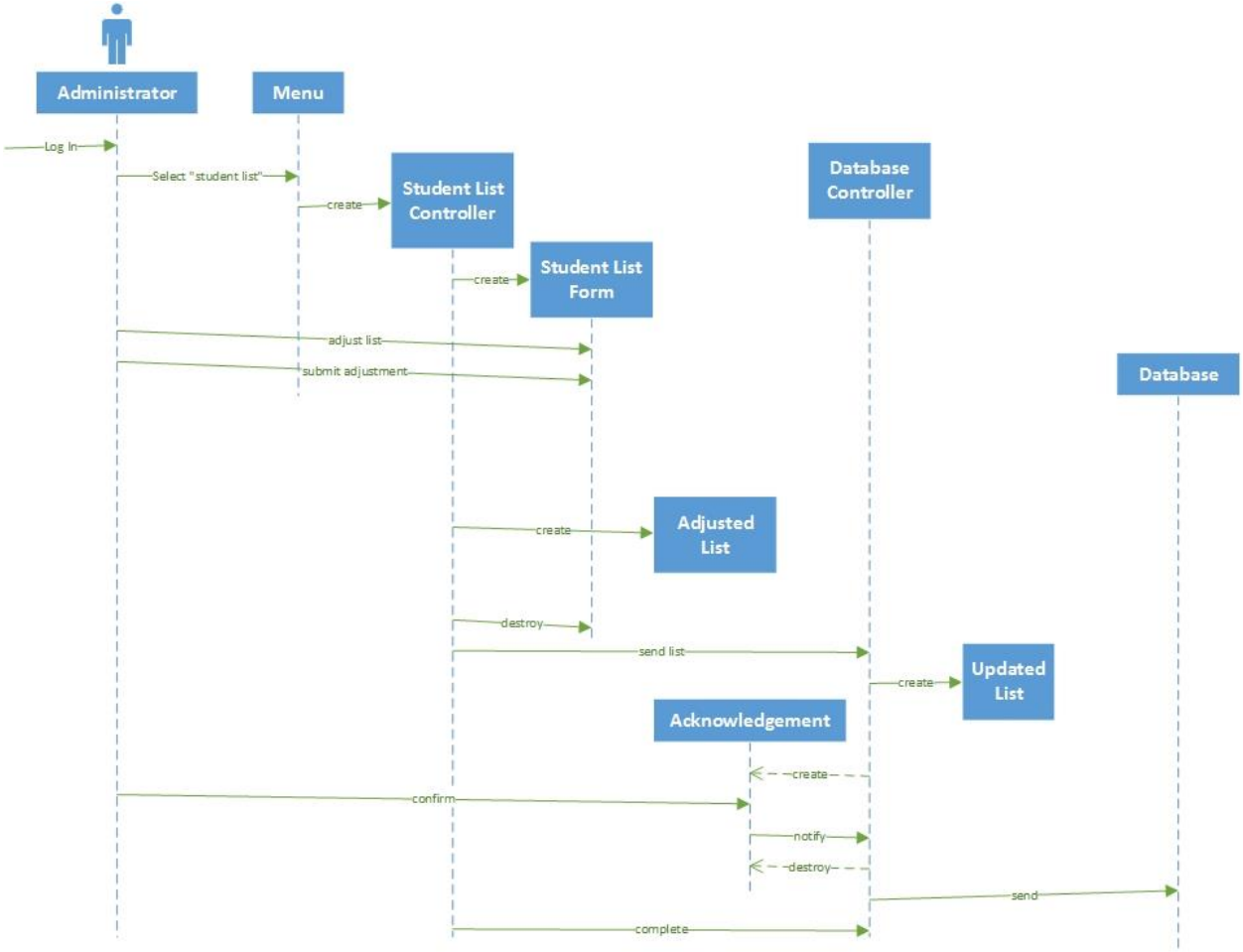
Successful log in with error in removal of student:

Administrator attempts to log in to system with user ID and password.
System verifies user ID and password.
System displays “adjustment” screen.
Administrator selects “student list” from list displayed by system.
System accesses database to retrieve active student list.
System displays student list with options for adjusting student status.
Administrator changes status for all students not currently enrolled in classes to “remove”.
Administrator submits new list to system for adjustment.
System verifies all students do not have active schedules.
Student file is found with current active schedule.
System displays error screen with list of all active students.
System returns user to “student list” screen.
Administrator logs out of system.
System sends information to database for final updating.

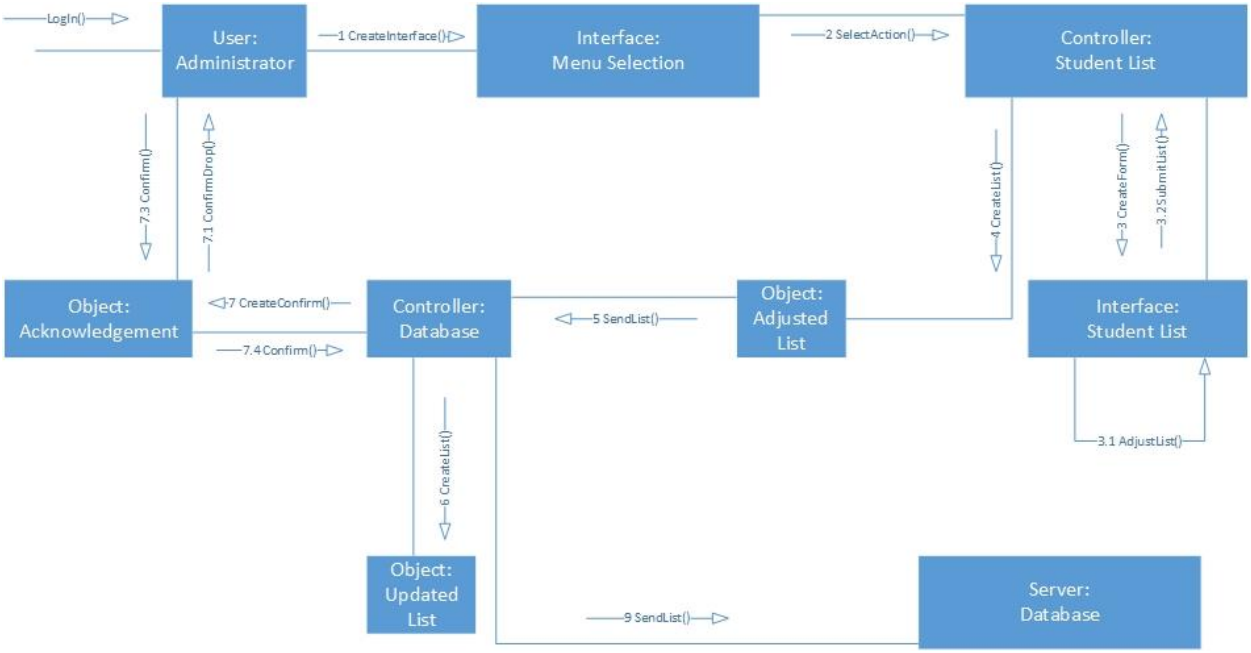
Remove user High Level SSD



Remove User Detailed SSD



Remove User Collaboration Diagram



Remove User Grasp Patterns

The GRASP patterns used in this collaboration diagram include creator, cohesion, coupling and controller. The button being pressed creates the controller for the system that will be used to drive the functionality of the program. The controllers used in this system include the Student List controller that facilitates the interaction with the system and the Database controller which controls all interaction with the database itself. These controllers are separate entities which guide the system to separate, highly cohesive objects and methods which helps maintain the high cohesion pattern of GRASP. Finally, since each object and method are doing their own work, the coupling of the system will remain low.

View Registration Status Use Case (Essential):

Actor:	**System:**
1. Student decides to check their registration status.	2. System accesses database to pull registration status for student. 3. System displays registration status for actor to view.
4. Student logs out of system.	

Alternative Courses:

1. Student is unable to log in due to invalid credentials.
2. Registration status unavailable because student is not yet enrolled.

View Registration Status Use Case (Concrete):

Actor (Student):	**System:**
Student attempts to log in to system	
	System verifies user ID and password
	System accesses and displays student information main screen
Student chooses “registration status” from the list of available options displayed by system	
	System displays “registration status” screen
Student chooses semester to be checked from the menu displayed	
	System accesses the database to retrieve the information for that user associated with that semester
	System displays the status of registration
Student logs out of the system	

View Registration Status Scenarios

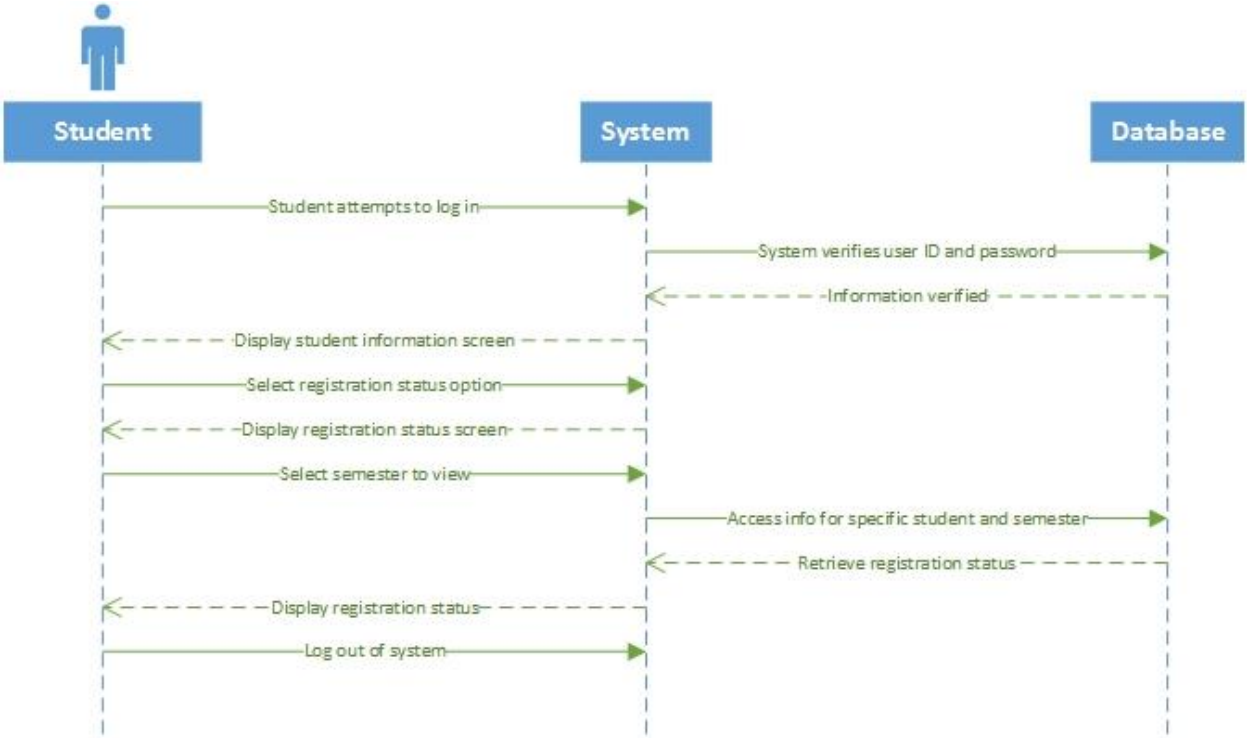
Successful log in with registration status “Active”:

Student attempts to log in to system with user ID and password.
System verifies user ID and password.
System displays student information main screen.
Student selects “registration status” from menu displayed on screen.
System displays registration status screen for student.
Student chooses desired semester from list provided.
System accesses database for specific user and specific semester.
System retrieves registration status and displays information for user.
Student logs out of system.

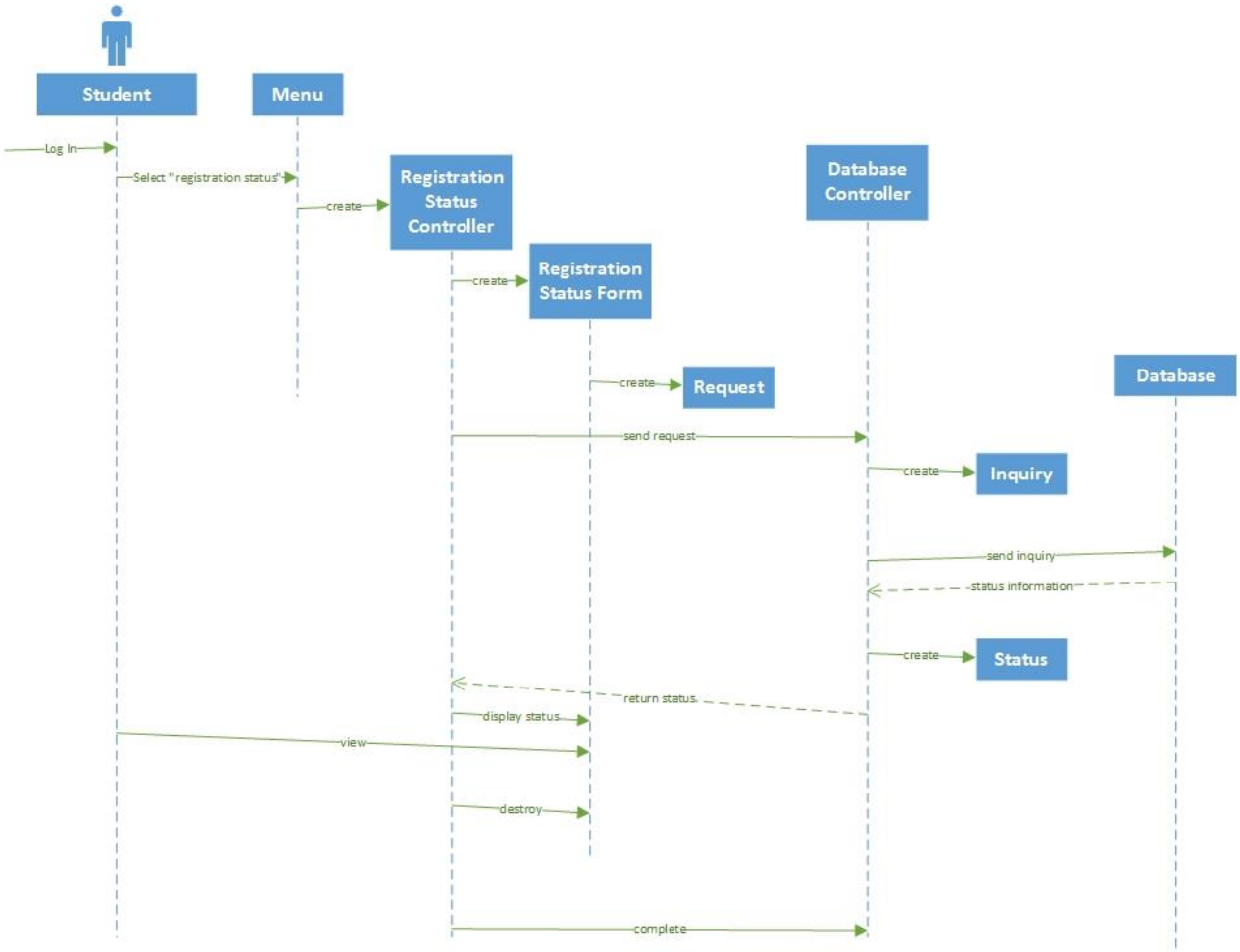
Successful log in with registration status “Not Active”:

Student attempts to log in to system with user ID and password.
System verifies user ID and password.
System displays student information main screen.
Student selects “registration status” from menu displayed on screen.
System displays registration status screen for student.
Student chooses desired semester from list provided.
System accesses database for specific user and specific semester.
System retrieves registration status.
System displays notification to student that advisor must be seen to change registration status.
Student logs out of system.

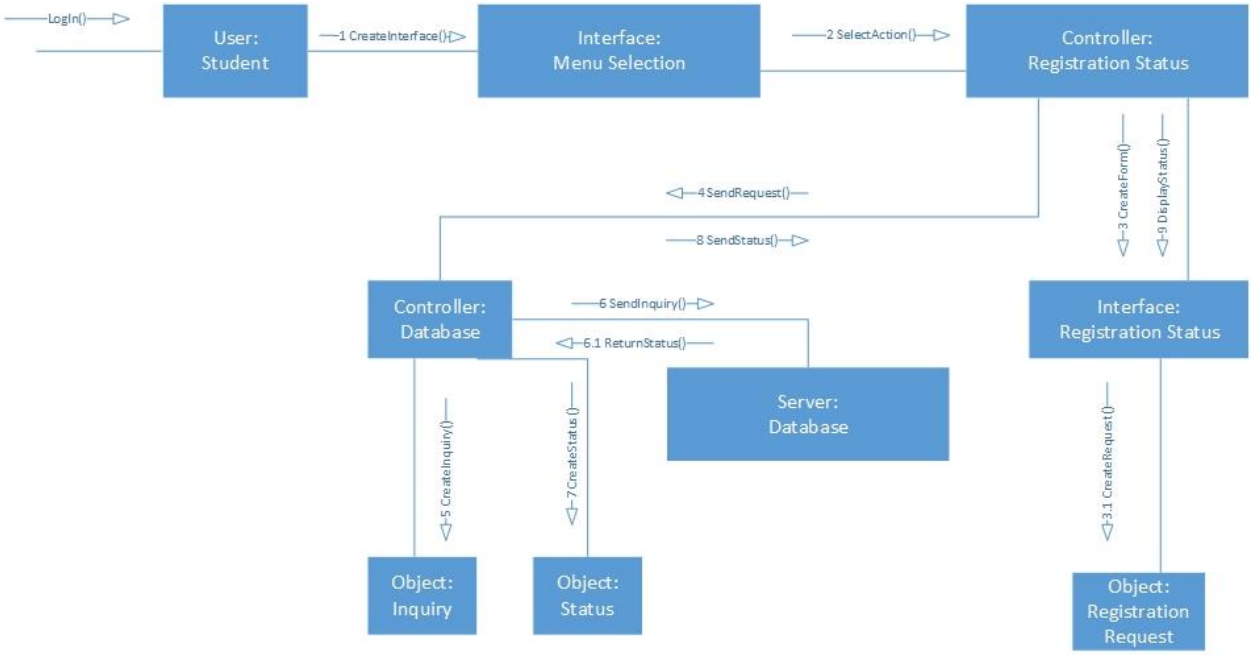
View Registration Status High Level SSD



View Registration Status Detailed SSD



View Registration Status Collaboration Diagram



View Registration Status GRASP Patterns

The GRASP patterns used in this collaboration diagram include creator, cohesion, coupling and controller. The button being pressed creates the controller for the system that will be used to drive the functionality of the program. The controllers used in this system include the Registration Status controller that facilitates the interaction with the system and the Database controller which controls all interaction with the database itself. These controllers are separate entities which guide the system to separate, highly cohesive objects and methods which helps maintain the high cohesion pattern of GRASP. Finally, since each object and method are doing their own work, the coupling of the system will remain low.

Log In Use Case (essential)

Use Case: Log In

Participants: User (Student, Admin, or Instructor), Server

Precondition: System booted up and log in screen open.

Actor Intentions:	System Responsibility:
1. User requests connection to system.	2. Receives user request.
3. User submits username and password.	4. Checks for username in database.
	5. If username found, verify inputted password against given password.
	6. If passwords match, check user type.
7. User is displayed his/her homepage based on user type.	

Log In Scenarios

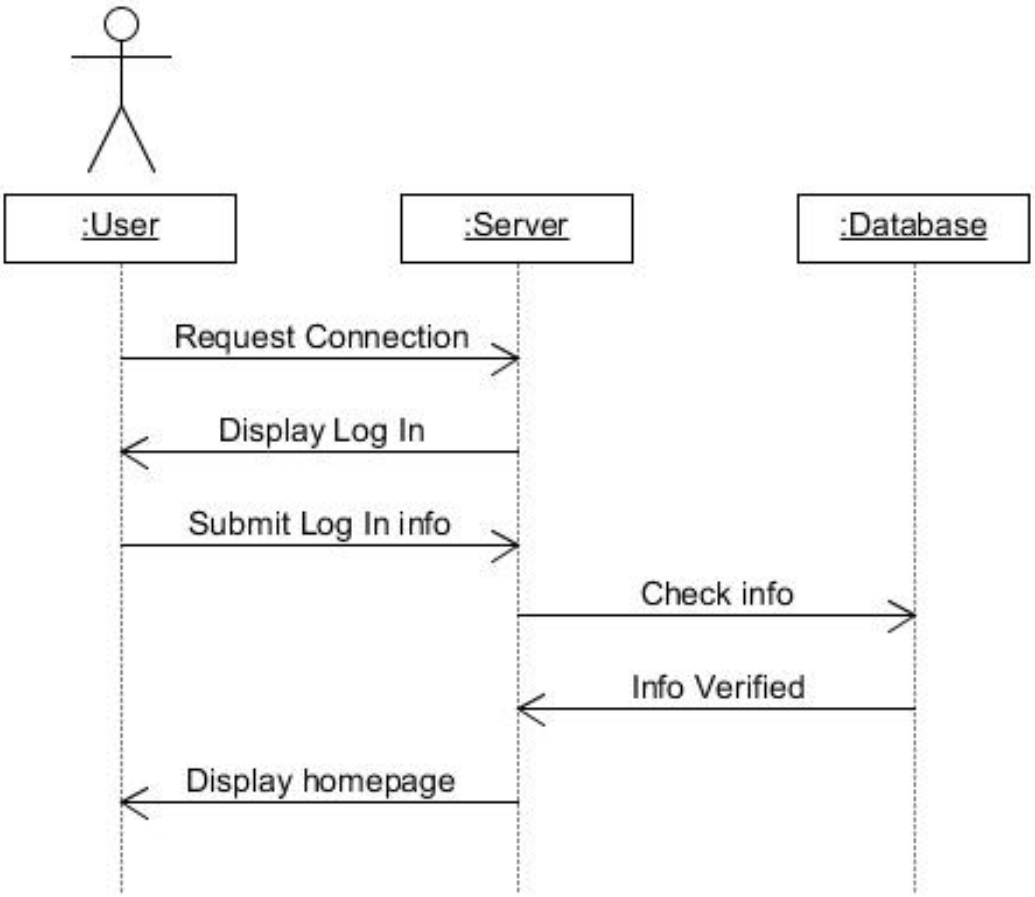
Successful Log in:

User types in username and password into given fields.
User clicks “Log In”
System receives log in request.
System checks for and finds username in database.
System checks inputted password against password in database.
System verifies user type.
System sends user’s homepage.
User is displayed his/her homepage.

Unsuccessful Log in (username not found):

User types in username and password into given fields.
User clicks “Log In”
System receives log in request.
System checks for username in database and does not find it.
System sends error message: “Username not found”
User is displayed error message and prompted to reenter log in information.

Login High Level SSD



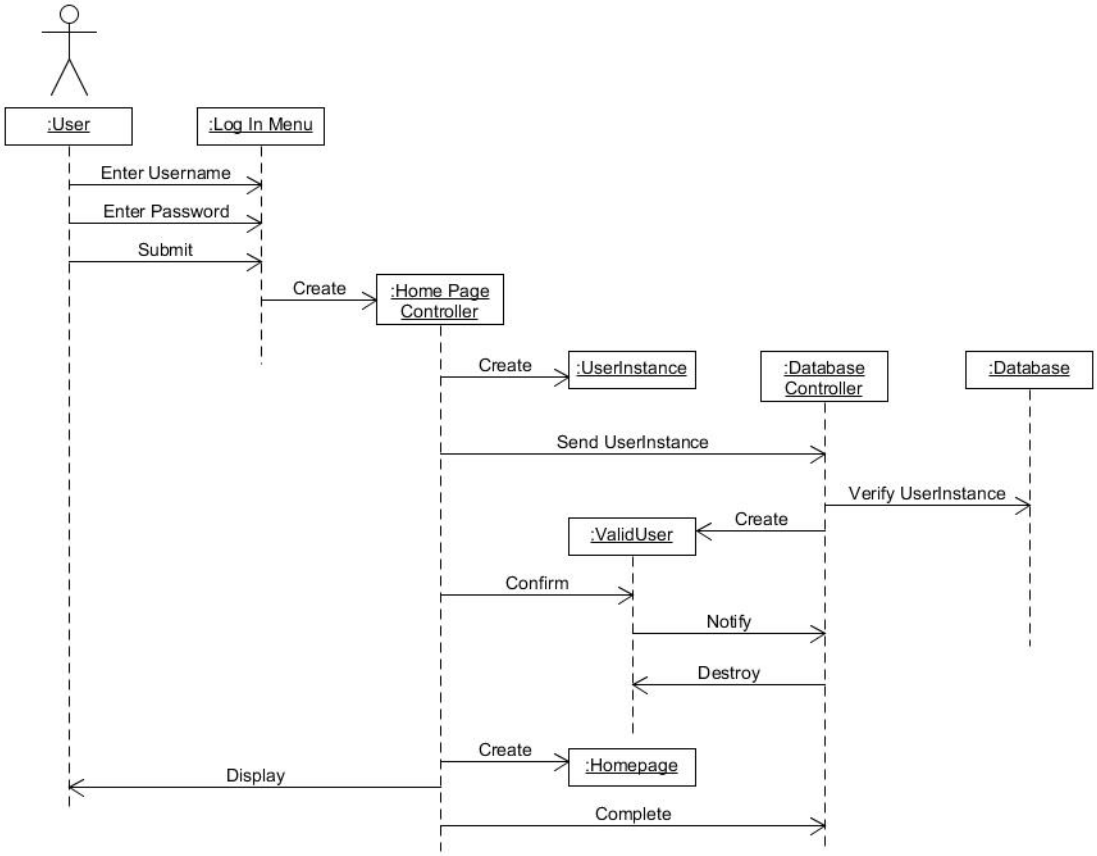
Log In Use Case (Concrete)

Participants: User (Student, Admin, Instructor), Server,

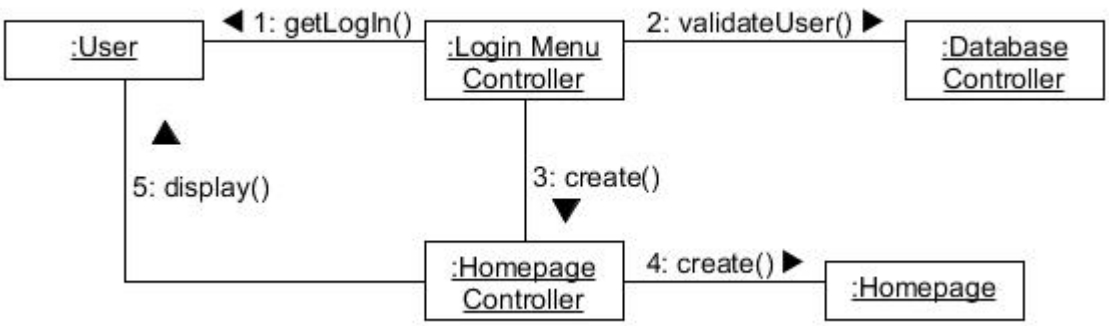
Precondition: System booted up and log in screen open.

