Grade Achiever - Design Report

SENG 350 - Software Architecture
SEAR Software - Group 3 - 4

Grade Achiever Detailed Design

This is a detailed design and Architecture Decision Record of “Grade Achiever” developed by SEAR Software. This app’s purpose is to provide students with recommendations on how much they should study for each gradable item in order to achieve their desired grade, while also offering functionality to help them keep track of their schedule.

Features

Grade Achiever is a tool for students to use during their semesters at university, or college. It allows users to create courses and add each assignment, midterm, test, etc. they must complete for that course as what we call a “gradable item”. They also set a desired grade goal for each course and the app takes that information, along with how much each gradable item is worth, the difficulty of the course, and the previous results of the student, to recommend a study time for all gradable items.

The web app will include features that allow users to

- Submit course details by uploading a course outline (PDF).
- List gradable item deadlines in order.
- View ideal study time based off previous study time and grades.
- Receive email notifications when deadline is approaching based on weight of assignment/midterm/final.
- Configure notification settings.
- Add/remove gradable items/courses.
- Change/enter course grade goal.
- View overall current semester grade.
- Add a difficulty level to a course that is factored in to suggested study time.

Architecture Diagrams

The sequence diagram in Figure 1 shows the standard behaviour and structure of the web app during runtime. It follows a basic happy path where a user logs in to view the homepage:

1. User is selected from log-in selection
2. SessionController verifies that user is in database and sends back a response json with the details of the user
3. SessionController contacts overview controller to load up the overview page
4. Overview controller loads up details for each of the courses under user details by using the course controller
5. The course controller contacts the gradable item controller to get details for each gradable item
6. The overview controller receives this information from the course controller and populates the overview page
7. User sees loaded view template filled with information from overview controller

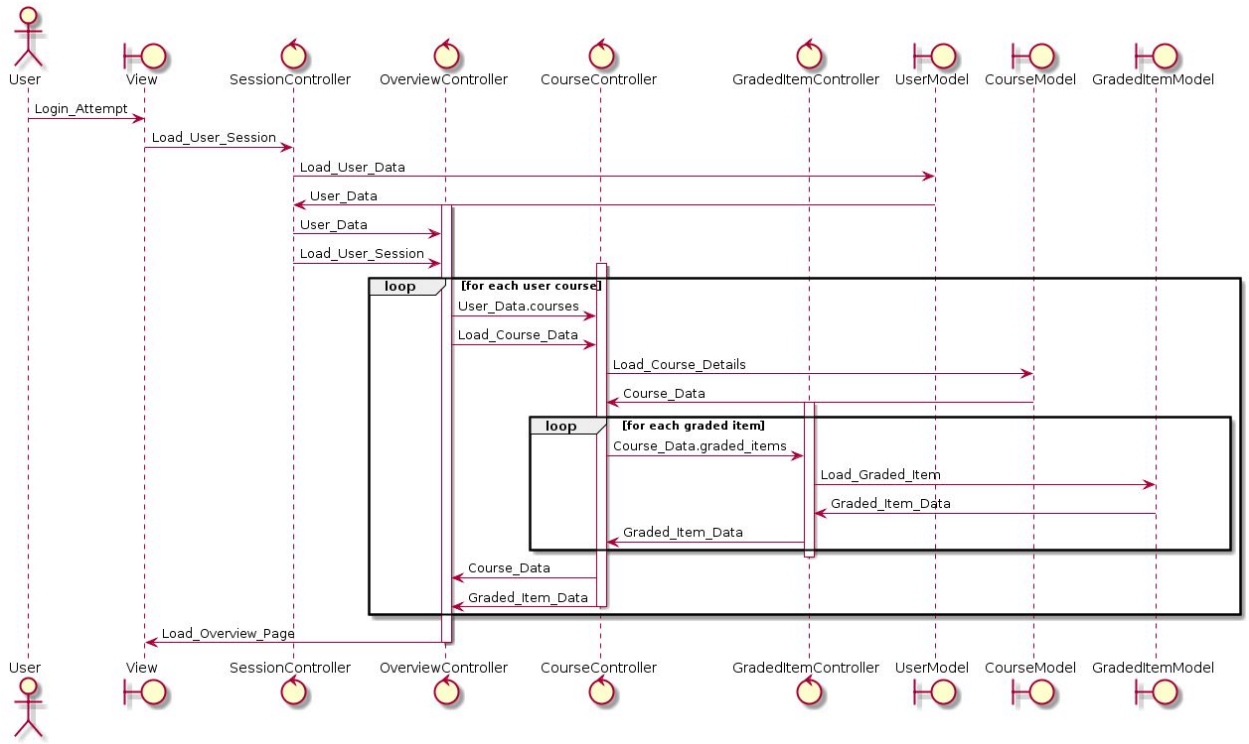

Figure 1: Behaviour diagram of a user login to view their homepage

The module diagram of the web app showing the general structure of the app is shown in Figure 2. This diagram was done in the Module-View-Controller style to capture the modularity of the system. It has been designed to ensure single responsibility for all of the functions and modules we design, hence the multiple controllers. As shown in the sequence diagram above, the controllers talk to each other to build their classes and package the data for the views. They each individually talk to their respective models and database tables to gather up the data they care about before sending it off, either to a view to be rendered for the user, or to another controller to be further packaged. This design also benefits the testability of our app, as each controller can be tested individually with manually provided data during testing.

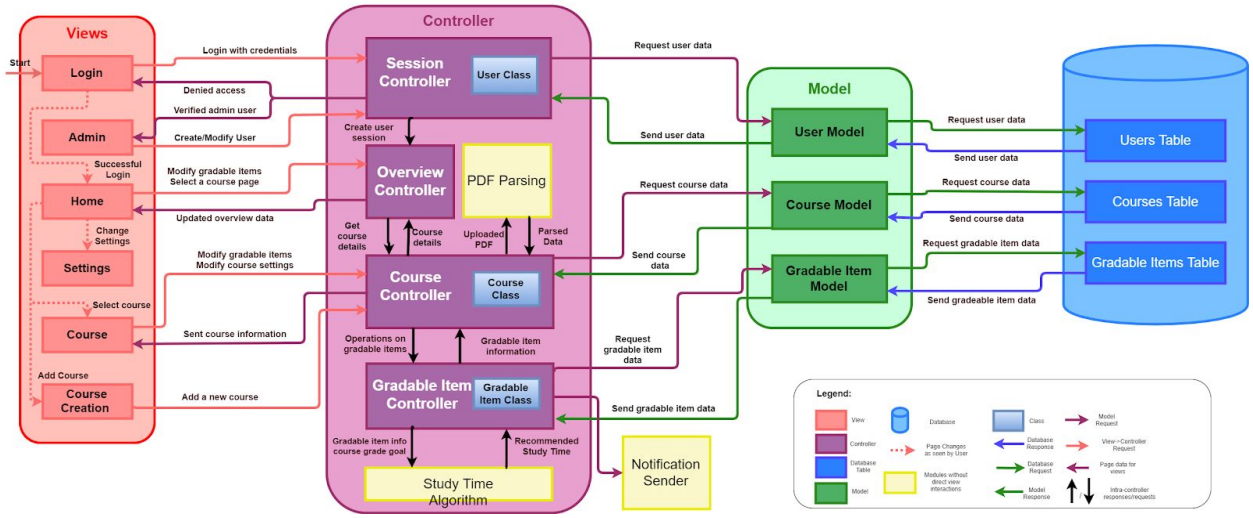

Figure 2: Module diagram of system in MVC style

Architecture Decision Record

Study Hours Algorithm

Context: We want to have an automatic way for our application to recommend a number of study hours to each user based on the amount they have studied in order to help them achieve their target grade.
Decision: The study hours algorithm calculates the ideal study time for a user based off previous study times and grades received. This is calculated based off of a standard equation that is multiplied by a factor to better fit a student (eg. 1.3 * 10hrs, where 10hrs is the base output, and 1.3 is the scaling factor based on previous data showing this student usually requires more time to gain the recommended grade). Having this additional factor specialized to

each student allows this algorithm to adapt to each user. The exact function to calculate the scaling factor and base recommended hours have not been fully determined.
Status: Early planning stages. The algorithm will first have to be designed and tested using an initial batch of user data then refined to better predict needed study hours.
Assumptions: The amount of time a user spends studying/working on a gradable item has a causal relationship with the grade the user receives.
Consequences:
The initial implementation of the study hours algorithm, while able to adapt to input from a single user, will not change its global recommendations based on recurring patterns.