Actor Intentions:	System Responsibility:
1. User types in username and password into given fields.	
2. User clicks “Submit”	3. System receives log in request.
	4. System checks username against database for a match.
	5. System finds username in database.
	6. System checks inputted password against password in database.
7. User is displayed his/her homepage.	

Log In Detailed SSD



Log In Collaboration Diagram



Log In GRASP Patterns

This collaboration diagram uses the GRASP patterns: creator, controller, cohesion, and coupling. After clicking the "Log In" button and validating the log in information a controller is created that will serve as the origin point for many further processes. The controllers used are: LoginMenu Controller which handles sending user data for validation, Database Controller to handle validation of user info, and a Homepage Controller which will display the homepage specific to the type of user. Lower coupling is achieved by separating functionality into separate loosely connected objects (e.g. controllers). Another effect of this separation of functionality is that each object has a very specific purpose, and to that end, has intimate knowledge of all self-contained functionality. This leads to higher cohesion.

View Schedule Use Case (Essential)

Participants: User (Student or Instructor), Server

Precondition: Logged into the system as a student or instructor.

Actor Intentions:	System Responsibility:
1. User clicks on view schedule button on homepage.	2. Receives schedule request.
	3. Checks database for schedule matched to current user.
	4. If schedule is matched, send schedule.
5. Schedule is displayed his/her schedule.	

View Schedule Scenarios

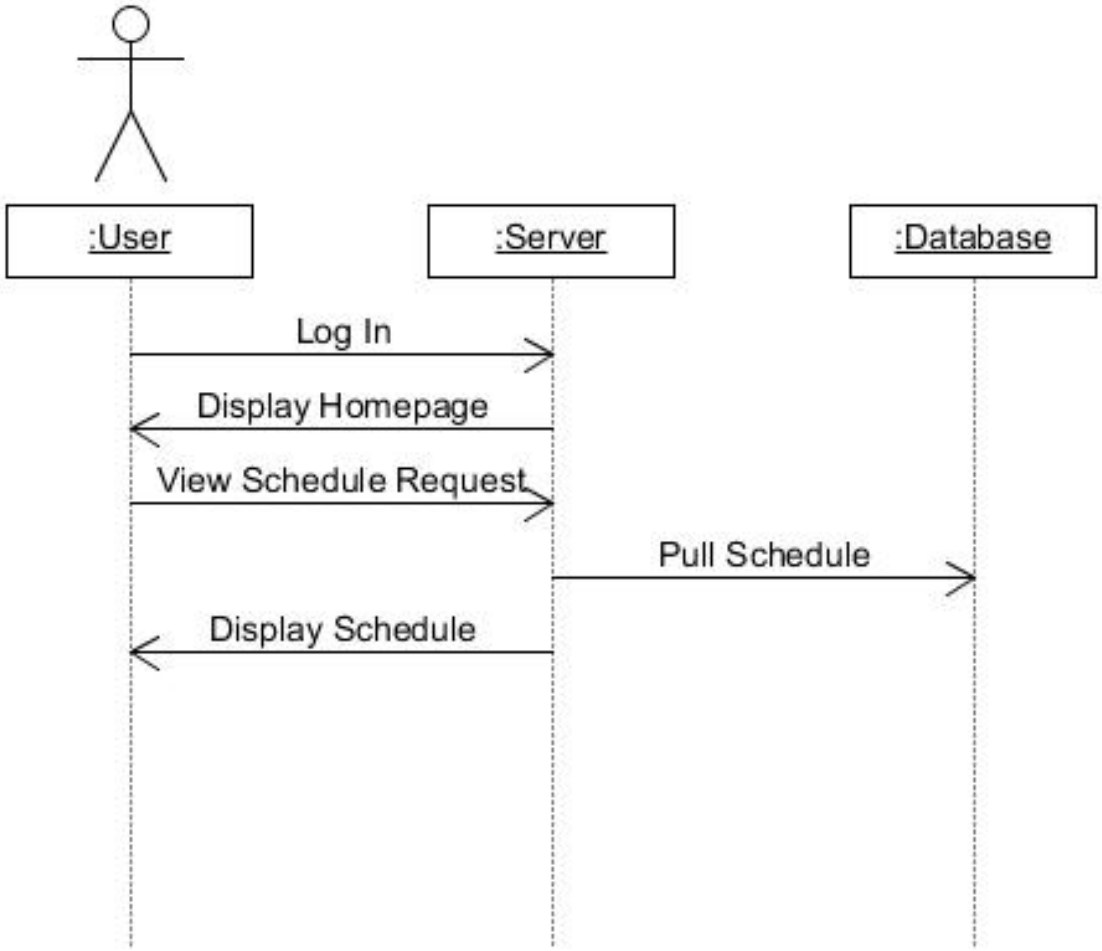
Schedule Successfully Displayed:

User successfully logs in.
User clicks on “View Schedule” button.
System receives request for schedule.
System searches database for schedule matching the user’s information.
System sends schedule.
User is displayed schedule page.

Dropped Connection:

User successfully logs in.
User clicks “View Schedule” button.
System fails to receive request.
Request time out.
User is displayed “Request Time Out” error message.

View Schedule High Level SSD



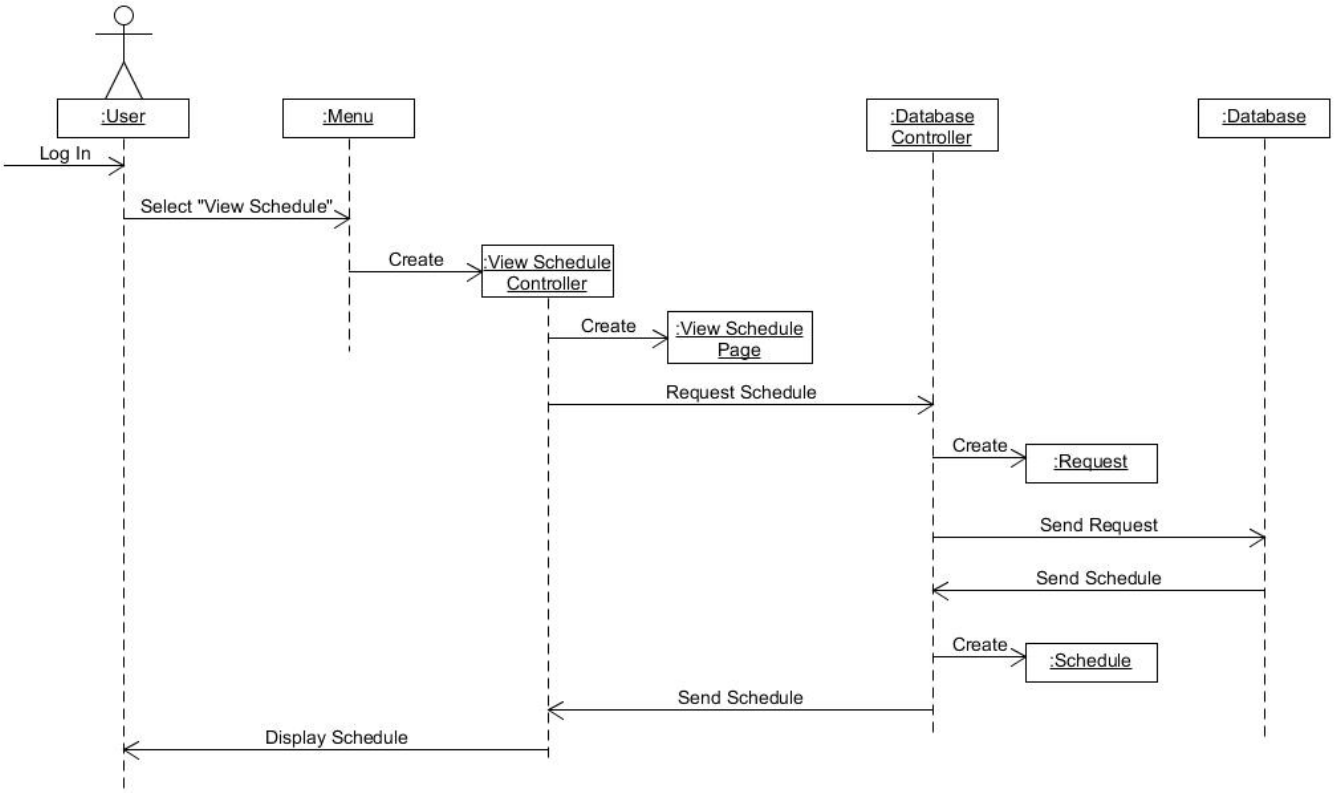
View Schedule Use Case (Concrete)

Participants: User (Student or Instructor), Server

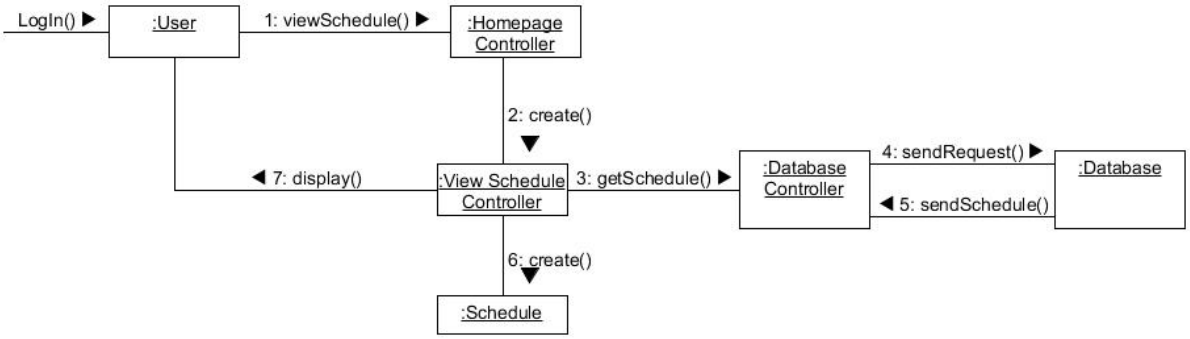
Precondition: User is logged in and viewing homepage.

User Intentions:	System Responsibility:
1. User clicks on “View Schedule” button.	2. System receives view schedule request.
	3. System checks database for schedule matching current user.
	4. System finds current user’s schedule
	5. System sends schedule.
6. User is displayed his/her schedule.	

View Schedule Detailed SSD



View Schedule Collaboration Diagram



View Schedule GRASP Patterns

This collaboration diagram uses the GRASP patterns: creator, controller, cohesion, and coupling. Once the user has logged in and clicked the "View Schedule" button, a controller specific to the view schedule process will be created. The controllers used are: Homepage which will create the view schedule controller, View schedule which will handle getting the schedule and creating the schedule page, and a database controller which will send requests to the database, receive data back from the database, and send it back to another controller. We achieve low coupling and high cohesion by splitting functionality into distinct objects (e.g. controllers) which have intimate knowledge of self-contained functionality.

Add Users Use Case (Essential)

Participants: User(Administrator), Server

Precondition: Logged in as administrator.

Actor Intentions:	System Responsibility:
1. Admin clicks “Add Users” button on homepage.	2. Server receives add user request.
	3. Sends add user page.
4. “Add User” page displayed to the user.	
5. Admin enters in new user information into the given fields and clicks submit.	6. New user information is added to the database.

Add Users Scenarios

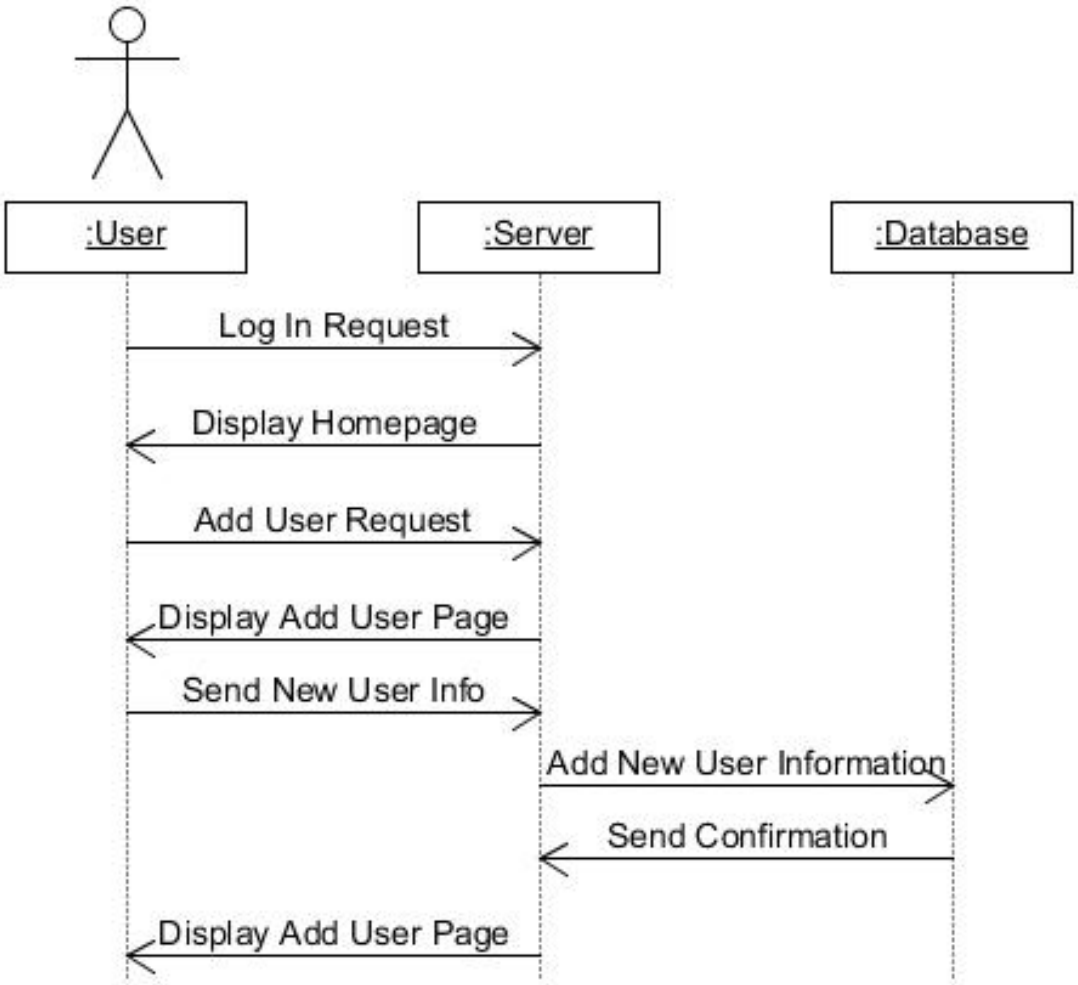
Successful Add User:

Admin logs into system.
System sends homepage.
Admin is displayed his/her homepage.
Admin clicks on “Add User” button.
System receives Add User request.
Admin is displayed Add User page.
Admin enters in new user information into given fields.
Admin clicks “Submit”
System adds new user to database.
System returns user to Add User page.

Duplicate User Information:

Admin logs into system.
System sends homepage.
Admin is displayed his/her homepage.
Admin clicks on “Add User” button.
System receives Add User request.
Admin is displayed Add User page.
Admin fills in new user information into given field.
System attempts to add new user information to database.
System finds duplicate information.
Admin is displayed error message: “Duplicate user information”

Add Users High Level SSD



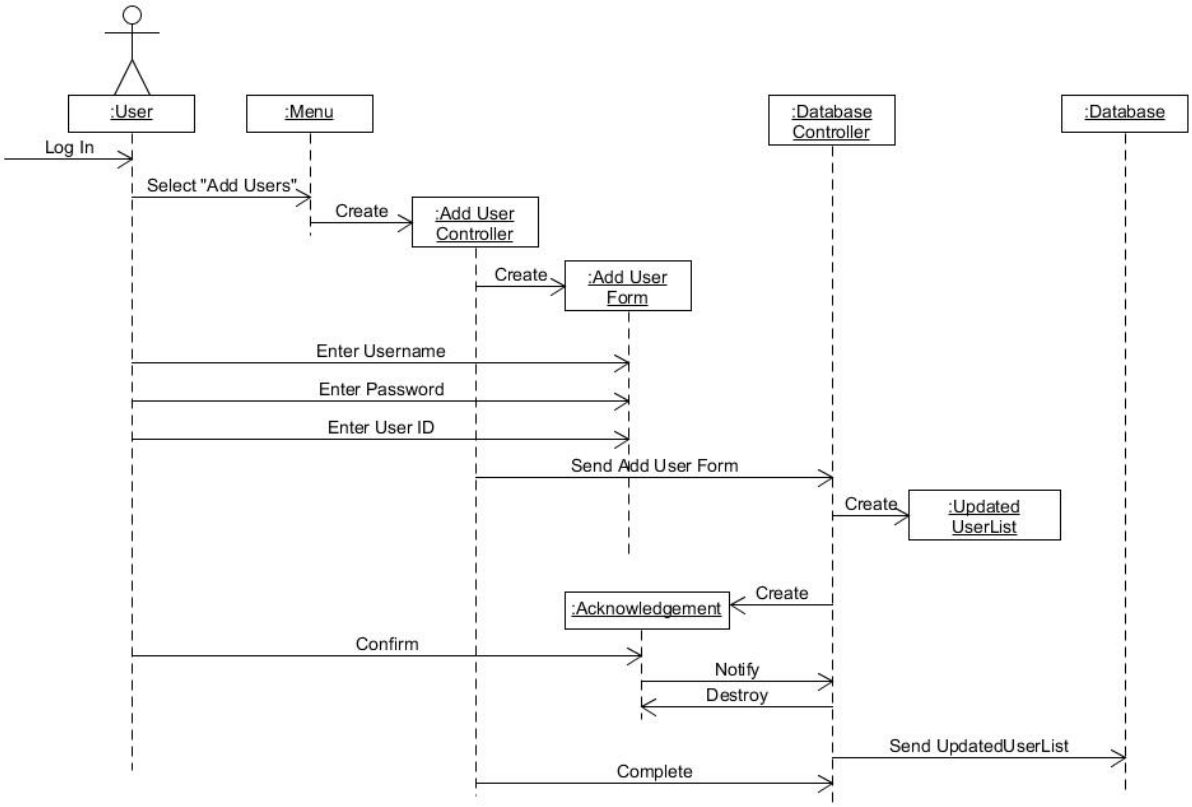
Add Users Use Case (Concrete)

Participants: User(Admin), Server

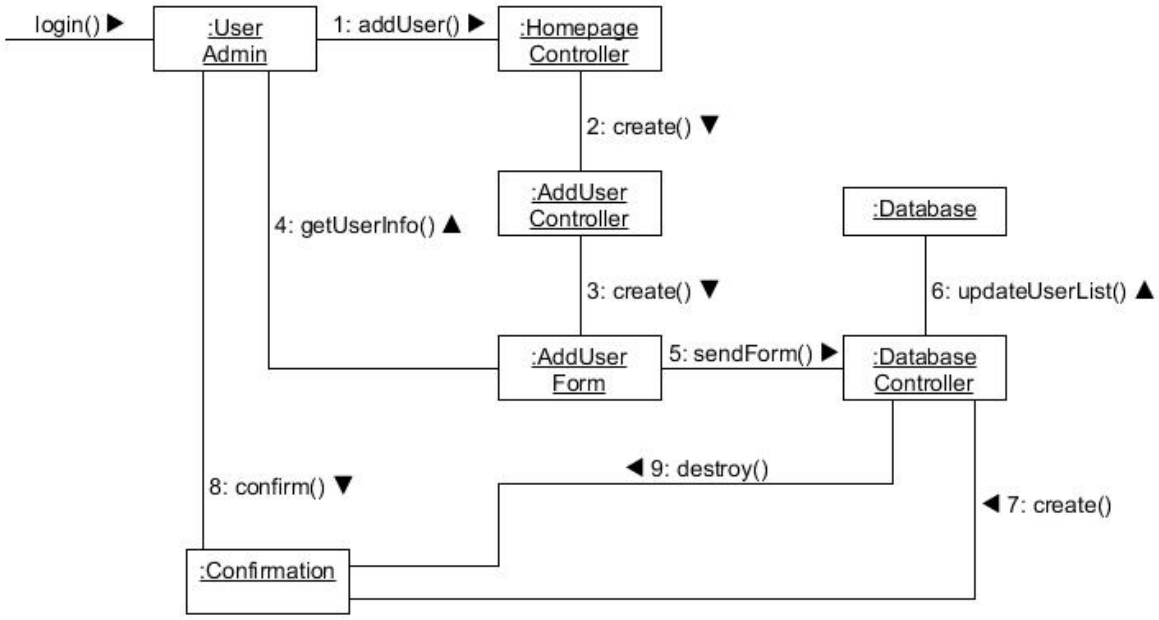
Precondition: Admin logged into system.

User Intentions:	System Responsibility:
1. User clicks “Add User”	2. System receives add user request.
	3. System sends add user page.
4. User enters new username.	
5. User enters new password.	
6. User enters new user ID.	
7. User clicks submit.	8. System checks username against database for duplicates.
	9. System checks new user ID against database for duplicates.
	10. System adds new user to database.
	11. System sends confirmation.
12. User is returned to blank add user page.	

Add Users Detailed SSD



Add Users Collaboration Diagram



Add Users GRASP Patterns

This collaboration diagram uses the GRASP patterns: creator, controller, cohesion, and coupling. The controllers used are as follows: Homepage controller which serves to initiate adding a new user by creating another controller, the add user controller which creates the add user form filled out by the user and then verifies the form with the database, the database controller which updates the data in the database. The separation of functionality into distinct separate objects give us low coupling, and since each object's functions work together we have high cohesion.

Log Out Use Case (essential)

Participants: User (Student, Professor, or Admin), Registration System

Precondition: User is already logged into the system.

Actor Intentions:	System Responsibility:
1. User clicks on log out button on homepage	2. Receives log out request
	3. Saves updated user info in database
	4. Logs out User
	5. Sends User to login screen
6. User is displayed the login screen	

Log Out Scenarios

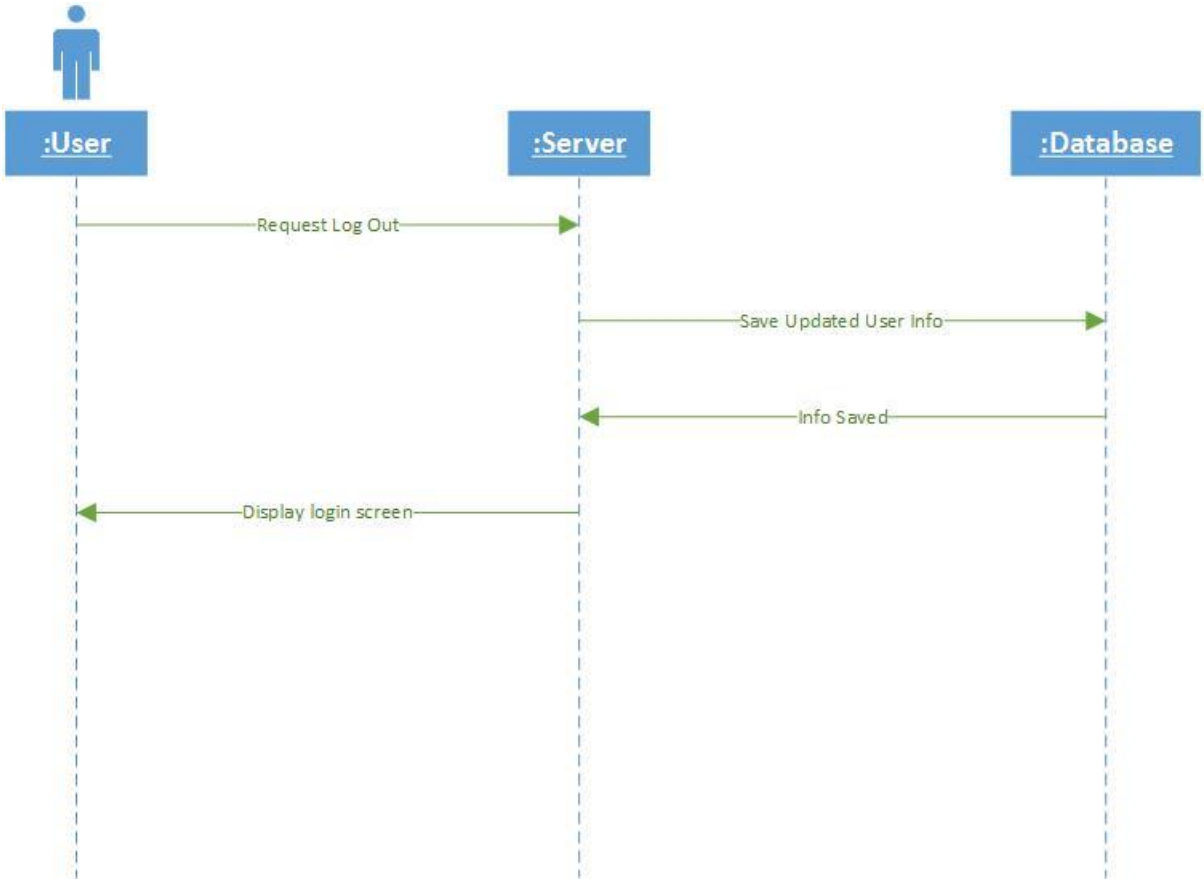
Successful Log Out:

User requests to log out
Request received by system
System sends relevant updated User info to database
System confirms that info is saved
System logs User out and sends User to login screen
User is displayed the login screen

Unsuccessful Log Out Because Request Could Not Be Received:

User loses internet connection
User requests to log out
Request could not be received by system because no internet connection
Student times out
System automatically logs out Student

Log Out High Level SSD



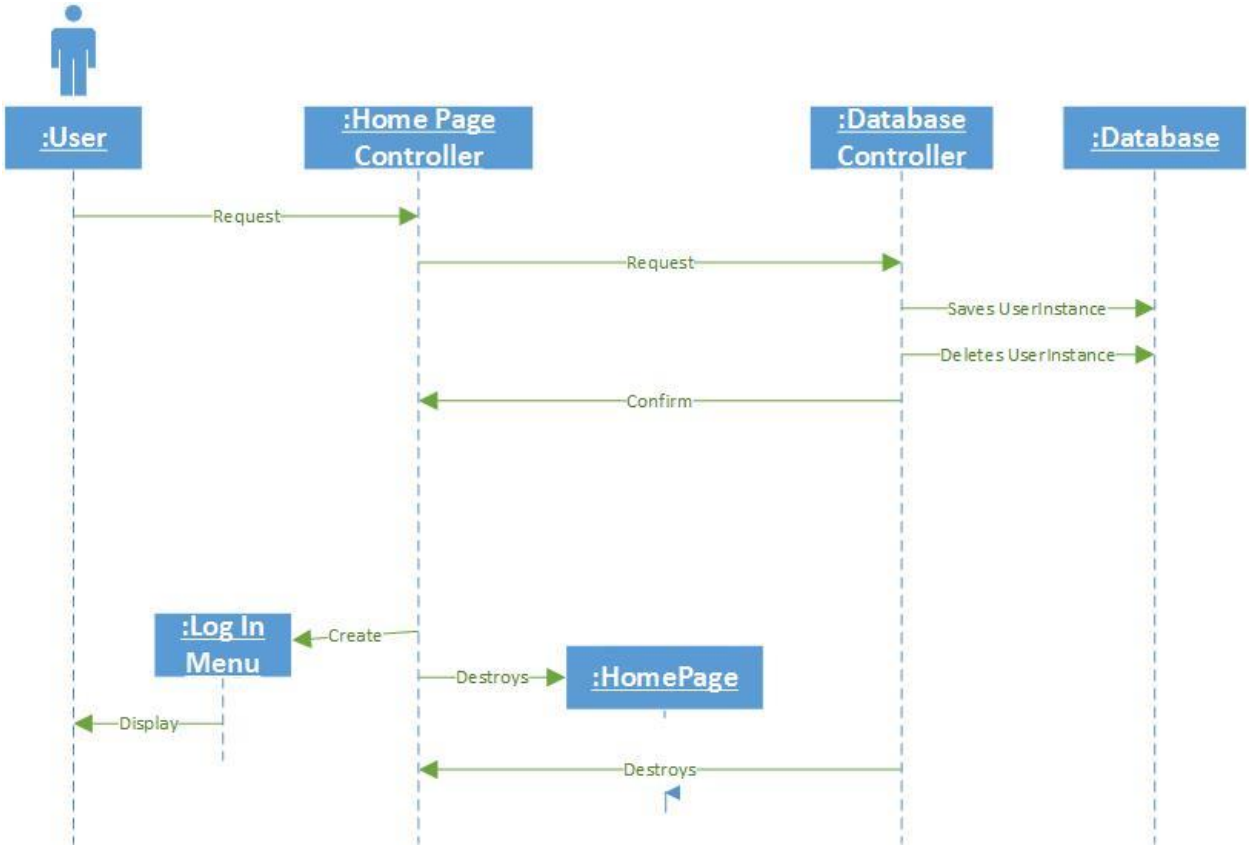
Log Out Use Case (Concrete)

Participants: User (Student, Professor, or Admin), Registration System

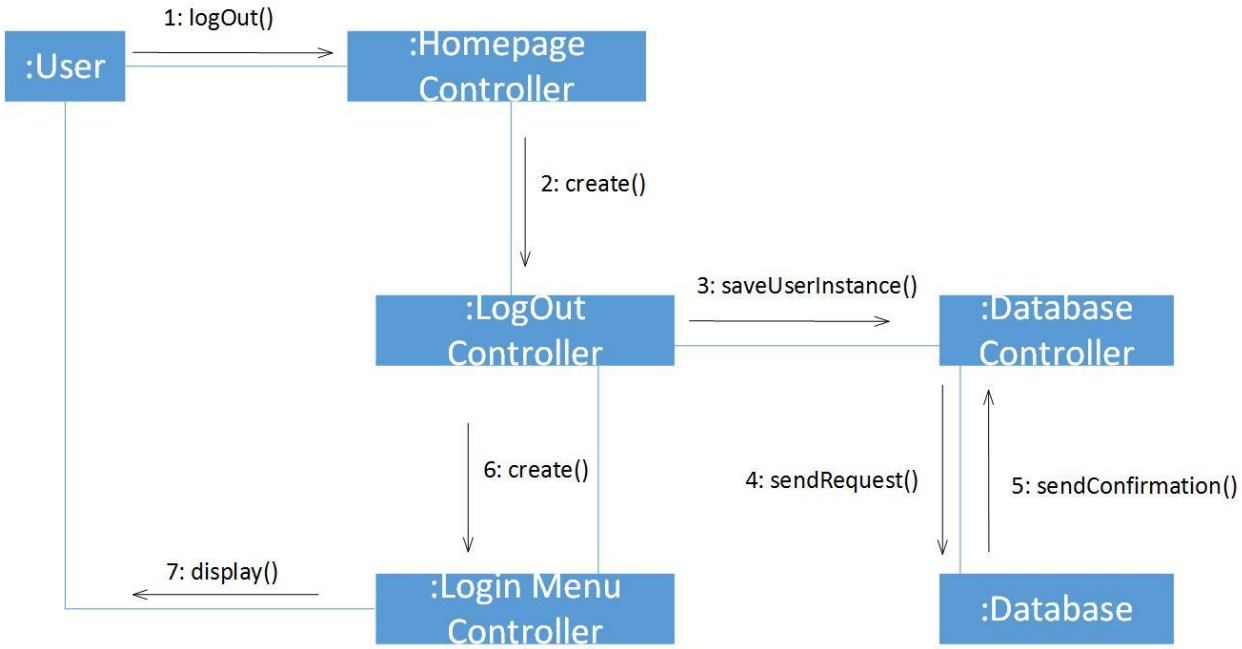
Precondition: User is already logged into the system.

Actor Intentions:	System Responsibility:
1. User clicks on log out button on homepage	2. Receives log out request
	3. Saves updated user info in database
	4. Confirms that user info was updated
	5. Logs out User
	6. Sends User to login screen
7. User is displayed the login screen	

Log Out Detailed SSD



Log Out Collaboration Diagram



Log Out GRASP Patterns

This collaboration diagram uses the GRASP patterns: creator, controller, cohesion, and coupling. After clicking the "Log Out" button, an updated user instance of the user is saved before displaying the login menu to the user once he/she has been logged out successfully. The controllers used are: Homepage Controller which displays the homepage specific to the type of user and gives the user options for further processes, LogOut Controller which handles saving the user instance and creating the last Controller, Database Controller which updates the data in the database, and LogIn Menu which is displayed at the end of the process to log back in if the user desires. Lower coupling is achieved by separating functionality into separate loosely connected objects. By doing this, higher cohesion is achieved because of the specific purpose of each object (i.e. controllers.)

Enter Final Course Grade Use Case (Essential)

Participants: User (Professor), Registration System

Precondition: Professor is already logged into the system.

Actor Intentions:	System Responsibility:
1. Professor clicks the button to enter final course grades	2. Receives enter final course grades request
	3. Displays currently enrolled classes to Professor to choose from
4. Professor chooses desired class	5. Pulls students in desired class from database
	6. Displays students to Professor with input for final grades
7. Professor inputs final grades and submits it to System	8. Receives grades and updates them in the database
	9. Displays message to Professor that grades were updated successfully

Enter Final Course Grade Scenarios

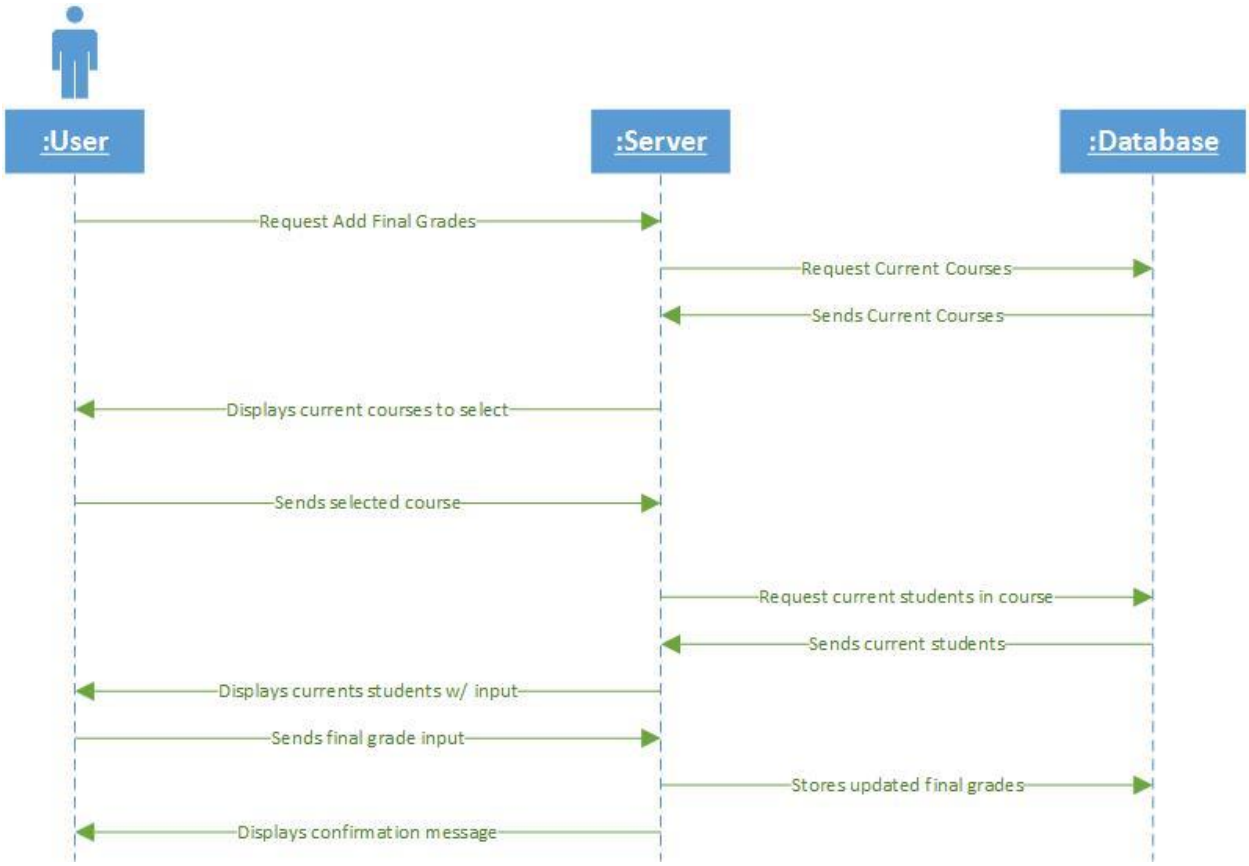
Successful Input of Final Course Grade:

Professor attempts to log in to system with user ID and password
System verifies user ID and password
System displays home screen to Professor
Professor request to enter final course grades
System receives request
System fetches current classes for Professor from database
System redirects Professor to page displaying current classes
Professor selects a current class
System receives selection
System fetches list of students in selected class from database
System redirects Professor to page displaying current students and input for final grades
Professor inputs final grades and submits it to system
System receives submit and saves updated final grades by sending them to the database
Database receives updated final grades and stores them successfully
System receives confirmation

Final Grades Update Unsuccessful Because of Invalid Input:

Professor attempts to log in to system with user ID and password
System verifies user ID and password
System displays home screen to Professor
Professor request to enter final course grades
System receives request
System fetches current classes for Professor from database
System redirects Professor to page displaying current classes
Professor selects a current class
System receives selection
System fetches list of students in selected class from database
System redirects Professor to page displaying current students and input for final grades
Professor inputs final grades which includes invalid characters (i.e. letters, symbols)
Professor submits final grades to system
System receives submit and generates error because of invalid characters
System sends message to Professor that final grades were unsuccessfully updated (because of invalid input) and requests for new input

Enter Final Course Grade High Level SSD



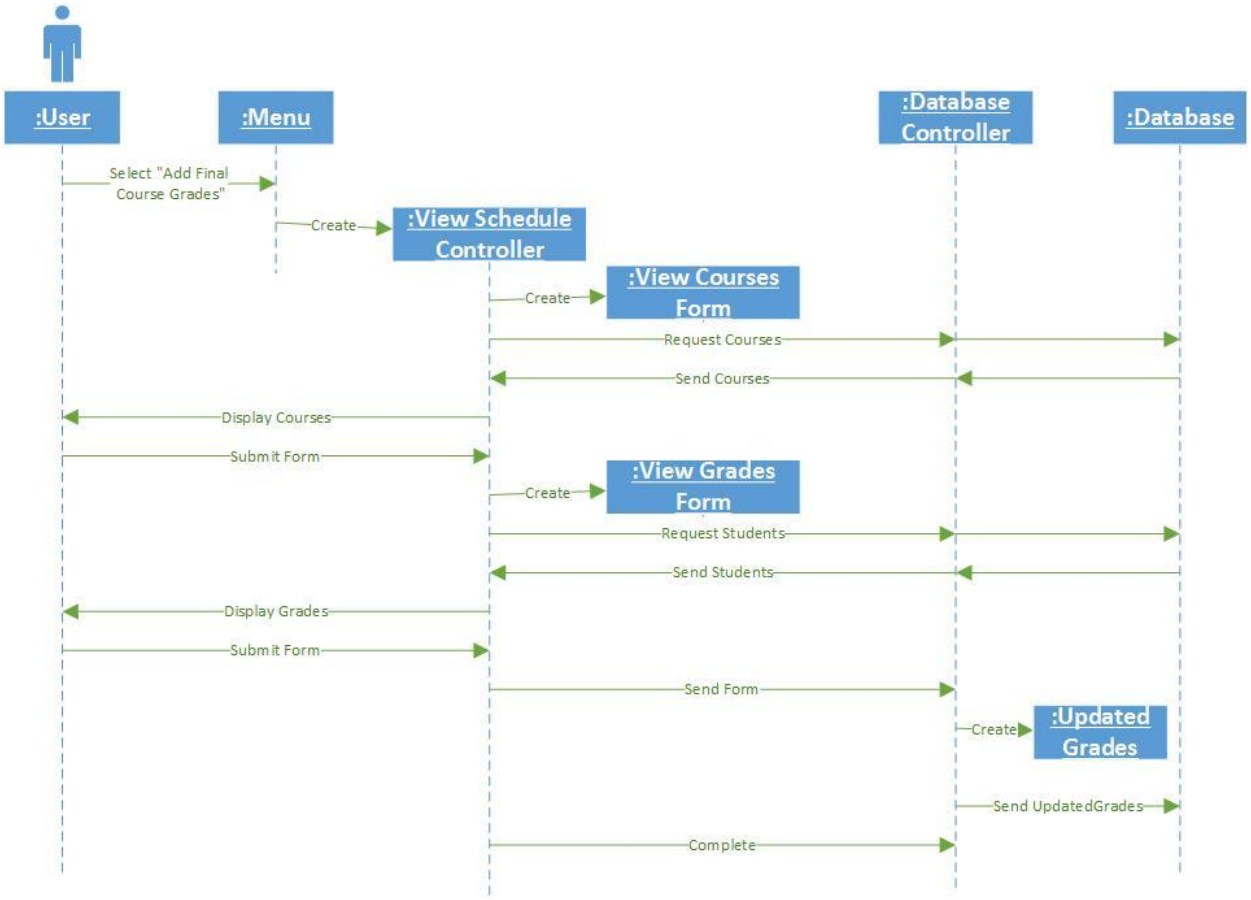
Enter Final Grade Use Case (Concrete)

Participants: User (Professor), Registration System

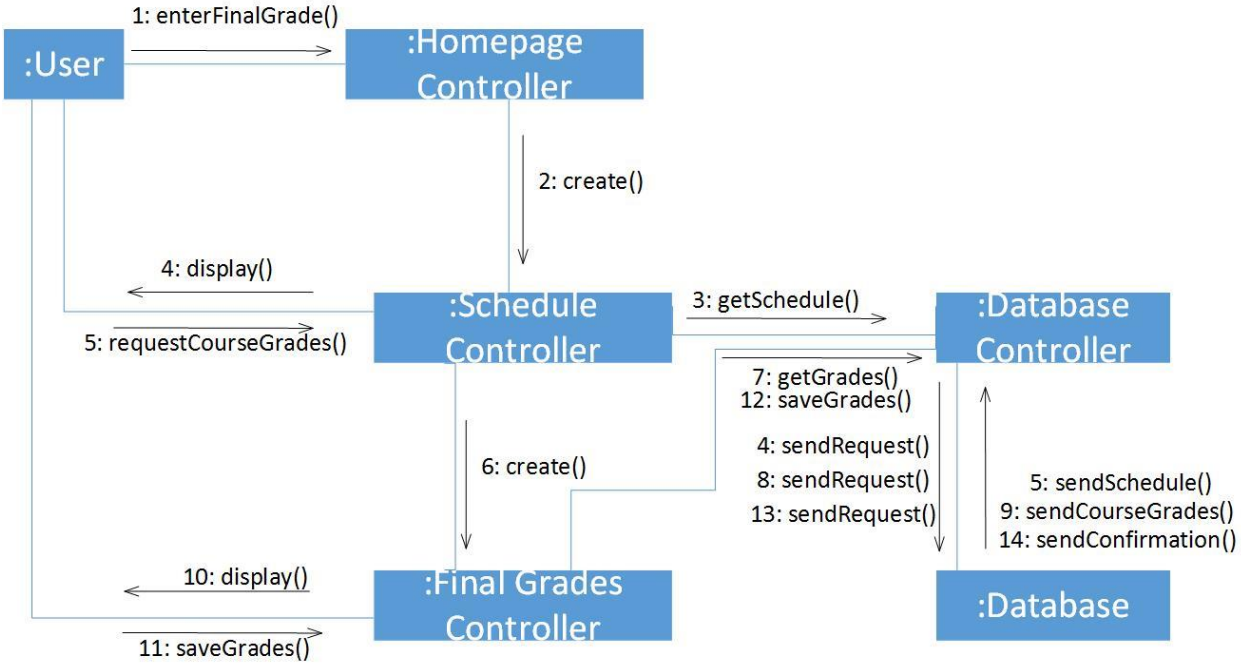
Precondition: Professor is already logged into the system.

Actor Intentions:	System Responsibility:
1. Professor clicks the button to enter final course grades	2. Receives enter final course grades request
	3. Compiles list of currently enrolled classes
4. Displayed currently enrolled classes to Professor to choose from	
5. Professor chooses desired class to enter final course grades	6. Compiles list of students in selected class from database
7. Displayed students to Professor with input for final grades	
8. Professor inputs final grades and submits it to System	9. Receives grades and updates them in the database
10. Displays message to Professor that grades were updated successfully	

Enter Final Grade Detailed SSD



Enter Final Course Grade Collaboration Diagram



Enter Final Course Grade GRASP Patterns

This collaboration diagram uses the GRASP patterns: creator, controller, cohesion, and coupling. Once the user has logged in and clicked the "Enter Final Grade" button, a controller specific to the view schedule (i.e. courses to choose from) process will be created. Once the user has chosen the desired course, a controller specific to the enter final grade process will be created, displaying the list of students in the course and an option for the user to enter final grades. The controllers used are: Homepage Controller which displays the homepage specific to the type of user and gives the user options for further processes, Schedule Controller which handles displaying current courses for the user to choose from and fetching the students in the selected course, Database Controller which updates the data in the database, and Final Grades Controller which handles adding final grades for the selected students. Lower coupling is achieved by separating functionality into separate loosely connected objects. By doing this, higher cohesion is achieved because of the specific purpose of each object (i.e. controllers.)

Add Courses Use Case (Essential)

Participants: User (Student, Professor), Registration System

Precondition: User is already logged into the system.