One algorithm design idea:
Variables:

- Grade goal
- % weight of item
- Due date
- Perceived difficulty level (higher weight at beginning of semester)
- Previous gradable item grades (higher weight as more previous grades are enter)

Functions:

Recommended gradable item grade
This function determines the grade needed from a gradable item in order to maintain the grade goal.
Input: current course grade, grade goal, % weight of gradable item
Output: recommended grade for gradable item

Weight factor
This function returns a scaling factor that determines whether the student is told to study more or less for a gradable item
Input: recommended gradable item grade, recommended study time, actual grade, actual study time, (*or*) perceived difficulty
Output: weight factor (increase or decrease)

Recommended Study Time
This is the main function for the algorithm - it returns the total amount of recommended study hours for a gradable item
Input: recommended gradable item grade, due date modifier, weight factor, weight of gradable item
Output: recommended study time (hours)

$$Recommended\ Study\ Time\ (hrs) = grade * 10 * due\ date * weight\ factor * (1 + weight)$$

The dueDateModifier is a weight value based on how late in the semester is (assumption is that gradable items get harder the later it is in the semester).

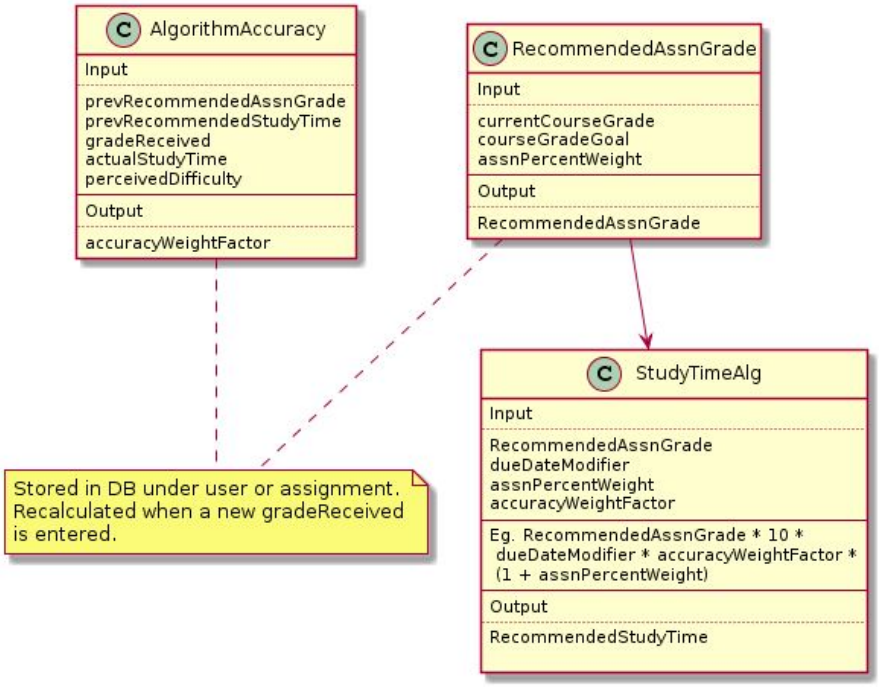

Figure 3: Class diagram of one potential algorithm implementation

Related User Stories:

- User Story #6 Grade Goal
    - The study hours algorithm takes a grade goal as input and as output returns the number of hours the user should study.
- User Story #2 See Study Time Recommendation
    - The study hours algorithm produces a time recommendation for the user in hours.
- User Story # 1 Study Time Accuracy
    - The study hours algorithm is responsible for both taking in the time spent, time recommended and grade received on previous gradable items and ensuring that future study time calculations are more accurate.

Related ASRs

- Accuracy:
    - It is important that the study hours algorithm be accurate to retain users of the app. As the algorithm is being designed to know how much effort for a user translates into a grade and therefore the amount of additional effort is required for getting the desired grade, it should continually get more accurate.
- Modifiability:
    - In order to keep this app as modular and modifiable as possible to reduce future development costs, the algorithm will function as an independent entity. It shall interact only with the gradable items controller in the case of a new gradable item being added

- Performance:
    - The study hours algorithm is being designed with simplicity in mind - it will take several parameters and use a simple mathematical formula to calculate the recommended hours. Thus, high performance should be relatively easy to achieve for this algorithm

Course Items PDF-Parsing

Context: Users need to manually input their courses and all assignments or other gradable items including:

- gradable item names (eg. Midterm 1, Exam, Assignment 1)
- due dates (within semester, possibly listed as TBA)
- weight (% of final grade).