Actor Intentions:	System Responsibility:
1. User clicks on add course button on homepage	2. Receives add course request
	3. System checks database for current courses of User
4. User displayed current courses, input for new courses, and a class search button	
5. User inputs CRN for new course and submits it to the System	6. System receives input and attempts to match CRN with available courses in the database
	7. System matches CRN and adds class to User schedule (if the class is not full)
8. User displayed message that class was added successfully	

Add Courses Scenarios

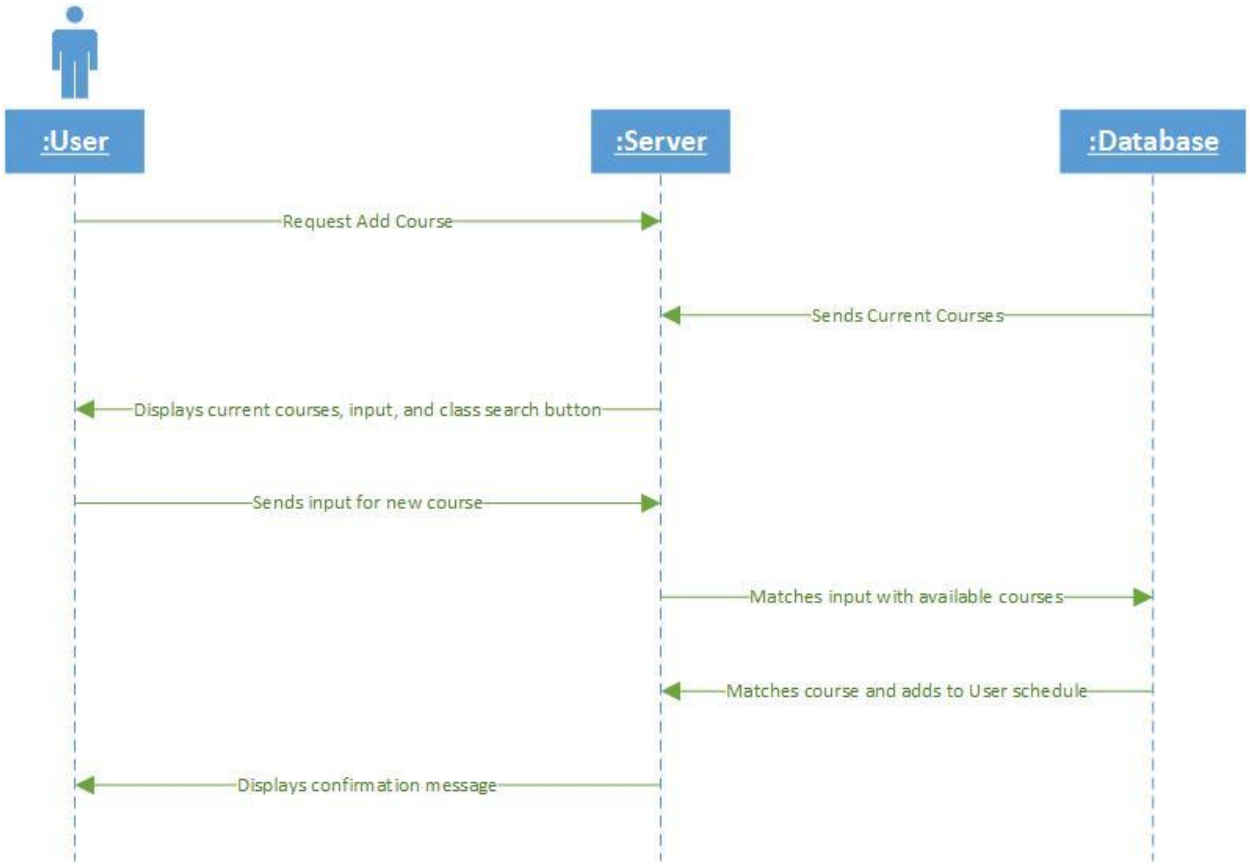
Course Added Successfully by CRN:

Student attempts to log in to system with user ID and password
System verifies user ID and password
System displays home screen to Student
Student requests to add a course
System receives request
System requests current courses for Student from database
Database displays current courses, input for a new course via CRN, and option to search classes to Student
Student inputs CRN of class he/she wants to be added and sends request
System receives request and searches the database for course via CRN
System matches CRN with valid course
System confirms course is not full
System adds Student to course and saves it in the database
System displays message to the Student that he/she was added to the course successfully

Course Not Added Because Course Is Full:

Student attempts to log in to system with user ID and password
System verifies user ID and password
System displays home screen to Student
Student requests to add a course
System receives request
System requests current courses for Student from database
Database displays current courses, input for a new course via CRN, and option to search classes to Student
Student inputs CRN of class he/she wants to be added and sends request
System receives request and searches the database for course via CRN
System matches CRN with valid course
System sees that the course is full
System displays message to the Student that the class is full and requests for new input

Add Courses High Level SSD



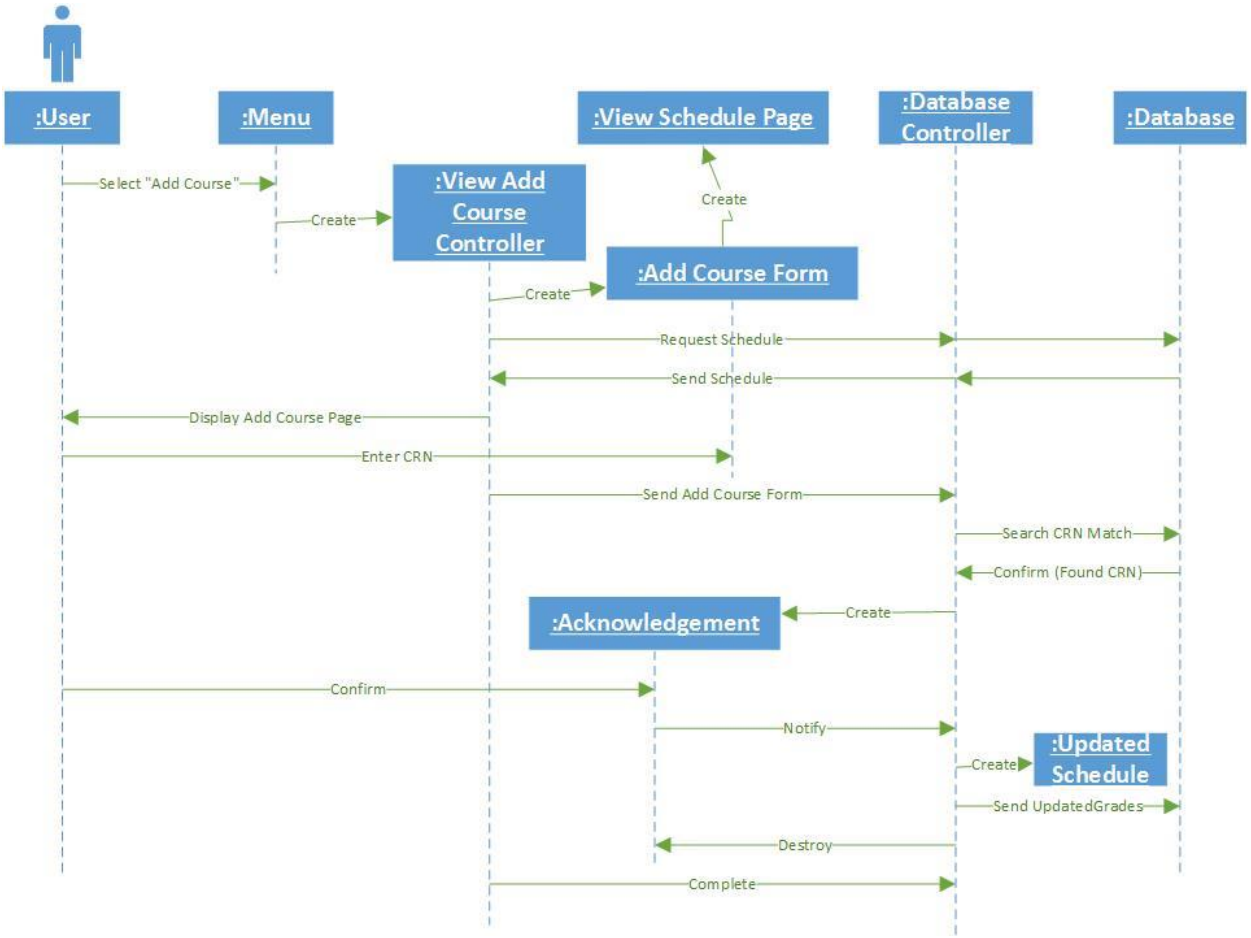
Add Courses Use Case (Concrete)

Participants: User (Student, Professor), Registration System

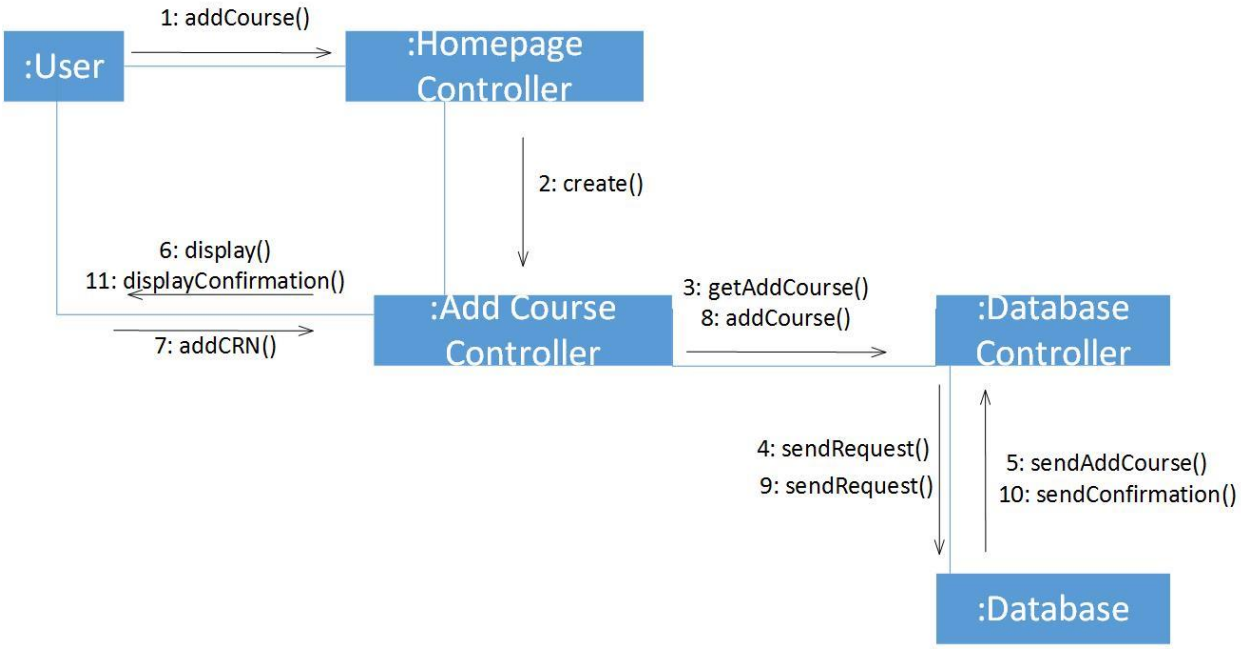
Precondition: User is already logged into the system.

Actor Intentions:	System Responsibility:
1. User clicks on add course button on homepage	2. Receives add course request
	3. System checks database for current courses of User
4. User displayed current courses, input for new courses, and a class search button	
5. User inputs the integer of the CRN for desired new course and submits it to the System	6. System receives input and attempts to match the inputted CRN (integer) with available course CRNs (integers) in the database
	7. System finds match of CRN and adds class to User schedule (if the class is not full)
8. User displayed message that class was added successfully	

Add Courses Detailed SSD



Add Course Collaboration Diagram



Add Course GRASP Patterns

This collaboration diagram uses the GRASP patterns: creator, controller, cohesion, and coupling. Once the user has logged in and clicked the "Add Course" button, a controller specific to the add course process will be created to handle adding a new course. The controllers used are: Homepage Controller which displays the homepage specific to the type of user and gives the user options for further processes, Add Course Controller which handles matching the CRN (input by the user) with a current course and adding it to the database, and the Database Controller which updates the data in the database. Lower coupling is achieved by separating functionality into separate loosely connected objects. By doing this, higher cohesion is achieved because of the specific purpose of each object (i.e. controllers.)

View Enrollment Summary Use Case (Essential)

Participants: User (Professor), Registration System

Precondition: Professor is already logged into the system.

Actor Intentions:	System Responsibility:
1. Professor clicks on view enrollment summary button on homepage	2. Receives view enrollment summary request
	3. Compiles list of students from Professor's currently enrolled classes
4. Professor is displayed list of current students	

View Enrollment Summary Scenarios

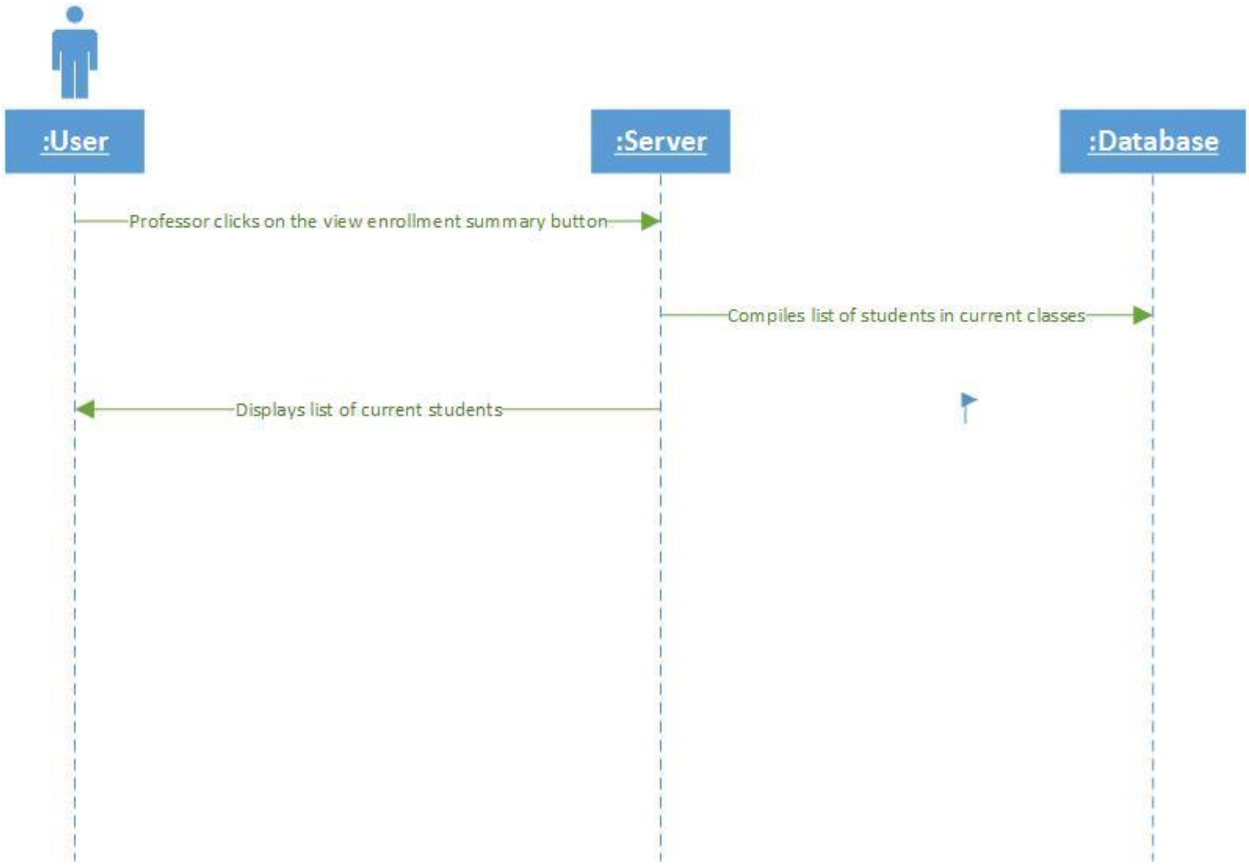
View Enrollment Summary Successful:

Professor attempts to log in to system with user ID and password
System verifies user ID and password
System displays home screen to Professor
Professor request to view enrollment summary
System receives request
System fetches list of current students from database
(by compiling list of students in each course the Professor is enrolled in)
System displays list of current students to Professor

View Enrollment Summary Unsuccessful:

Professor attempts to log in to system with user ID and password
System verifies user ID and password
System displays home screen to Professor
Professor request to view enrollment summary
System receives request
System attempts to fetch list of current students from database
Request unsuccessful because the Professor is not enrolled in any courses
System displays message to Professor to add a course

View Enrollment Summary High Level SSD



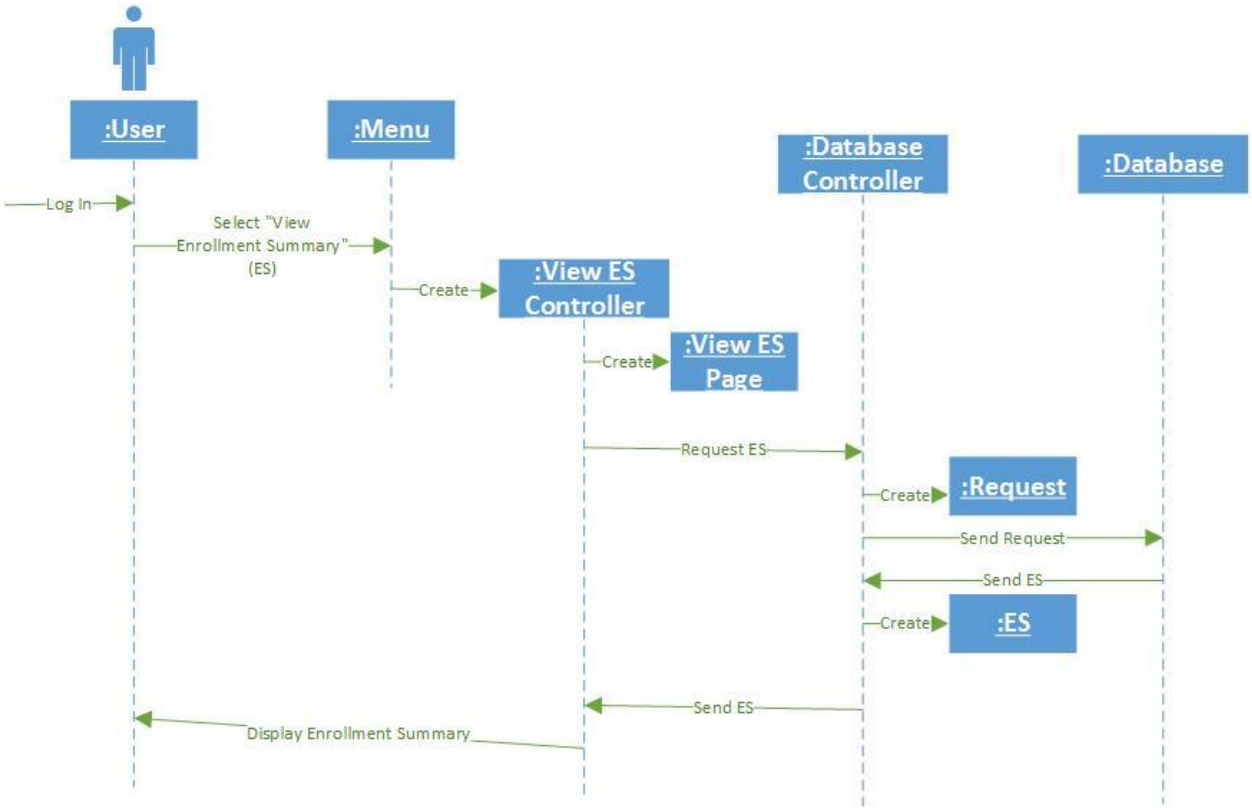
View Enrollment Summary Use Case (Concrete)

Participants: User (Professor), Registration System

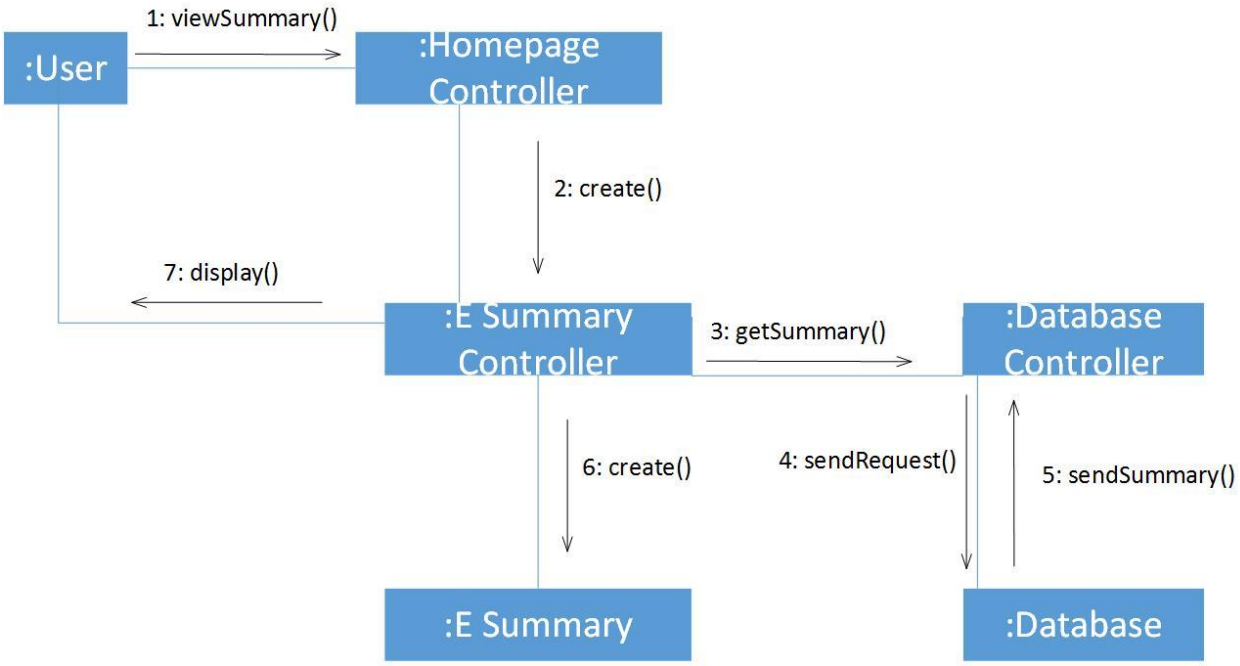
Precondition: Professor is already logged into the system.

Actor Intentions:	System Responsibility:
1. Professor clicks on view enrollment summary button on homepage	2. Receives view enrollment summary request
	3. Compiles list of students from Professor's currently enrolled classes
4. Professor is displayed list of current students	

View Enrollment Summary Detailed SSD



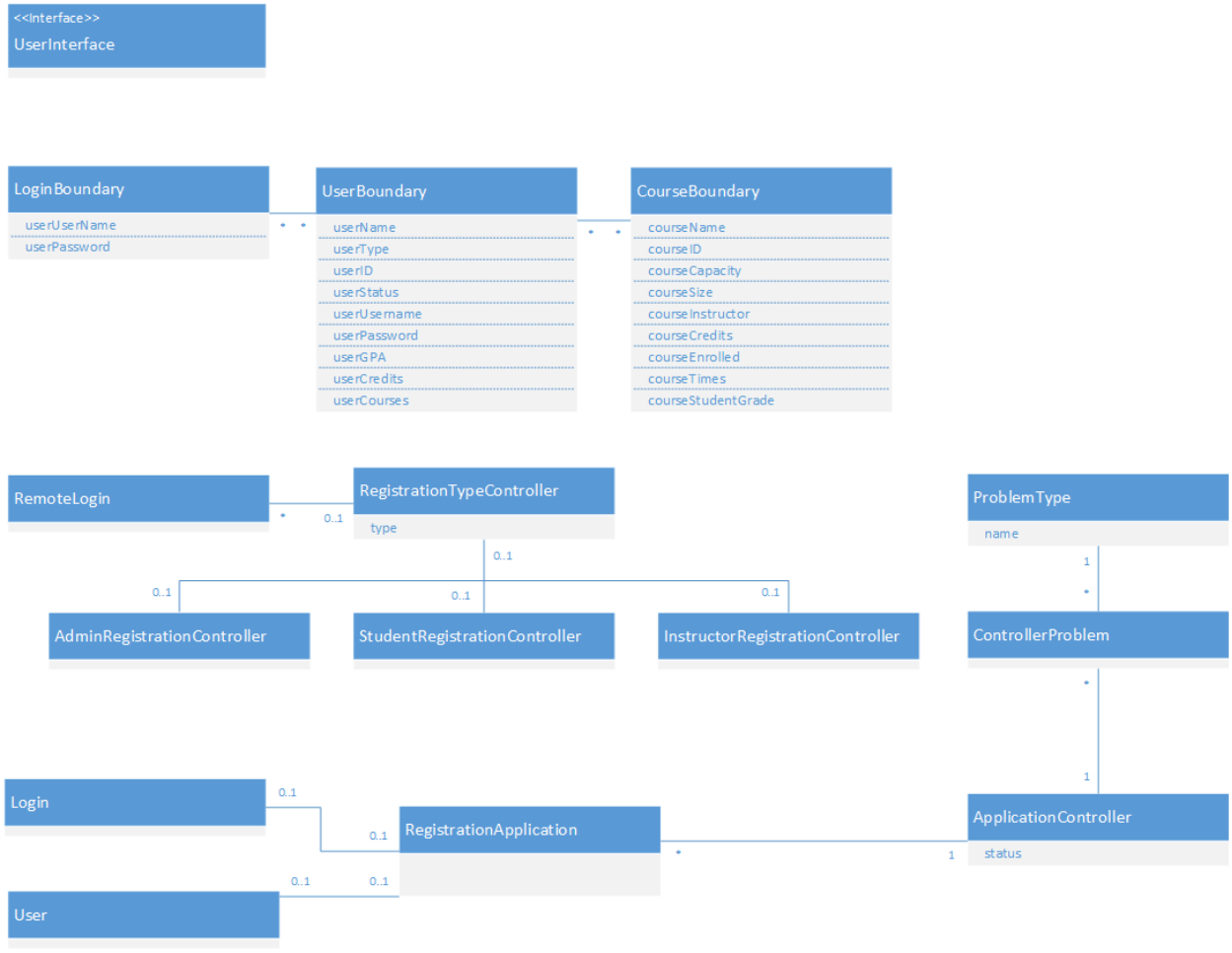
View Enrollment Summary Collaboration Diagram



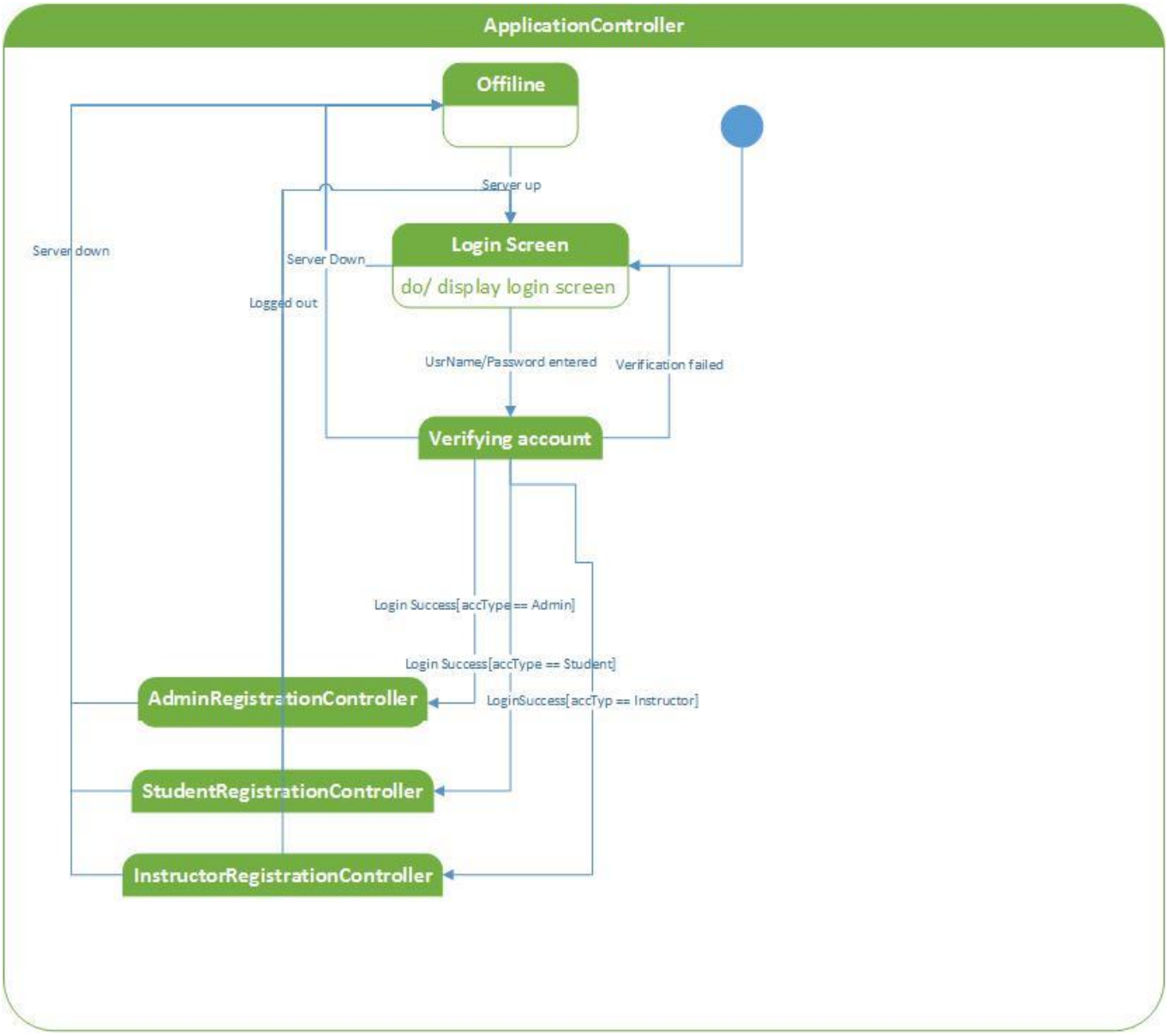
View Enrollment GRASP Patterns

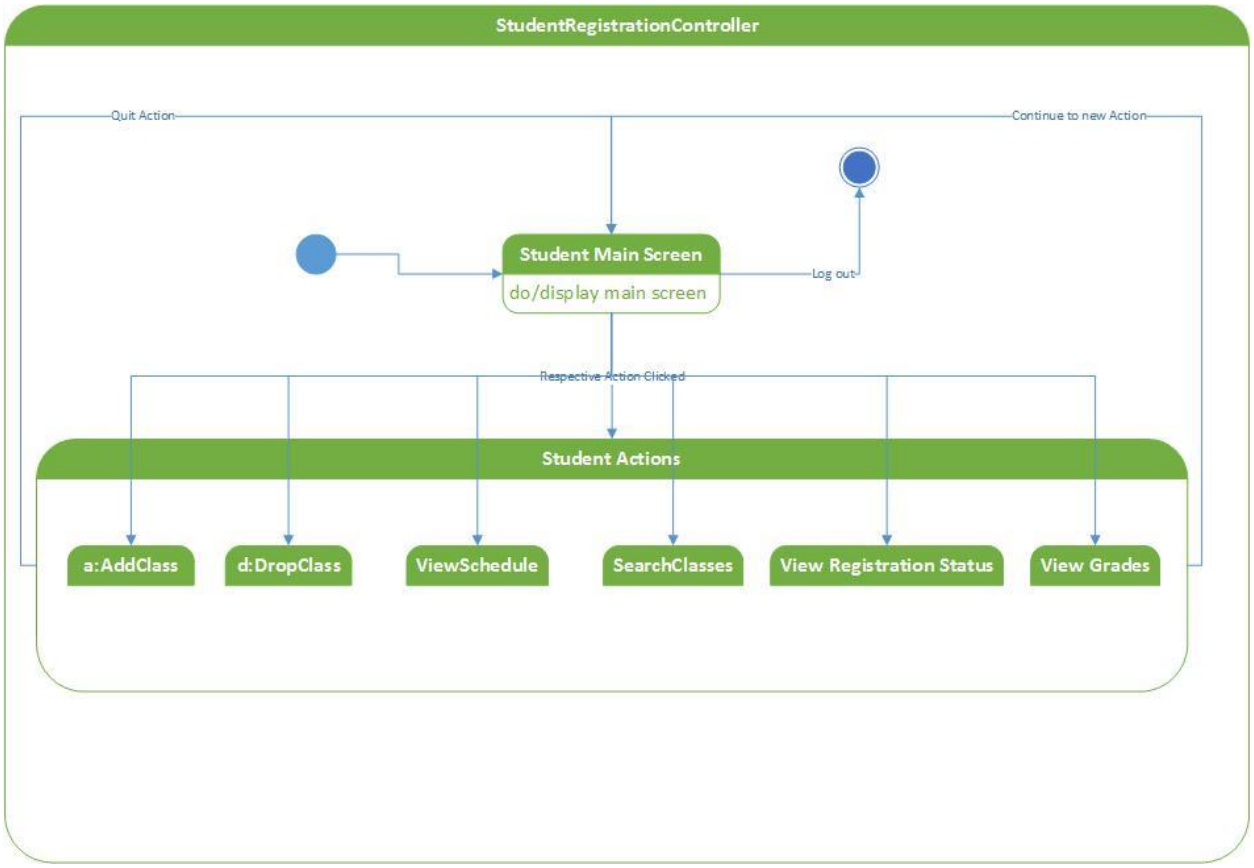
This collaboration diagram uses the GRASP patterns: creator, controller, cohesion, and coupling. Once the user has logged in and clicked the "View Summary" button, a controller specific to the view summary process will be created to handle fetching user specific data from the database and displaying it the user. The controllers used are: Homepage Controller which displays the homepage specific to the type of user and gives the user options for further processes, E Summary Controller which handles displaying user specific data, and the Database Controller which updates the data in the database. Lower coupling is achieved by separating functionality into separate loosely connected objects. By doing this, higher cohesion is achieved because of the specific purpose of each object (i.e. controllers.)

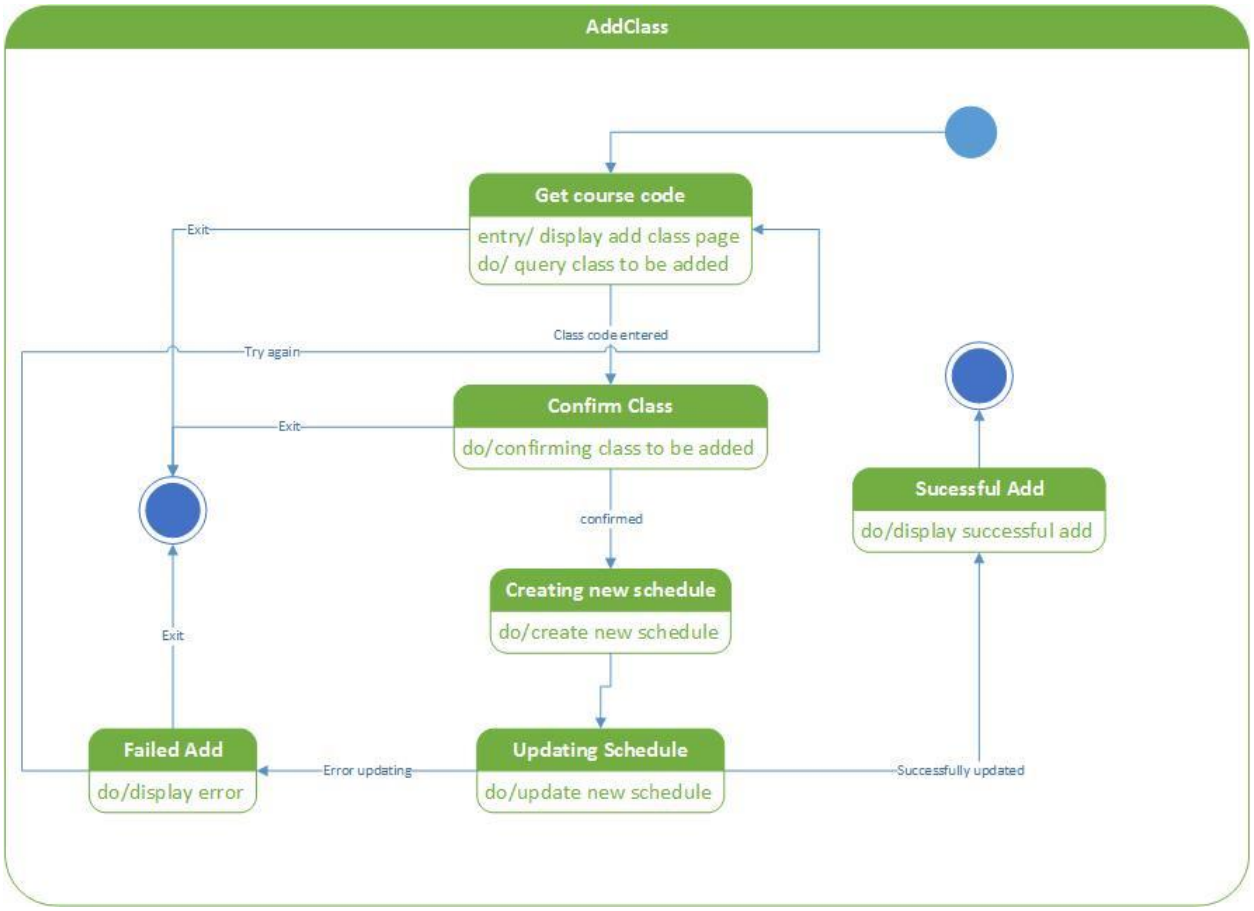
Application Class Model

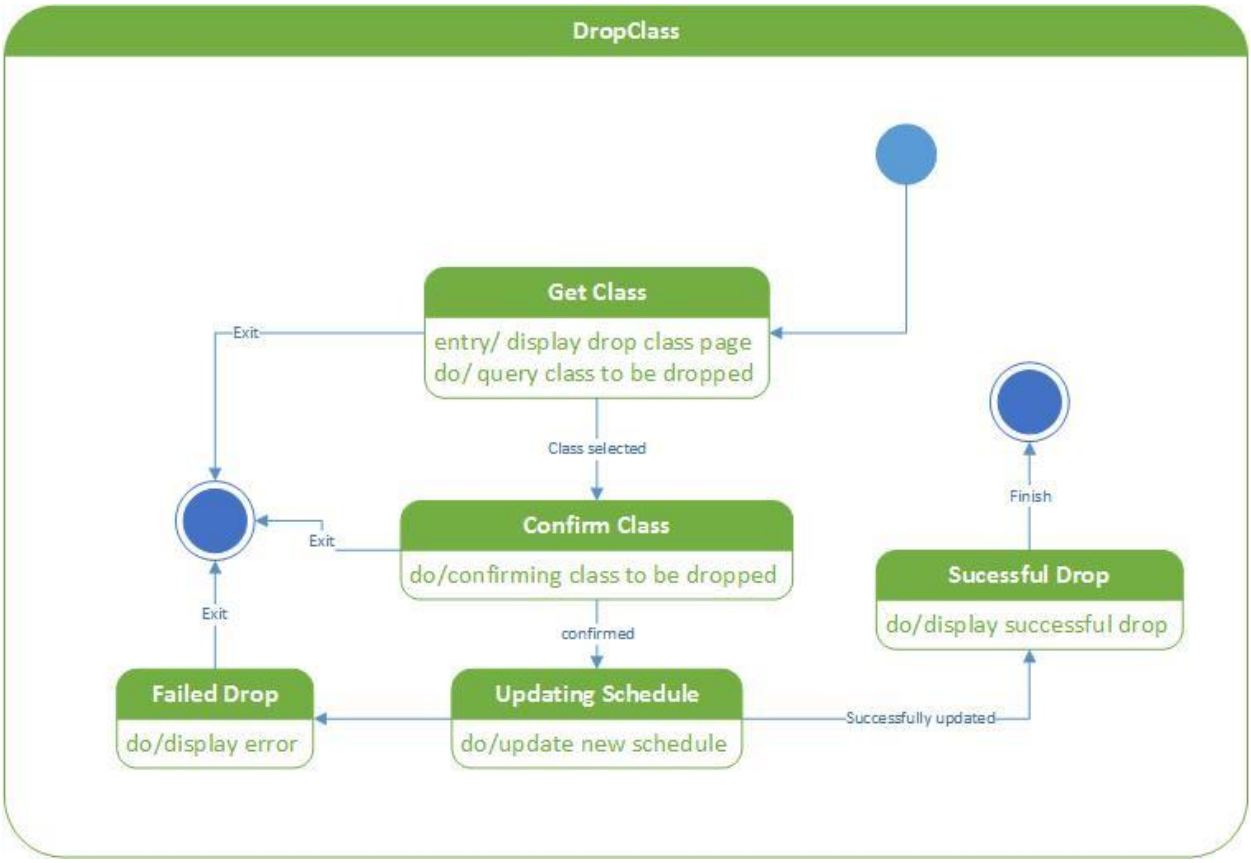


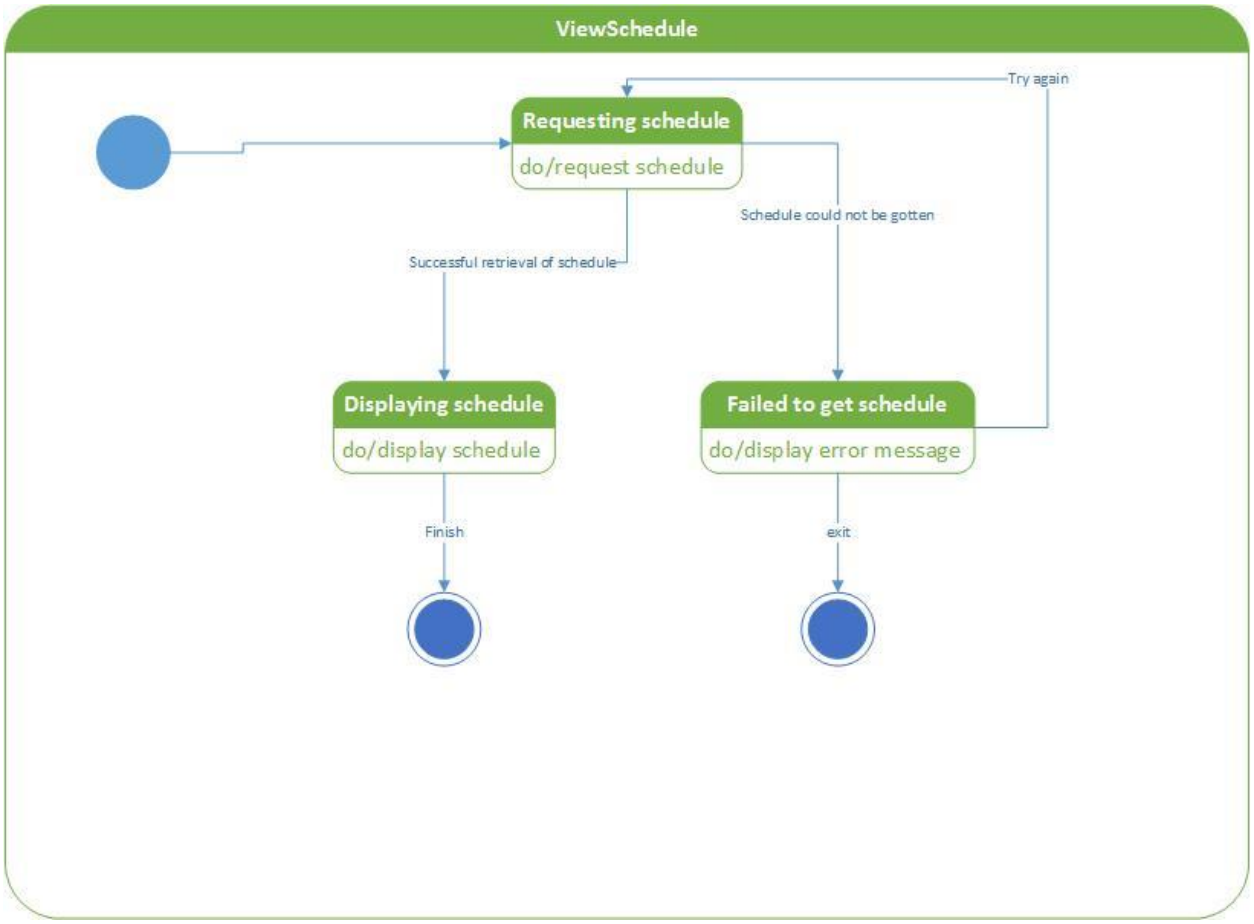
Application State Model

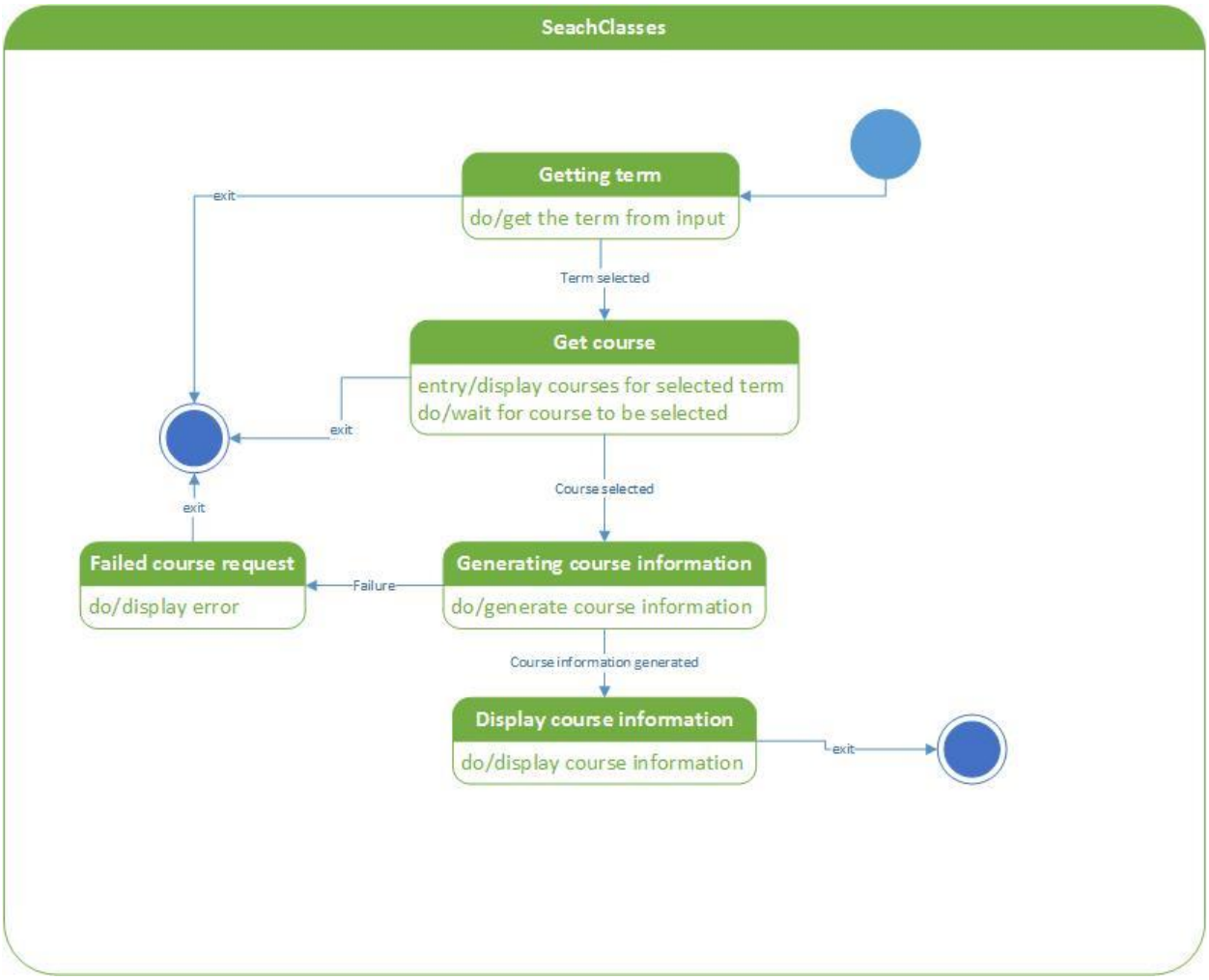


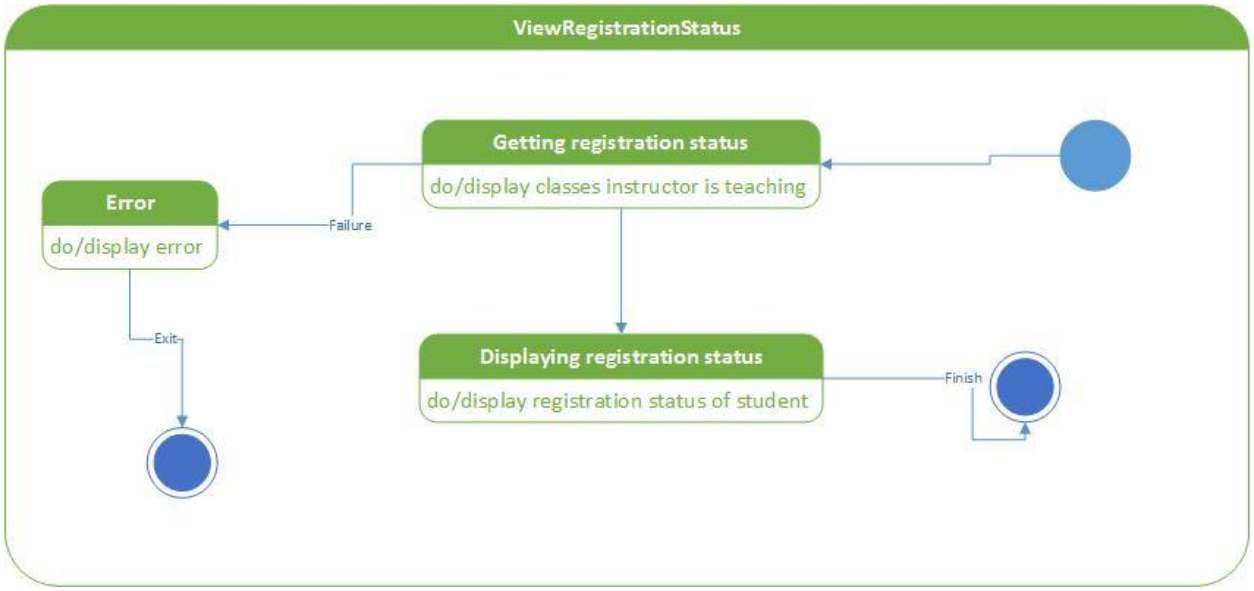

StudentRegistrationController
Quit Action
Continue to new Action
Student Main Screen
do/display main screen
Log out
Respective Action Clicked
Student Actions
a:AddClass
d:DropClass
ViewSchedule
SearchClasses
View Registration Status
View Grades

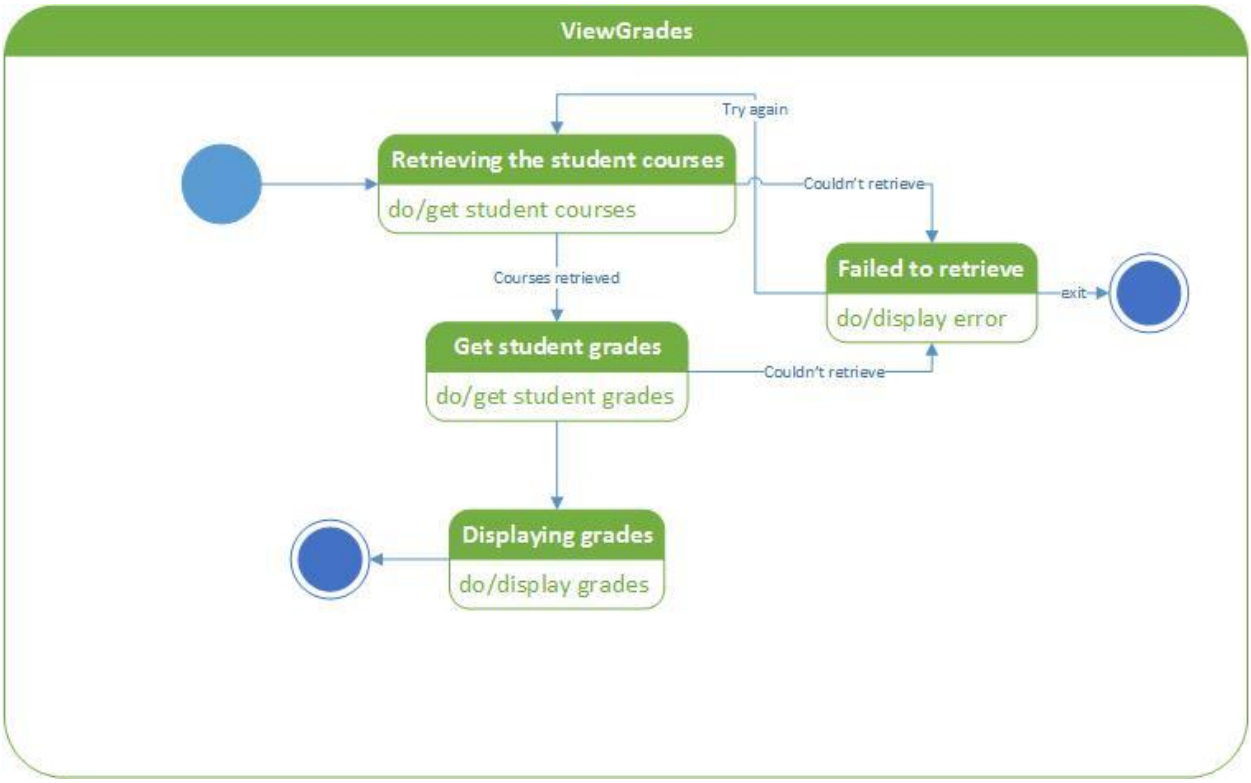

AddClass
Get course code
entry/ display add class page
do/ query class to be added
Exit
Class code entered
Try again
Confirm Class
do/confirming class to be added
Exit
confirmed
Creating new schedule
do/create new schedule
Updating Schedule
do/update new schedule
Error updating
Failed Add
do/display error
Exit
Successfully updated
Sucessful Add
do/display successful add