This puts a large time requirement on the user and involves a lot of entering values into fields.
Decision: The pdf-parse library https://www.npmjs.com/package/pdf-parse for TypeScript has been chosen to limit the user input requirement and to meet out QAS #1 (Data Input Simplicity and Efficiency). After the initial parsing, the user will also be given the ability to edit any gradable items. Having this option instead of requiring a user to manually input all gradable items will increase user satisfaction.
Status: Decided
Assumptions: This library will work and require minimal corrections from a user. Course outlines are usually handed out online as pdfs. A user would know how to upload a pdf file and convert a word doc to pdf if necessary.
Consequences: Users will have to confirm that the parsing library correctly inputs the gradable items and their weights, due dates. External library will need to stay up-to-date. Extensive testing and cases will need to be developed for differences in course outlines.
Related User Stories:

- As a user, I want to be able to create and edit gradable items, so that I can see how much time I need to study for it.
    - The PDF parser will be responsible for automatically generating gradable items from each of the gradable items found in the course outline for a course.

Related ASRs:

- Modifiability:
    - In order to keep the app as modifiable as possible, the PDF Parser will be a separate entity within the app. This should minimize coupling with other modules and thus increase ease of modifiability.
- Usability:
    - This is the ASR primarily catered to by the PDF Parser. Allowing the user to simply submit a PDF instead of manually entering course details greatly reduces the amount of time users need to spend to achieve desired goals and therefore greatly increases usability.

Notification Sender

Context: A controller is needed to manage sending users notifications at intervals that would be useful. The controller can either be spawned by a cron-job at arbitrary intervals, or be an always active process that sends a notification at any intervals as set by the user. The notifications can also be sent to the user in a variety of ways.
Decision: The notification sender will send out scheduled email notifications to users based on their desired frequency and content. This will be done by adding one-time entries to the cron file every time a gradable item record is created for when it needs to notify a user.
Status: Decided
Consequences: The notification sender will need to have the functionality to send an email. Creating one time entries in the cron file can also add additional complexity if notification settings are modified.

Related user stories:

- User Story #3 Edit notification settings
    - The notification controller is responsible for ensuring each user only receives a notifications when desired. When the user modifies settings, the gradable item controller must modify cron entries for the user to only spawn the notification sender according to the user's notification settings.

Related ASRs:

- Customizability:
    - The app should not deter students if they feel they are receiving too few or too many notifications, which is being solved by the addition of notification settings.

Database Design

Context: Information on users and user specific data, such as settings, courses and gradable items, needs to be stored and retrieved during each session. Thus, a database needs to be decided upon, as well as models for interfacing with the database.
Decision: MongoDB will be used as the database of this web app. This will allow storage of key value pairs, and as this web application is currently small scale, an approach with no SQL will not limit the use or speed. Additionally, three different models will be used to interface with the database: the user model, the course model, and the gradable item model. This design was decided upon to segregate data that was needed by the separate controllers as can be seen in the MVC Diagram. For example, when we modify only a single gradable item, instead of grabbing an entire user object, modifying it and storing it, we're doing it with much smaller objects individually. This allows the course object to remain small because it just has a small data structure which maps to other, larger, data structure.
Status: Decided

Consequences: Relations between entities in the database cannot be automatically enforced by the database and will instead require developers to manually enforce them.

Related user stories:

- User Story #7 View current grades
    - This information is stored inside the “Users” and “Courses” table respectively within the database and this requirement is thus facilitated by the database.
- User Story #6 Set Grade Goals
    - Thes desired grade goal for each course is also stored in the database in the courses table.
- User Story #5 Create/Delete users
    - The list of users is stored within the database, and will be able to be updated when the admin creates or deletes user accounts.
- User Story #3 Edit Notification settings
    - Each user’s notification settings will be stored within the “Users” table in the database and will be polled using the user model by the notification controller whenever it runs to see if a user needs to be notified
- User Story #2 Create/edit gradable item
    - Any information on gradable items needs to be stored within the database, and this information will be exchanged exclusively through the gradable items model and the gradable items controller
- User Story #1 Log time spent on gradable item
    - When a user logs time spent on a gradable item, the gradable item model updates the database with the time spent on that item.