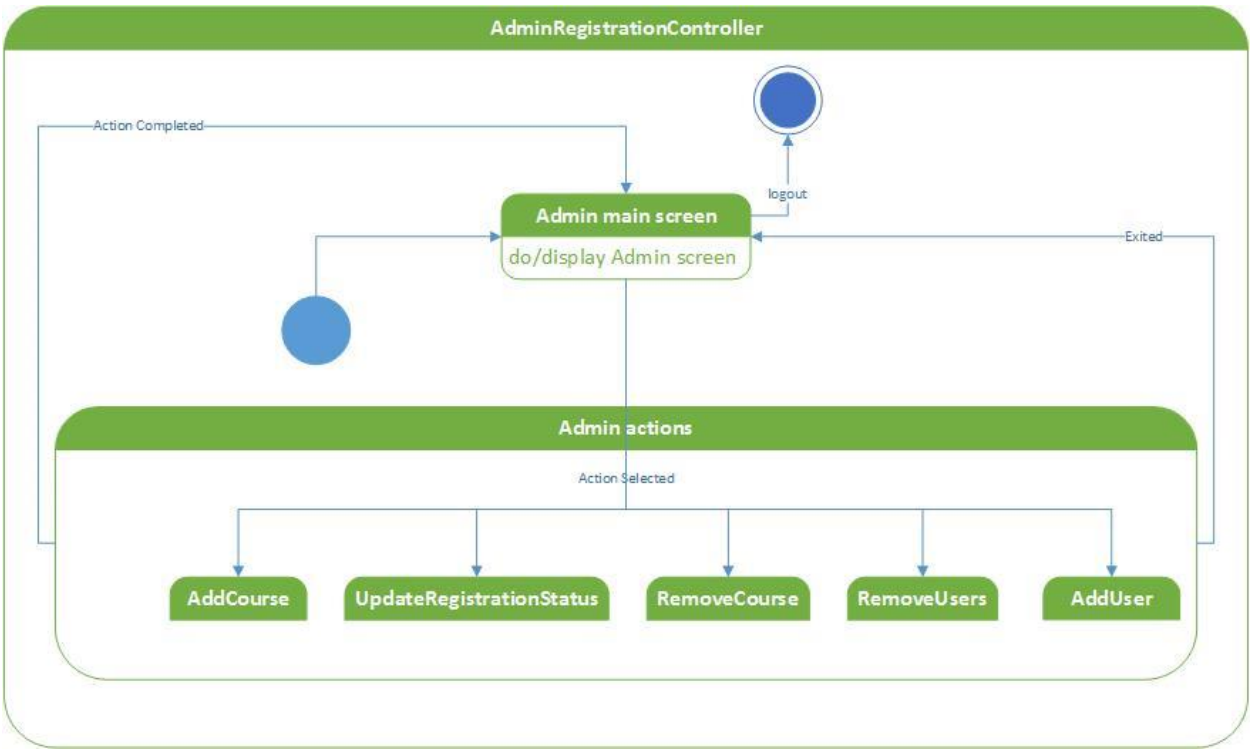

DropClass
Get Class
entry/ display drop class page
do/ query class to be dropped
Exit
Class selected
Confirm Class
do/confirming class to be dropped
Exit
Exit
confirmed
Failed Drop
do/display error
Updating Schedule
do/update new schedule
Successfully updated
Finish
Sucessful Drop
do/display successful drop

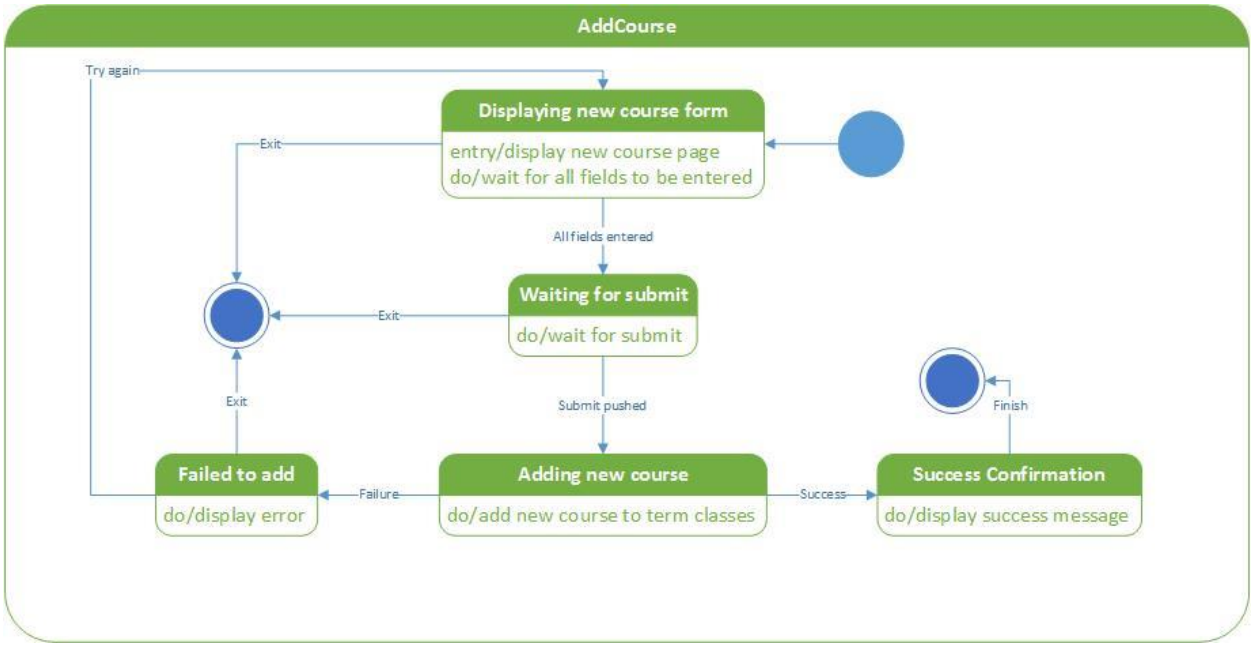

ViewSchedule
Try again
Requesting schedule
do/request schedule
Schedule could not be gotten
Successful retrieval of schedule
Displaying schedule
do/display schedule
Failed to get schedule
do/display error message
Finish
exit

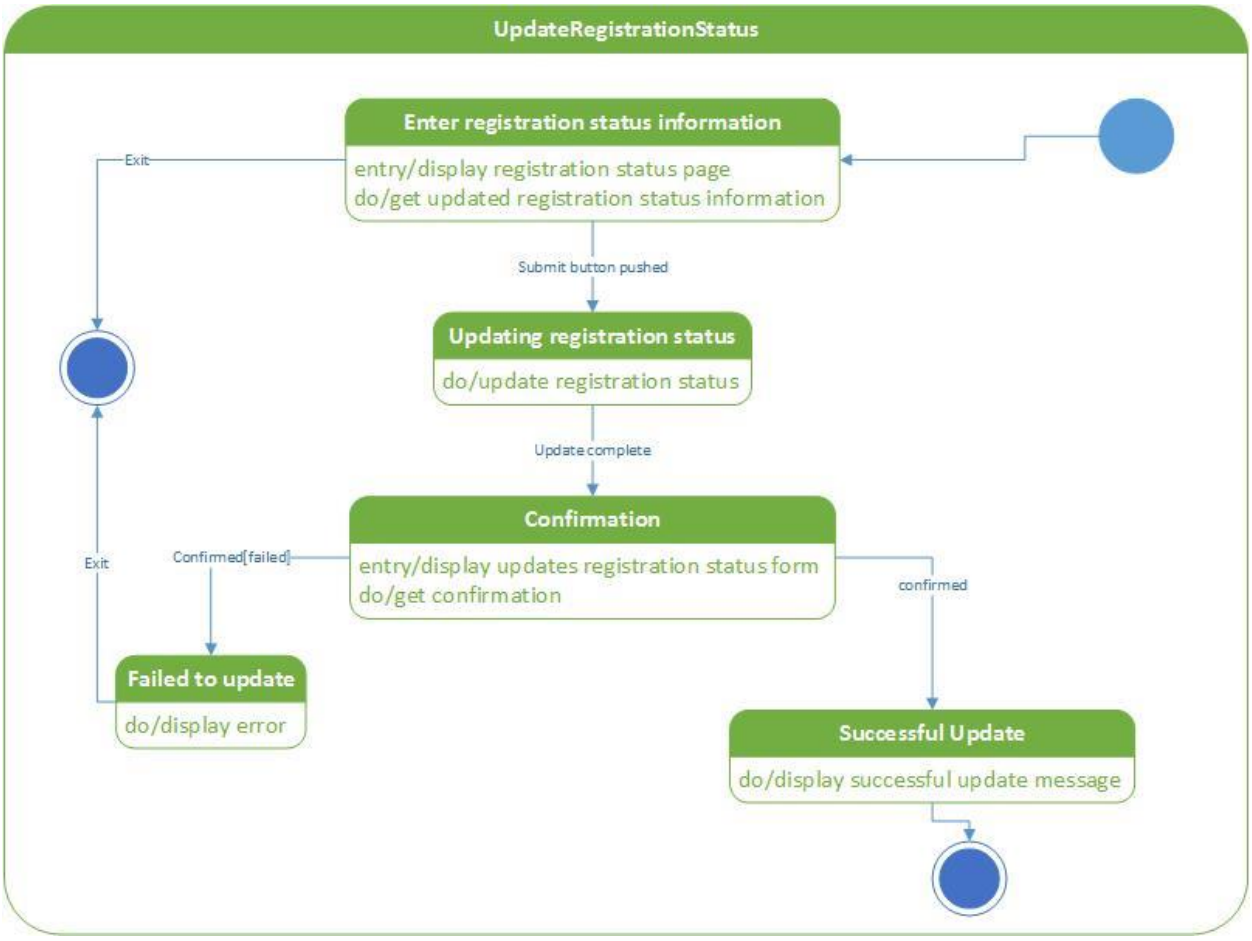

SeachClasses
Getting term
do/get the term from input
exit
Term selected
Get course
entry/display courses for selected term
do/wait for course to be selected
exit
Course selected
exit
Failed course request
do/display error
Failure
Generating course information
do/generate course information
Course information generated
Display course information
do/display course information
exit

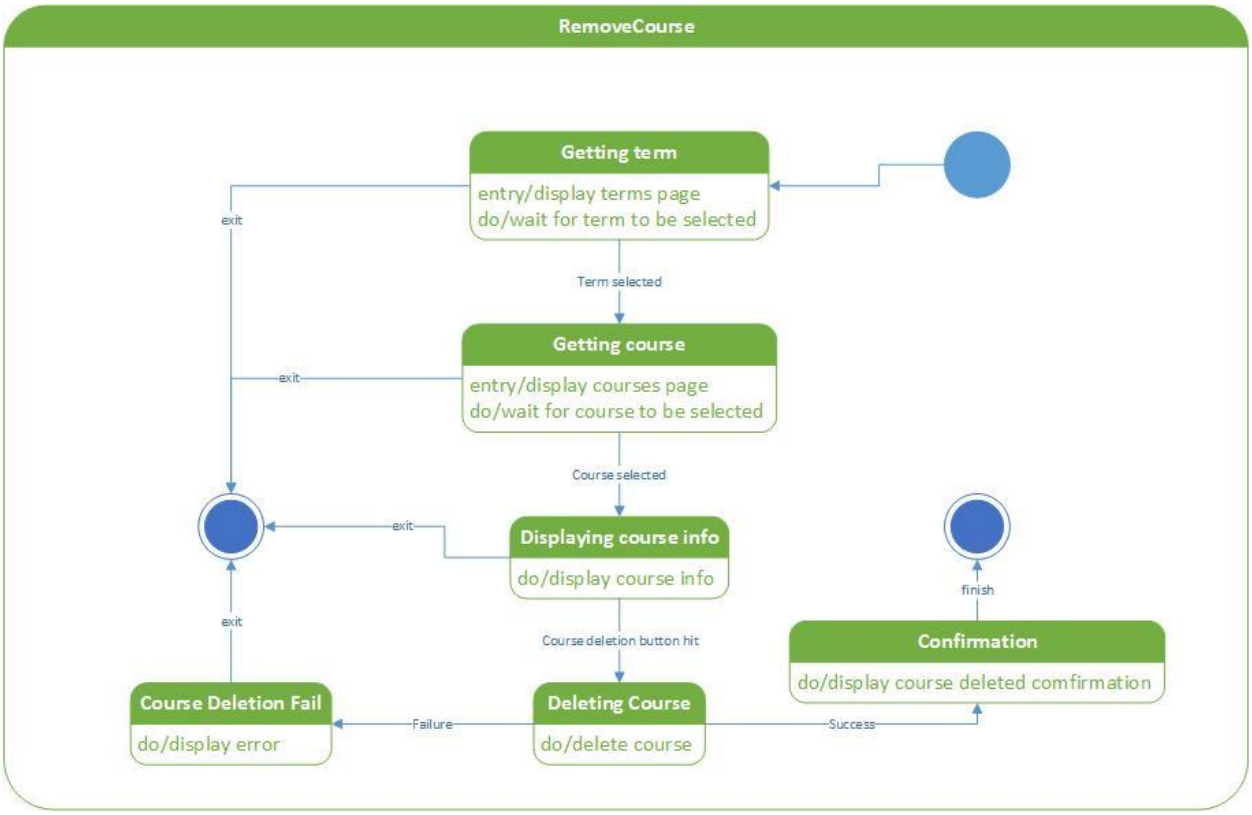

ViewRegistrationStatus
Getting registration status
do/display classes instructor is teaching
Error
do/display error
Failure
Exit
Displaying registration status
do/display registration status of student
Finish

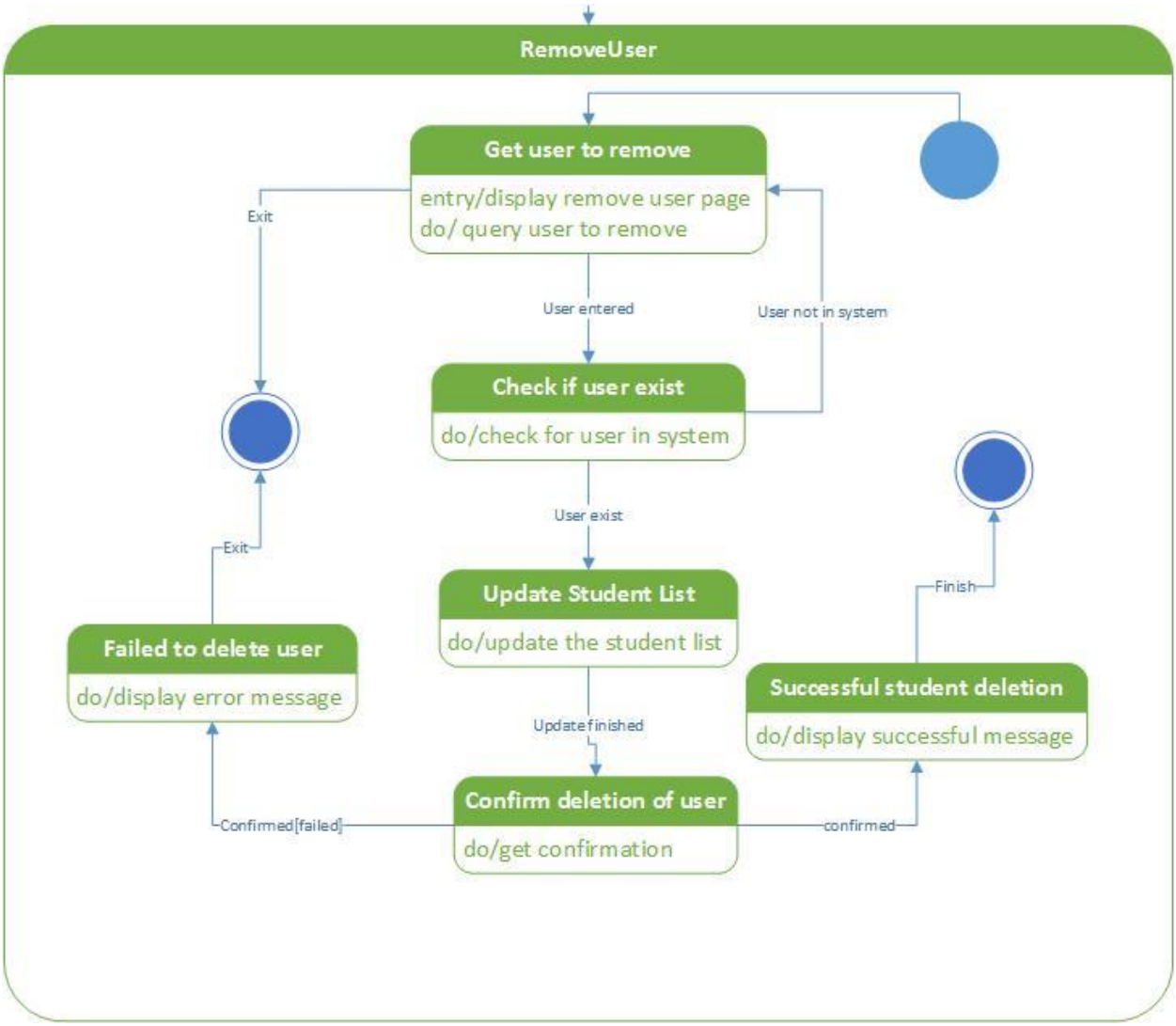

ViewGrades
Try again
Retrieving the student courses
do/get student courses
Couldn't retrieve
Courses retrieved
Failed to retrieve
do/display error
exit
Get student grades
do/get student grades
Couldn't retrieve
Displaying grades
do/display grades

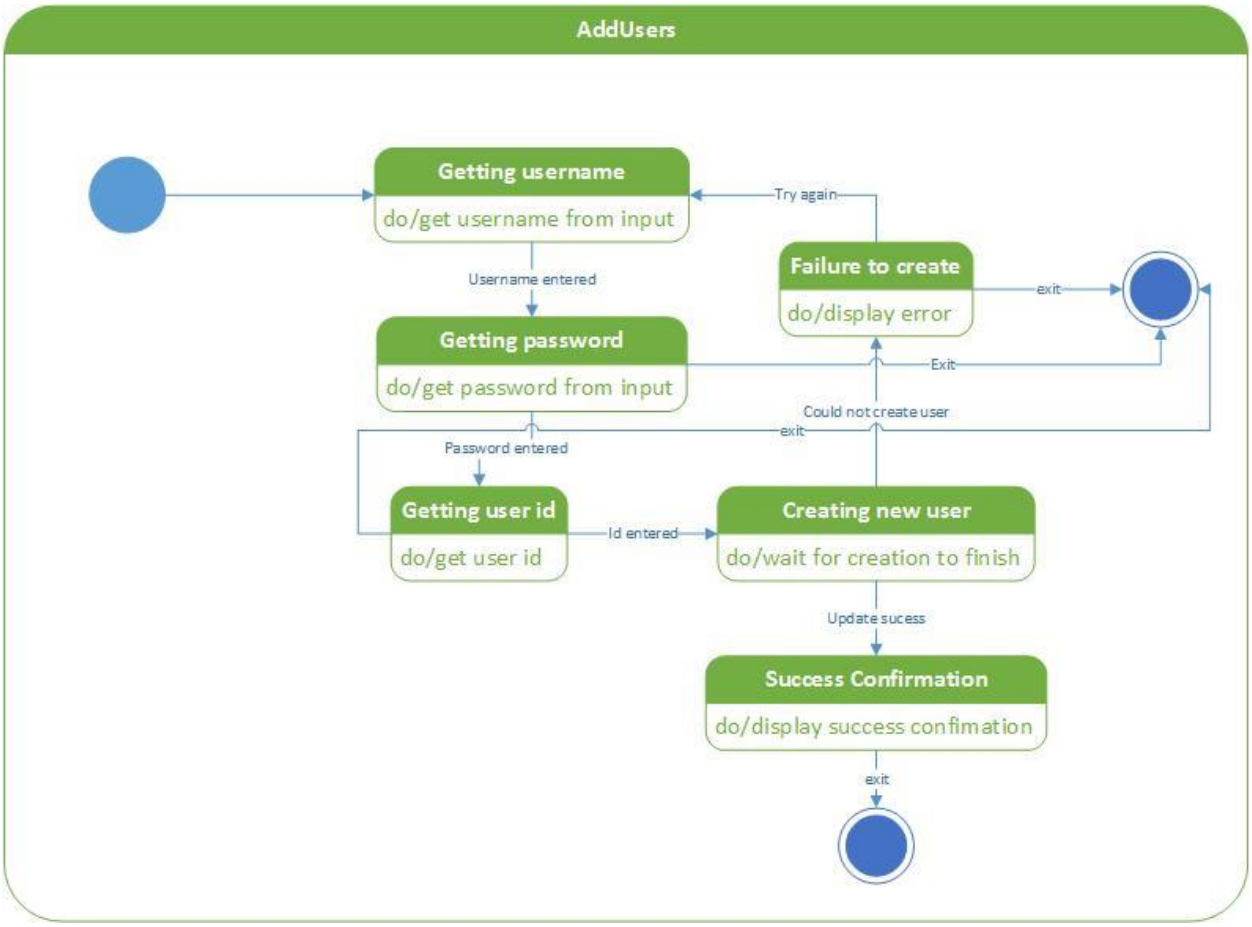

AdminRegistrationController
Action Completed
logout
Admin main screen
do/display Admin screen
Exited
Admin actions
Action Selected
AddCourse
UpdateRegistrationStatus
RemoveCourse
RemoveUsers
AddUser

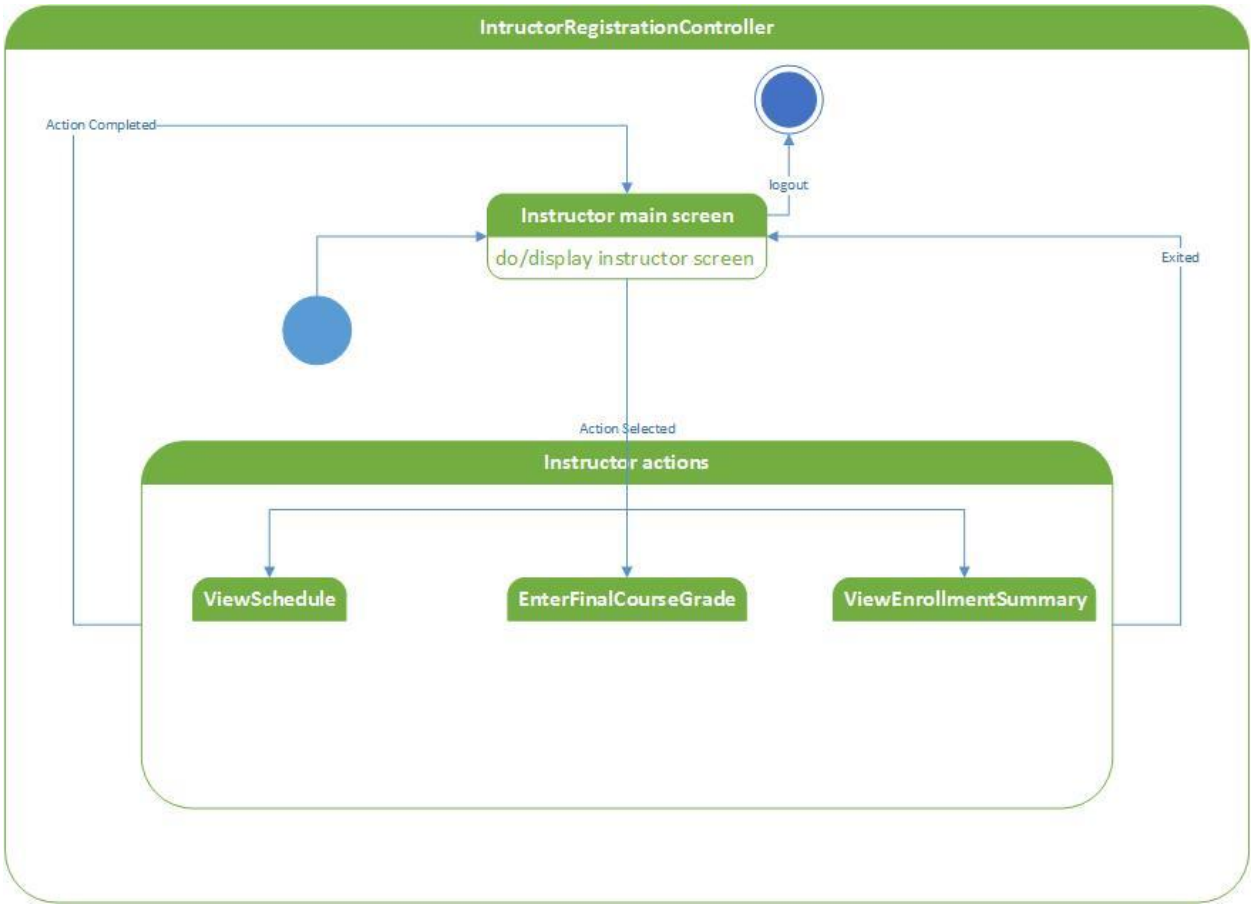

AddCourse
Try again
Displaying new course form
entry/display new course page
do/wait for all fields to be entered
Exit
All fields entered
Waiting for submit
do/wait for submit
Exit
Exit
Submit pushed
Finish
Failed to add
do/display error
Failure
Adding new course
do/add new course to term classes
Success
Success Confirmation
do/display success message

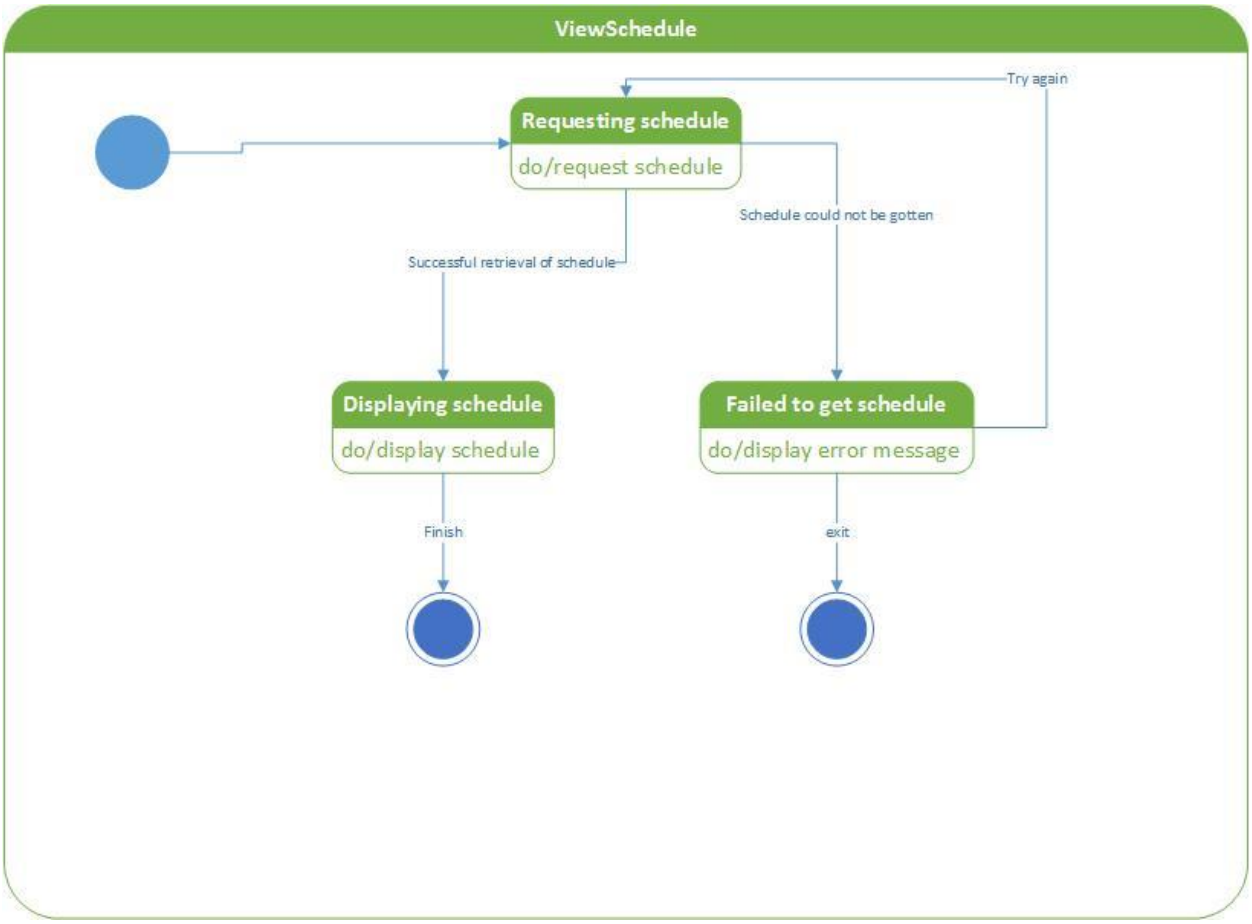

UpdateRegistrationStatus
Enter registration status information
entry/display registration status page
do/get updated registration status information
Exit
Submit button pushed
Updating registration status
do/update registration status
Update complete
Confirmation
entry/display updates registration status form
do/get confirmation
Confirmed[failed]
confirmed
Exit
Failed to update
do/display error
Successful Update
do/display successful update message

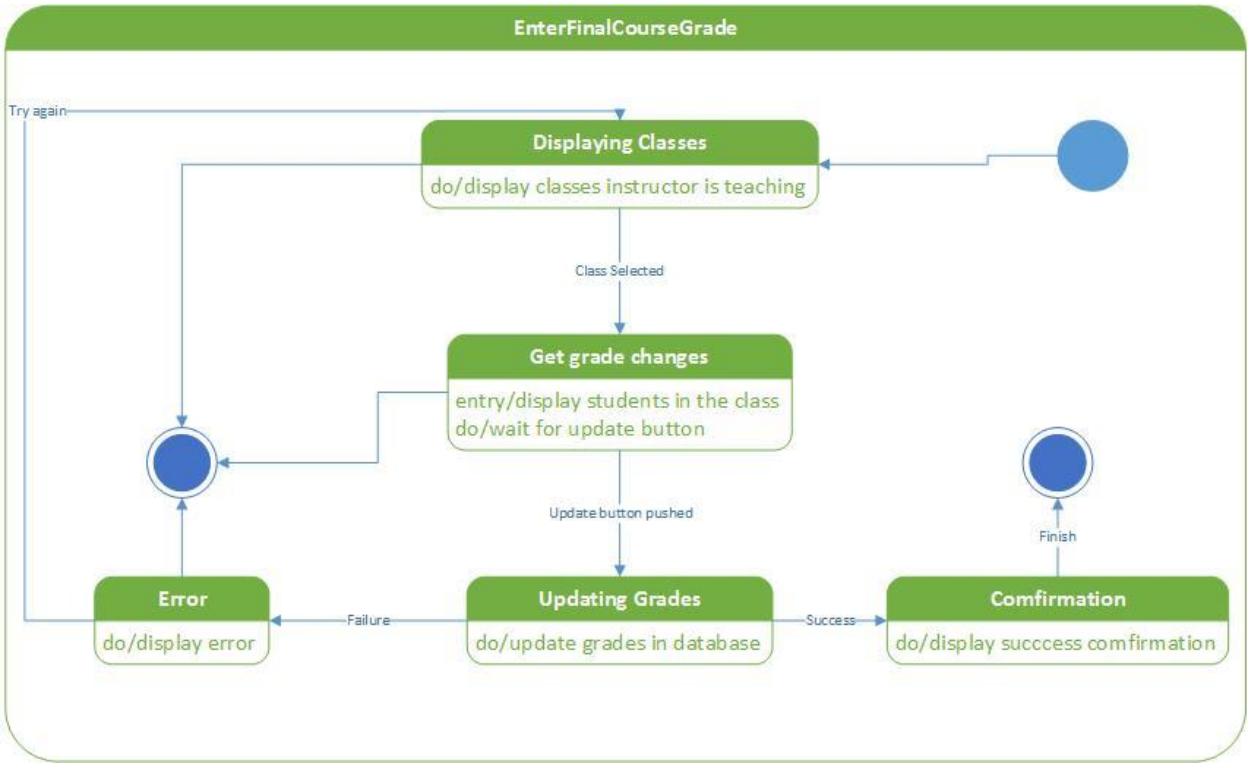

RemoveCourse
Getting term
entry/display terms page
do/wait for term to be selected
exit
Term selected
Getting course
entry/display courses page
do/wait for course to be selected
exit
Course selected
exit
Displaying course info
do/display course info
finish
exit
Course deletion button hit
Confirmation
do/display course deleted comfirmation
Course Deletion Fail
do/display error
Failure
Deleting Course
do/delete course
Success

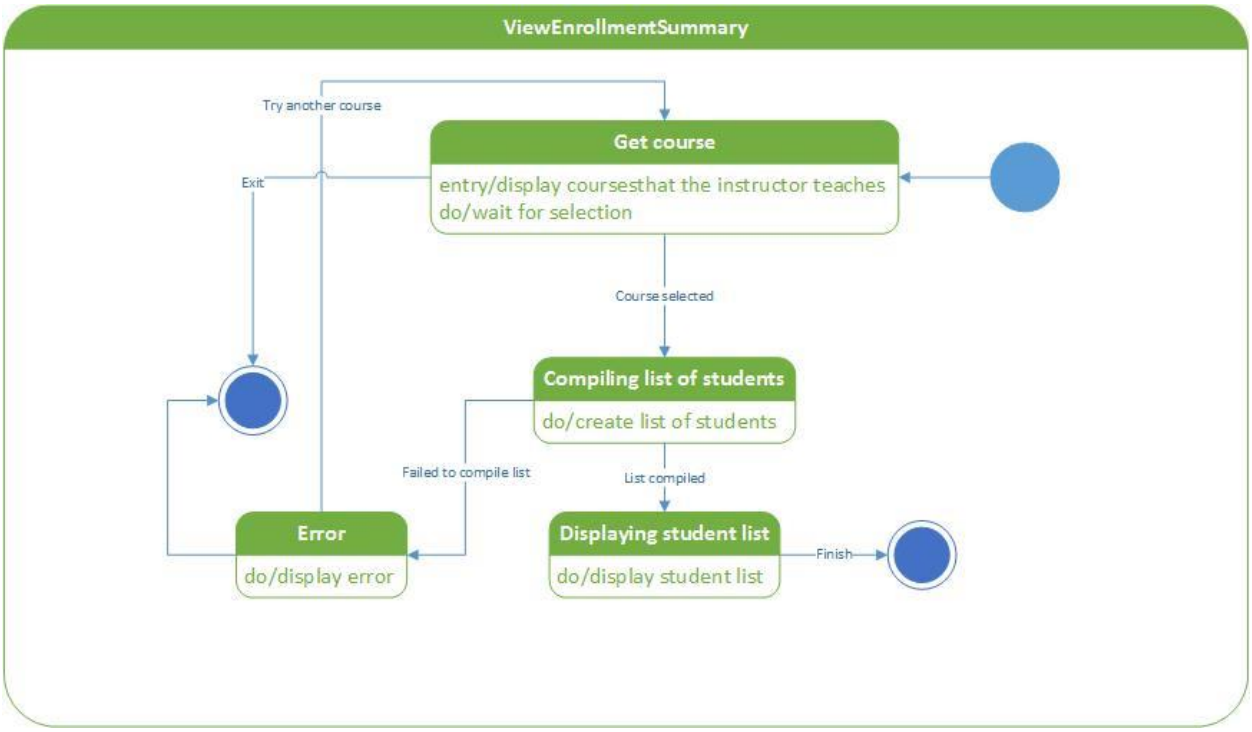

RemoveUser
Get user to remove
entry/display remove user page
do/ query user to remove
Exit
User entered
User not in system
Check if user exist
do/check for user in system
User exist
Exit
Update Student List
do/update the student list
Finish
Failed to delete user
do/display error message
Update finished
Successful student deletion
do/display successful message
Confirm deletion of user
do/get confirmation
Confirmed[failed]
confirmed


AddUsers
Getting username
do/get username from input
Try again
Username entered
Failure to create
do/display error
exit
Getting password
do/get password from input
Exit
Could not create user
exit
Password entered
Getting user id
do/get user id
Id entered
Creating new user
do/wait for creation to finish
Update sucess
Success Confirmation
do/display success confimation
exit

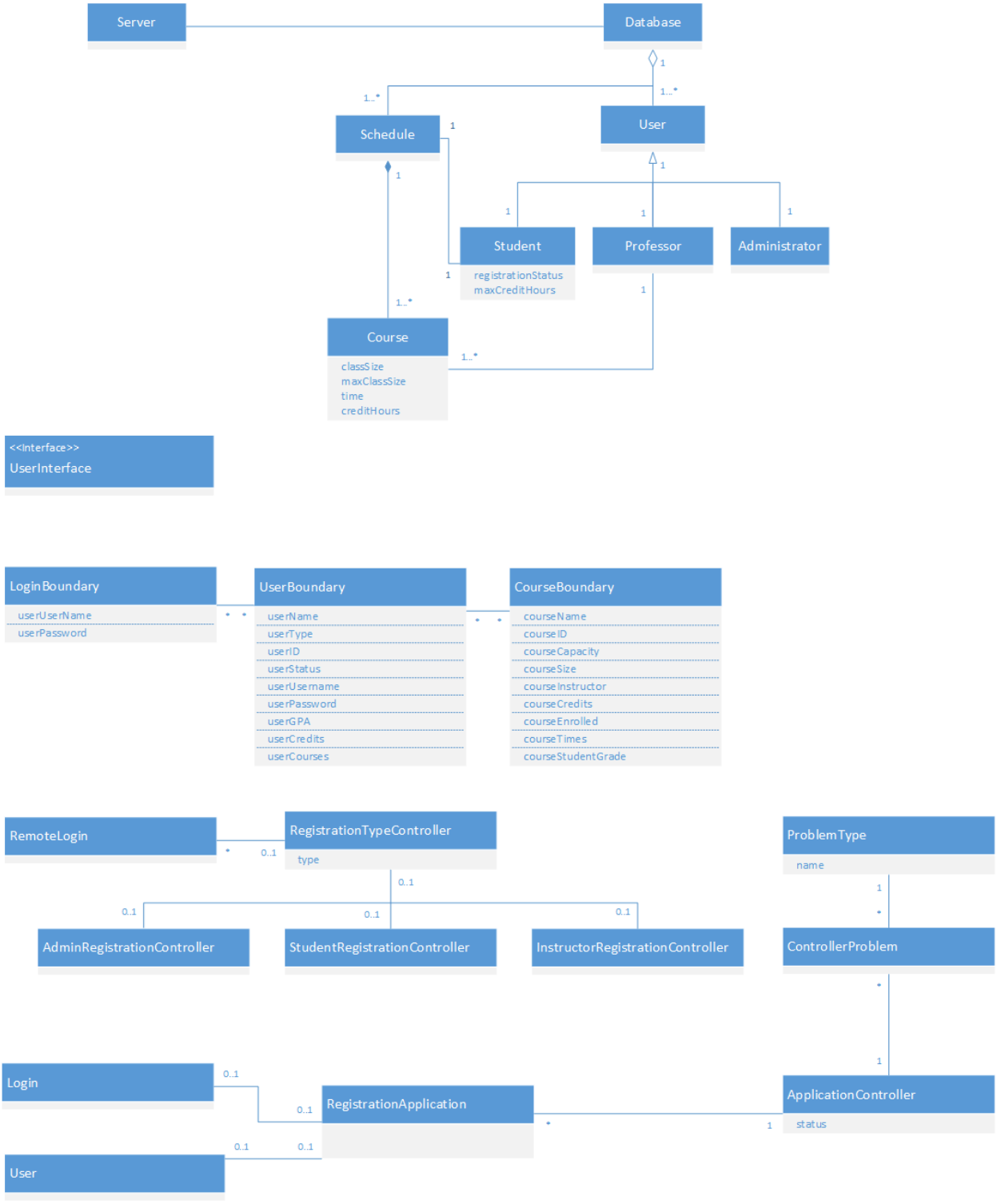

IntructorRegistrationController
Action Completed
logout
Instructor main screen
do/display instructor screen
Exited
Action Selected
Instructor actions
ViewSchedule
EnterFinalCourseGrade
ViewEnrollmentSummary


ViewSchedule
Try again
Requesting schedule
do/request schedule
Schedule could not be gotten
Successful retrieval of schedule
Displaying schedule
do/display schedule
Failed to get schedule
do/display error message
Finish
exit


EnterFinalCourseGrade
Try again
Displaying Classes
do/display classes instructor is teaching
Class Selected
Get grade changes
entry/display students in the class
do/wait for update button
Update button pushed
Finish
Error
do/display error
Failure
Updating Grades
do/update grades in database
Success
Comfirmation
do/display succcess comfirmation


ViewEnrollmentSummary
Try another course
Get course
entry/display coursesthat the instructor teaches
do/wait for selection
Exit
Course selected
Compiling list of students
do/create list of students
Failed to compile list
List compiled
Error
do/display error
Displaying student list
do/display student list
Finish

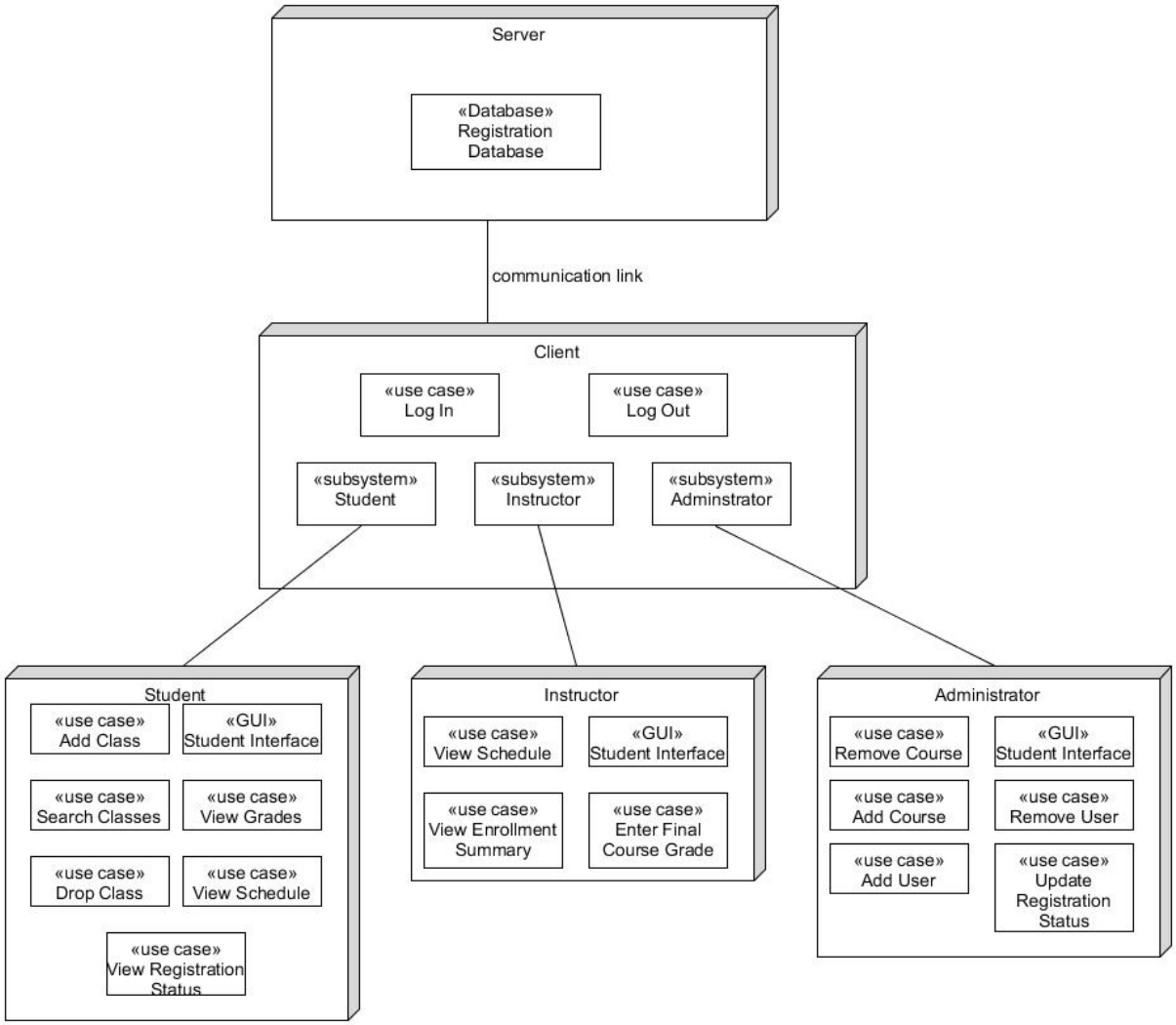
Consolidated Class Model



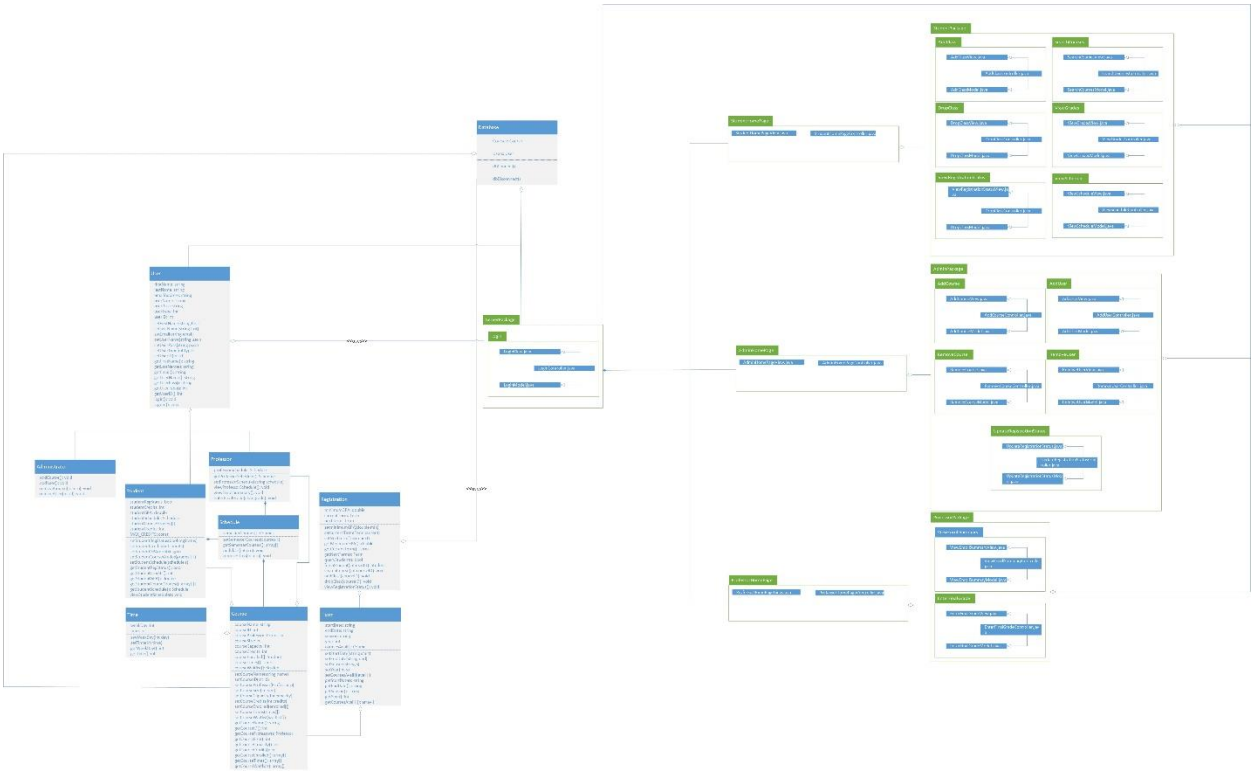
Architectural Design

For the architectural design of the registration system, the team has elected to model the project using a client/server style of architecture. This is mainly because the final design of the project is intended to be web-based. Client/Server would be necessary, or at least a good fit, since the system will heavily rely on networks. Also, with a client/server architecture some operations can be performed on a server and information can be stored on a database on the server. Some of the pros for using this style of architecture include an increased ease of encapsulation and this style makes it increasingly easier to use network operation since some of the operations can be done on the server. This style should make the distribution of data straightforward and it will be easy to add or upgrade servers if the system grows. However, there are also some cons of being dependent on servers. For example, there will be some redundant management in each server and there will be no shared data model which could make data interchange inefficient. Overall, the client/server style seems to be the most appropriate for a registration system that depends on a networked system and database.

Deployment Diagram



Design Class Diagram



[Link to full]

Object Design

A. AddClassModel

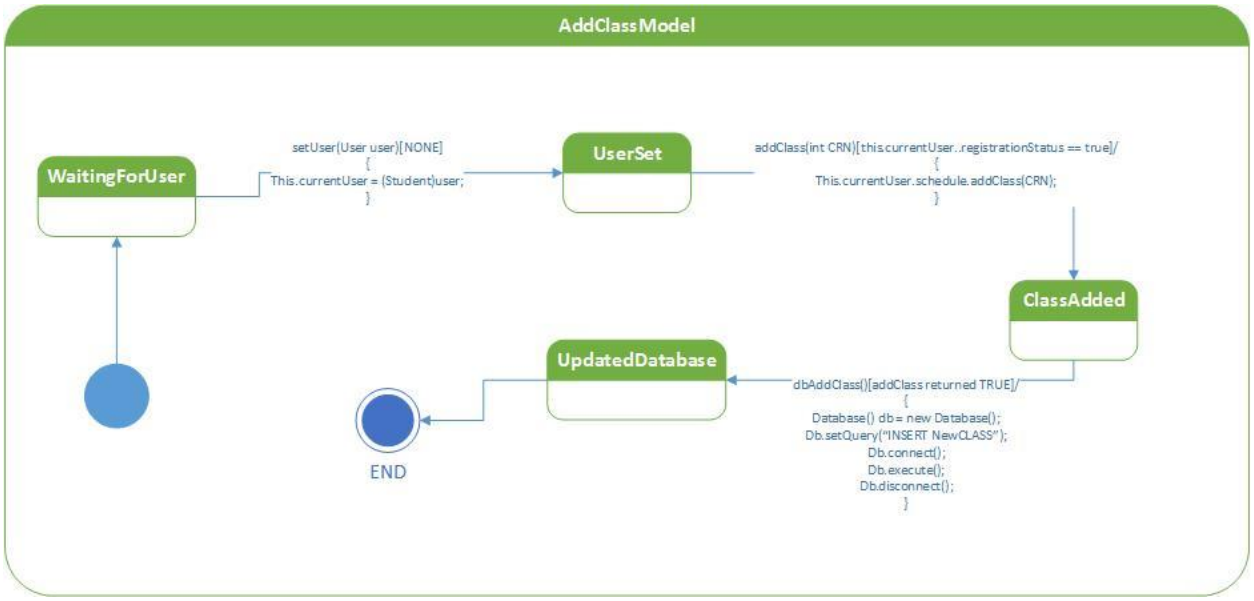
1.