Front-End Views

Context: The front-end views of the application need to show the full functionality of the web app, and be specific to each user (their courses/gradable items).
Decision: Pug, the template engine for Node.js, will be used. Pug will allow the different pages and views of the web app to be more uniform and cohesive. It will also allow for a simple template to fit all users. This makes the front-end design easily modifiable if new features need to be added, or other changes need to be made.
Status: pending
Consequences: Limiting the amount of front end views to 5 will result in far denser views. This helps decrease the amount of clicks required to navigate to a desired view, but may require the user to be provided a tutorial in order to fully understand the possibilities in each view.

Page Views: Five main views will be implemented to create the UI, with the homepage view providing almost all functionality.

- Login page
    - Select user

- Home screen
    - Courses
    - Next gradable items (schedule tracker)
    - Current semester GPA
    - Semester progress bar (or courses)
- Settings page
    - Notifications settings
        - Frequency for gradable items
        - Frequency for finals
    - Email
- Individual Course pages
    - Gradable Items specific to those courses (column based calendar)
    - Course goal
    - Current course grade
    - Course progress bar (eg. 25% completed)
- Admin Page
    - User account
        - Add/delete
- Course Creation Page (a modal on the home page)

Related user stories:

- User Story #5 Admin management
    - A separate admin page will allow an Admin user to manage other user accounts.
- User Story #4 Schedule list:
- The schedule tracker on the homepage will allow the user to see this.User Story # 3 Edit notification settings
    - Additionally, a view specific for changing settings for each user, allows the web app to be customizable and lets a user make adjustments to better meet their needs. These settings will be focused around changing notification preferences for gradable items.
- User Story #7 View current grades
    - The GPA and semester progress bars allows users to check their current gades and see how much farther in the semester they have to go.

Related ASRs:

- Customizability
    - Allowing personalization lets users use the app to its full benefits while changing it to meet their lifestyle.
- Usability
    - Limiting the amount of views should increase usability by reducing the amount of clicks required to perform any given action.

- Security

- As mentioned in QAS #7, all data transmitted between the client and server needs to be encrypted to ensure that user trust in the application is maintained. The front end will be designed to make sure all requests are encrypted before being transferred to the server-side controllers.

Test Plan

Context: In order for Grade Achiever to meet many of its QAS, it needs a comprehensive test plan to ensure all desired functionality is met. For both the integration of all of the various modules together and also the many functions that exist throughout the system, the app must be tested thoroughly throughout development and into production.
Decision: Testing of Grade Achiever will consist of automated unit testing, automated happy path integration testing, and automated QAS testing. Travis-CI, a linter, Postman, and manual user testing will also be used to manage the test suite and code. Automated unit testing will ensure that each function in the program works correctly. This will start as basic unit tests that are written during development, and then move towards a more complete set of tests for the later project milestones. This will ensure that improper inputs not covered by the happy path integration testing will not cause the app to crash and provide a sanity test when we add new features. Happy path integration testing will be very basic and ensure that when the system gets expected user behaviour, the app will react correctly and not throw any errors. A full unit testing suite and happy path integration testing will ensure our modules work and behave correctly within the system and the app can handle any external input from a user.

Automated QAS testing will test data input simplicity and efficiency and data safety. This will consist of tests that make sure all data inputs require less than five user actions and that all data sent between views and controllers is encrypted.

Travis-CI will be set up to manage builds and new commits to the web app repository. It will manage code coverage to meet the 75% requirement and automatically execute tests to guarantee that any code which reaches our GitHub repo does not break the application. A linter will be used in the form of ESlint for static code analysis.

Additionally, Postman will be used to test the web app's User, Course, and Gradable item models. Manual user testing will consist of checking the accuracy of the time estimate algorithm. The developers, and a limited study group, will use the time estimate algorithm and calculate whether the algorithm is accurate for them and how it can be improved.
Status: Pending. User testing in particular with the study algorithm app has not been confirmed.
Assumptions: Due to the time restrictions of this project, the time available to test this system is limited.

Consequences: Travis-CI can sometimes take a while to check whether a commit breaks the build. The study time algorithm will be mostly based off of estimates and user customization rather than data gathered on the average students study times.