Invariant: User must exist in the database.

Precondition: User.registrationStatus == TRUE

Post condition: Class is added to that user's schedule.

2. Class State Model



3.

```
addClass(int crn) {
        if(currentUser.registrationStatus == true){
                if(currentUser.schedule.addClass(crn)){
                dbAddClassUpdate();
                }
        }
        Else{
        DisplayError();
        }
}
```

B. Schedule

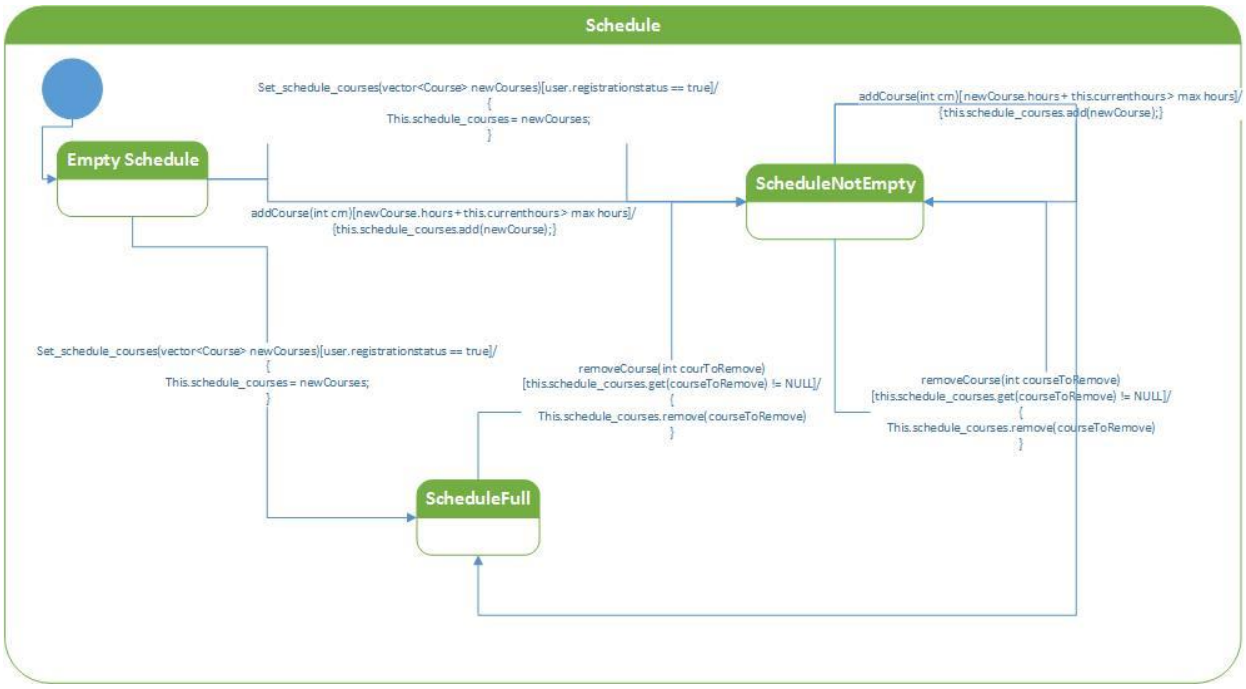
1.

Invariant: Only classes of type: Course are stored in a schedule.

Precondition: Courses must be modified via CRN number, which is stored in each course.

Postcondition: Schedule is modified with respect to its function call name.

2. Class State Model



3.

```
addCourse(int crn){
        Course courseToAdd = crnToCourse(crn);
        If((courseToAdd.hours + currentHours) < MAXHOURS){
                Schedule_course.add();
        }
        Else{
                displayError();
        }
}
```

C. LoginModel

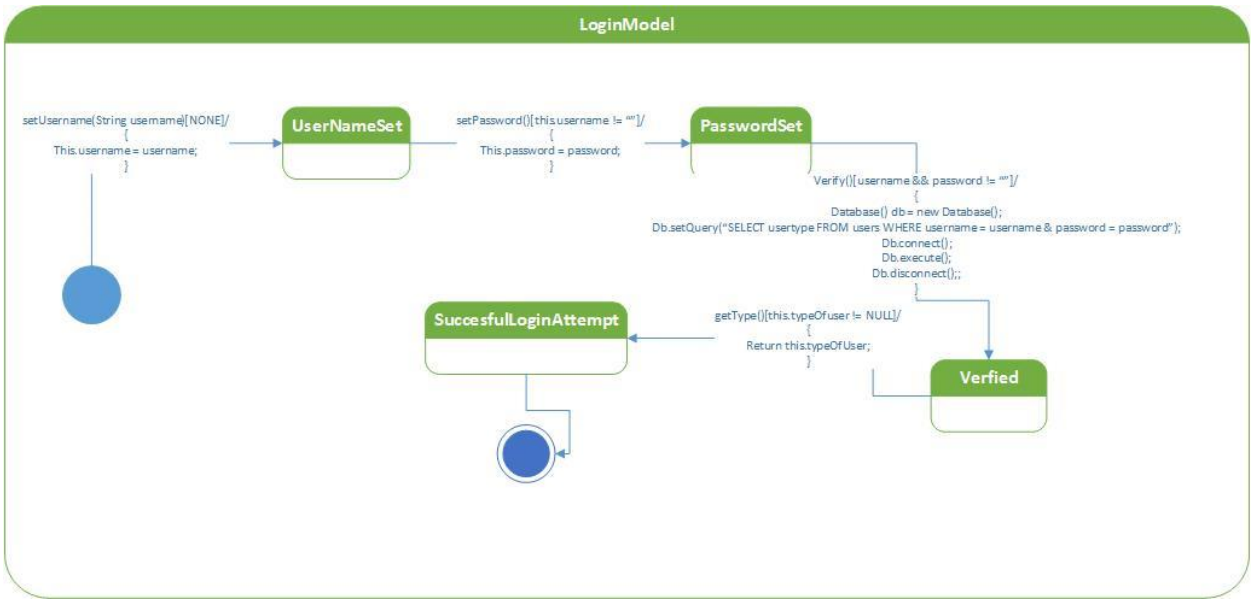
1.

Invariant: Must have a database with a Table of users, containing usernames and passwords.

Precondition: Username and password variables must be set before calling verify.

PostCondition: Type of user is returned after verify.

2. Class State Model



3.

```
Verify(){
        If((this.username && this.password != “”)){
                Database db = new Database();
                Db.setQuery(“SELECT type FROM users WHERE username=this.username &
                password=this.password”));
               Db.connect();
                result = db.executeQuery();
                this.typeOfUser = result.next();
                db.disconnect();
        }
}
```

D. DropClassModel

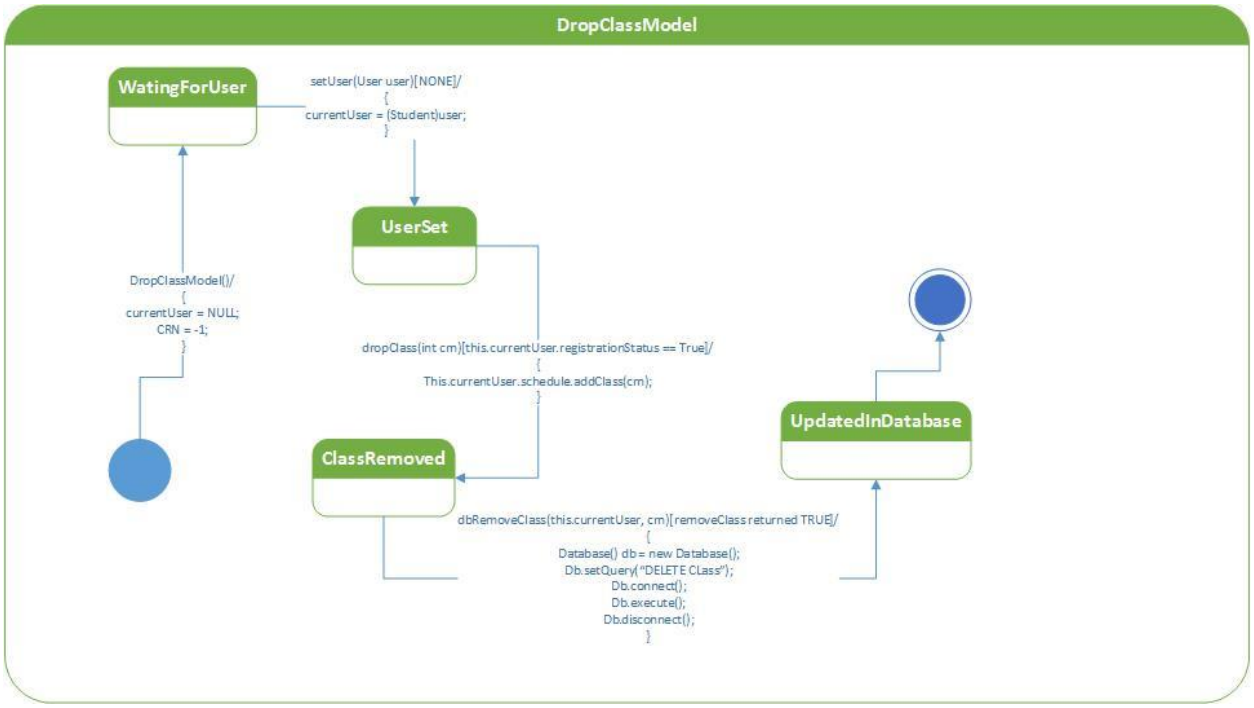
1.

Invariant: User must exist in database.

Precondition: user.registrationStatus != false;

PostCondition: Class is removed from students schedule;

2. Class State Model



3.

```
dropClass(){
        if(user.registrationStatus == TRUE){
                this.currentUser.schedule.dropClass(crn);
                dbDropUpdate();
        }
        Else{
                displayError();
        }
}
```

E. Database

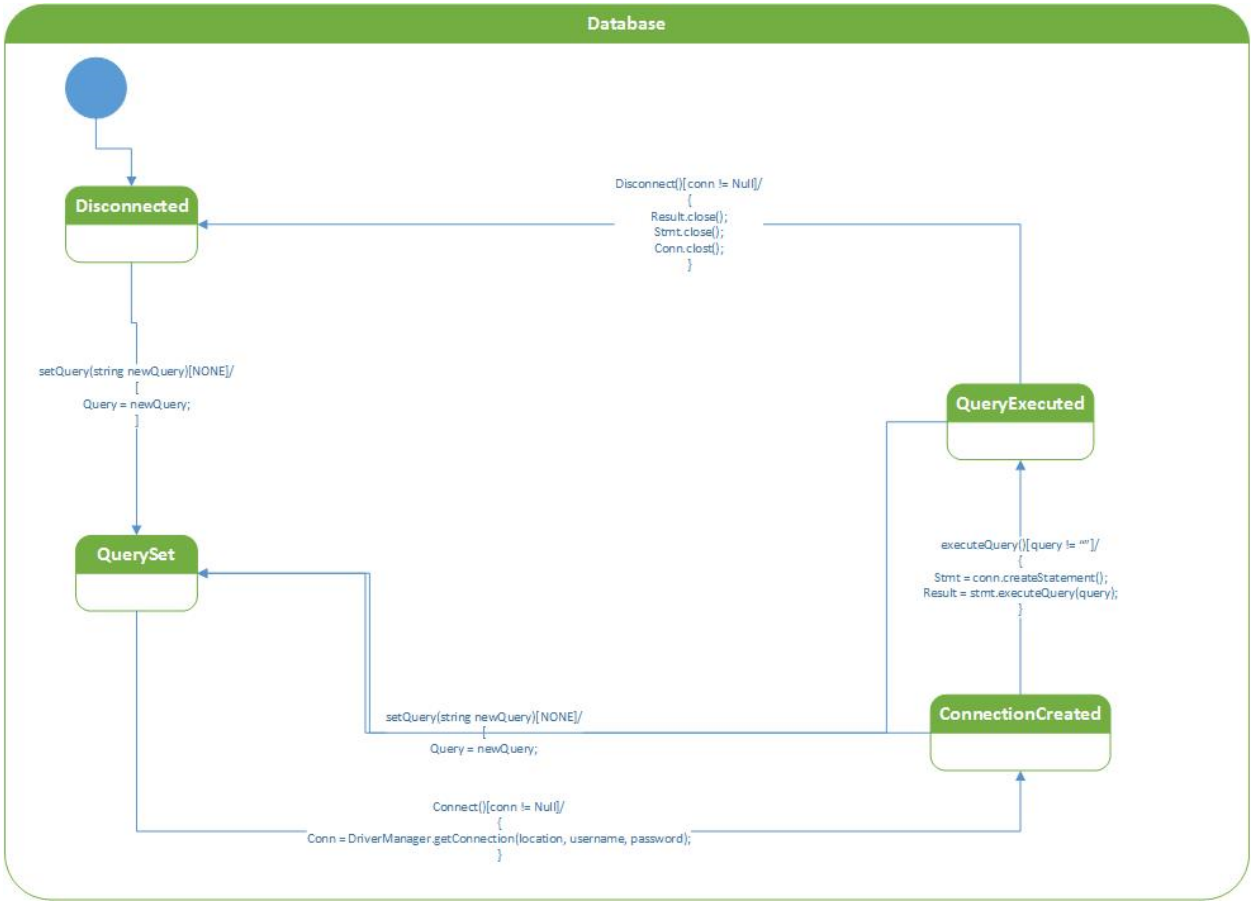
1.

Invariant: Connection to a database must be hard coded(host, username, password).

Precondition: Database uses prepared statements.

Postcondition: Should query the database for information and have ReturnedSet set.

2.



3.

```
Disconnect(){
        If(result != null){
                Result.close();
        }
        Elseif(set != null){
                Set.close();
        }
        Elseif(conn != null){
                Conn.close();
        }
}
```

Model Review

Original Analysis Review

With the analysis section of the project complete, we can now review the work we have done with the intention of making sure it holds true to completeness, correctness, and consistency. This must hold true not only within diagrams, but across the multiple diagrams completed. The Domain Analysis will first be addressed, followed by the Application Analysis. After this, an articulation of the strengths and weaknesses of the models against the criteria will be discussed.

Starting with the Domain Class Model, we have full completeness incorporating every important class that was derived during linguistic analysis. Each class gives functionality while holding true to the correctness criteria of the concept statement. Drawn from the Domain Class Model in a very abstracted view, every student has a schedule that is stored in a database, and our software centers around manipulating this schedule. We next see how this manipulation will take place with the Domain State Model. Consistency occurs from our Class Model to our State Model. A “user” logs into the system, he/she can then manipulate their “schedule”, which has a number of “courses” on it. Our Domain Analysis seems to hold true in completeness, correctness, and consistency.

The Application Analysis was surely a larger part of the analysis phase, and we did our best to maintain completeness, correctness, and consistency within and across all diagrams. Application Interaction Modeling was done successfully, as all requirements were covered, none of the models seem to be done wrong, and each solution is consistent with the other solutions. The Interaction Models account for all the functionally requested of the software. The Application Class Model is more detailed, giving us a stronger idea of functionality for each class. In the Application Class Model, we see that the relations, classes, and attributes allow us to meet all the requirements of the project. With accordance to the way we are implementing the project nothing seems to be wrong, every class has been successfully made and related to create a correct solution. The Application State Model does a good job showing the flow of states a user will go through in the software. Every state that a user will be in is modeled correctly. The State Model addresses every state that is possible, and is very consistent with the previous Application Analysis Models. When taking a look at all of the Modeling Diagrams together, we see that not only do they hold true to being correct and complete, that they are consistent with each other.

When discussing our strengths and weaknesses of our models, we see that our DAM and AAM model the criteria very smoothly. They are easy to understand and when we go into the design phase, it

should be a smooth transition. Every model seems to address the criteria in a correct way. With them being easy to understand, a weakness with some of our models may be that they are too abstract. This might result in more thinking during the design phase, but it shouldn't be a large problem. With regards to our consolidated class model, it also may be a weakness, as it may not be fully complete, correct, and consistent. This was foreseen, as our group didn't have a strong understanding on constructing this model. Concluding this review, we definitely have some weaknesses that we are going to have to work through, but with the strengths we also have, we should be able to move forward into the design phase fully prepared.

Design Evaluation

For the design section of the project, we took what we had completed from the analysis section of the project and further refined those ideas and diagrams to develop more concrete design diagrams and information to represent a detailed design model of the student registration system. We will now review the work we have done with the intention of making sure it holds true to completeness, correctness, and consistency. Consistency was the most difficult for our group as it must hold true not only within diagrams, but across the multiple diagrams completed. In order to give a complete evaluation of our design, we will walk through the report elements in sequential order as it is laid out in our report.

First, we will begin to look over the use cases. For the most part, the completeness is in fairly good shape as far as the covering the build of the prototype that we have created. However, if given the chance to produce a fully working system the completeness would fall short of expectations given that the current scenarios handled by the system would need to be greatly expanded to cover many more possible outcomes. From a consistency standpoint, the use cases are fairly well setup to handle scrutiny and are fairly consistent within the use cases. However, due to design changes during the prototype build there are a few issues that can be found, especially with naming and some minor inconsistencies in correspondence to the design class diagram and object design section of the report. Each use case will be address specifically in the following paragraphs as to determine its completeness, consistency, and design quality.

To start off, we will look at the Add Class, Search Classes, Remove Course, and View Grades use cases. In each of these four use cases, one of the major forms of inconsistency was its precondition. They all start assuming the user is logged in as the correct type of user and the diagrams do not show this login process. Also, the diagrams do not all show how the main controller for each use is

instantiated. The controller should be named the same as the use case and should be created by the homepage controller when the user selects an action from the home page menu. Aside from this fact, the Add Class and Search Class use case are designed correctly with quality in mind and complete in terms for the main or correct usage scenario, but more coverage of alternate scenario should be explored before moving past the prototype stage in order to truly be considered complete. The same can be said for both the Remove Course and View Grades use cases, however, the DSSD and collaboration diagrams for these were changed a little later in the process to account for the use case controller being create upon selection of an action from the homepage action menu. This may have cause some small inconsistencies within these individual use cases. Also, a term object is being used in these use cases to manage data, but this idea was not communicated well to the programmers and rest of the team, so it will not be represented as a full on object in the class diagram or final design of the project as it is not necessary to represent this piece of data as an object.

Now, we will address the Drop Class, Update Registration Status, Remove Users, and View Registration Status use cases. First, in Drop Class, the lack of a "button" object in the DSSD and collaboration diagram is the main issue. There were discussions in both directions during collaboration of coding design and apparent miscommunication led to these objects being omitted from the design. Next, in the Update Registration Status use case, there is one glaring issue from a consistency standpoint. The DSSD and collaboration diagram list the controller for this use case as "Registration Status", however in the final design of the project the controller was renamed to reflect the use case itself (UpdateRegistrationStatus). Third, in the Remove Users use case, the overall build of this use case is complete and consistent for the prototype that is being designed. Since the use case is very simple and straight forward with little option for variance, the prototype does cover the majority of possible outcomes. Finally, when examining the View Registration Status use case, we again see that from a consistency and completeness standpoint most everything about this use case has been covered. For all of these use cases, the design quality of the diagrams was fairly good. A few of the numberings within the collaboration diagrams were mislabeled. The steps themselves are in the correct order, but a few of the numbers were out of order (missing 7.2 on a couple of diagrams).

Next, we will look at the Log In, View Schedule, and Add User use cases. First off, in the use case Log In**,** there were some consistency issues to note in the transfer from the DSSD to the Collaboration Diagram. These issues are as follows: the classes UserInstance and ValidUser are completely left out of the CD, there is a name difference when transferring the Homepage Controller to the CD, and the obtaining of user information is condensed into a single method. Second, in the use case View Schedule,

there was some ambiguity in one of the method names (sendRequest()). This could be solved if we changed the method name of this and others like it to names that better represented the functionality of the particular method (e.g. sendScheduleRequest()). Lastly, in the use case Add Users, there were several issues. One to note is the consistency issues existing between the DSSD and collaboration diagrams. The class UpdatedUserList is completely removed and replaced with a method call. It is also important to note that, on these CDs, there were no boundary objects diagramed. However, aside from these inconsistencies, we believe the over design quality of these use cases is still high and complete enough to create a working prototype that can cover the scenarios covered by the diagrams.

Moving on to the Log Out, Add Final Course Grades, Add Courses, and View Enrollment Summary use cases, we have some of the same concerns found in the previously described use cases. This includes inconsistencies concerning the precondition of logging in, the lack of the "button" object's representation, and some processes (usually previously described in other use cases) not fully being fleshed out in terms of completeness. This can be seen in the Add Final Course Grade use case, where the user must go through a series of steps, including processes similar to the View Schedule use case, in order to get to the main function of the user actually adding new grades before sending them to be updated in the database. Another concern that we addressed is lack of knowledge of how *specifically* each controller is going to interact with one another as well as the database. This can be better understood by looking at the Log Out use case, where a request and confirmation to log out is known, but other processes that must take place before the user being logged out are not fully known, such as the method of saving the user instance. This concern may be applied to the other use cases as well, as complete knowledge of the system may only be realized during or after the stage of implementation. This concern can even be seen in the most complete use case out of these four, View Enrollment Summary. While the overall process is understood, the specific data pulled from the database and the process to organize this data into the Enrollment Summary object cannot be fully recognized from the use case diagrams. A final concern that should be mentioned is the difference between the Add Class and Add Course use cases. It is worth mentioning that the Add Class use case is for users of type Student or Professor, while the Add Course use case is for Admins only and handles adding a new course to the database. The Add Course use case mirrors much of the functionality of the Add User use case, and the Add Course use case may not be fully represented in its respected diagrams.

Summing up our use case evaluations, the DSSD and collaboration diagrams cover the positive outcomes of the use case. If anything is needed to make the design of this case stronger, the addition of diagrams showing the handling of negative outcomes would need to be added as well. Completeness

would need to be fleshed out a little more for a fully working system. For the most part, the correctness and consistency within the use cases was overall relatively high. The biggest issue that we faced came in the form of consistency between diagrams and naming of objects. Overall, the names and details are close between different parts of our design but may not match up 100% when looked at critically; however, the diagrams should still be easy to understand and still lead to a strong design. Now that we have covered the use cases, we will look at the design class diagram and object design.

The design class diagram was composed from the conceptual design the group constructed during Phase 1; turning the conceptual design into a more concrete design while following a client-server design pattern. During the process of translating the conceptual design into a more concrete design, errors and inconsistencies between the two designs may have occurred. Specifically, some of the class attributes have different but still very similar names, some classes have more attributes in the concrete design than the conceptual design, some classes may have been removed (e.g. server), and last but not least, the conceptual design has very little detail towards the package system that was implemented into the concrete design. There also may be some inconsistencies between the collaboration diagrams and the design class diagram. There are some slight semantic differences such as the various controllers having different, but extremely similar names. However, overall, the consistency of the design class diagram is consistent with the use case collaborations that it was built upon. In terms of correctness, there may have been other more efficient/correct ways to implement the storage of certain attributes, but it should still definitely work the way we have it set up. As for the completeness of the design class diagram, we feel like we have included everything that is necessary.

With regard to the object design, we feel as a team that it went fairly well. The biggest problem was deciding on classes with states. Since we are doing a registration system, most of our classes function more as a structure, than a class with state-dependent behavior. Our object designer did his best to choose classes that had some form of state-dependent behavior. All of the Object Designs seem to be for the most part correct and complete; however, there may be some consistency issues with the names of methods or member variables not completely matching those from the Design Class Diagram. The names express the same thing, but may have subtle differences. The Database Object Design (E.) also features some Java database connection imports that may not be in the DCD, but were needed in order to show the state of the Database class. Finally, the methods for the Model Classes (A., C., D.) were not all defined in the DCD. The Object Design goes ahead and defines needed methods so its class can fulfill its role. Other than the previously noted inconsistencies, our Object Designs do a fairly good job maintaining completeness, consistency, and correctness.

In conclusion, if the system were fully fleshed out to a working product, there would need to be more possible scenarios covered and thought out before the design could truly be considered complete. For the purposes of this prototype, however, given the constraints that we were working with, the system is mostly complete. Overall, the biggest issue we faced came in the form of consistency, especially in terms of naming, due in large part to the time constraints. For example, some parts use Professor while others use Instructor. This is the same with class vs. course. However, even though there are some inconsistency between some models, overall, the project design was not only correct but designed with quality in mind and should be relatively easy to follow. We believe this design will lead to a strong product that will function well.