# Study Planner Application

Nathan Francis - 100095913

Michael Meyne - 100070396

Phillip Perks - 100089021

CMP-5012B

University of East Anglia

## Introduction

In this project, we set out to implement a Study Planner application which we had already designed to a specification provided by the University of East Anglia. This document aims to enable any future developers to understand and evaluate the application's architecture, and subsequently configure and maintain it for future use and development. The documentation also contains UML diagrams to properly illustrate the internal structure of the program.

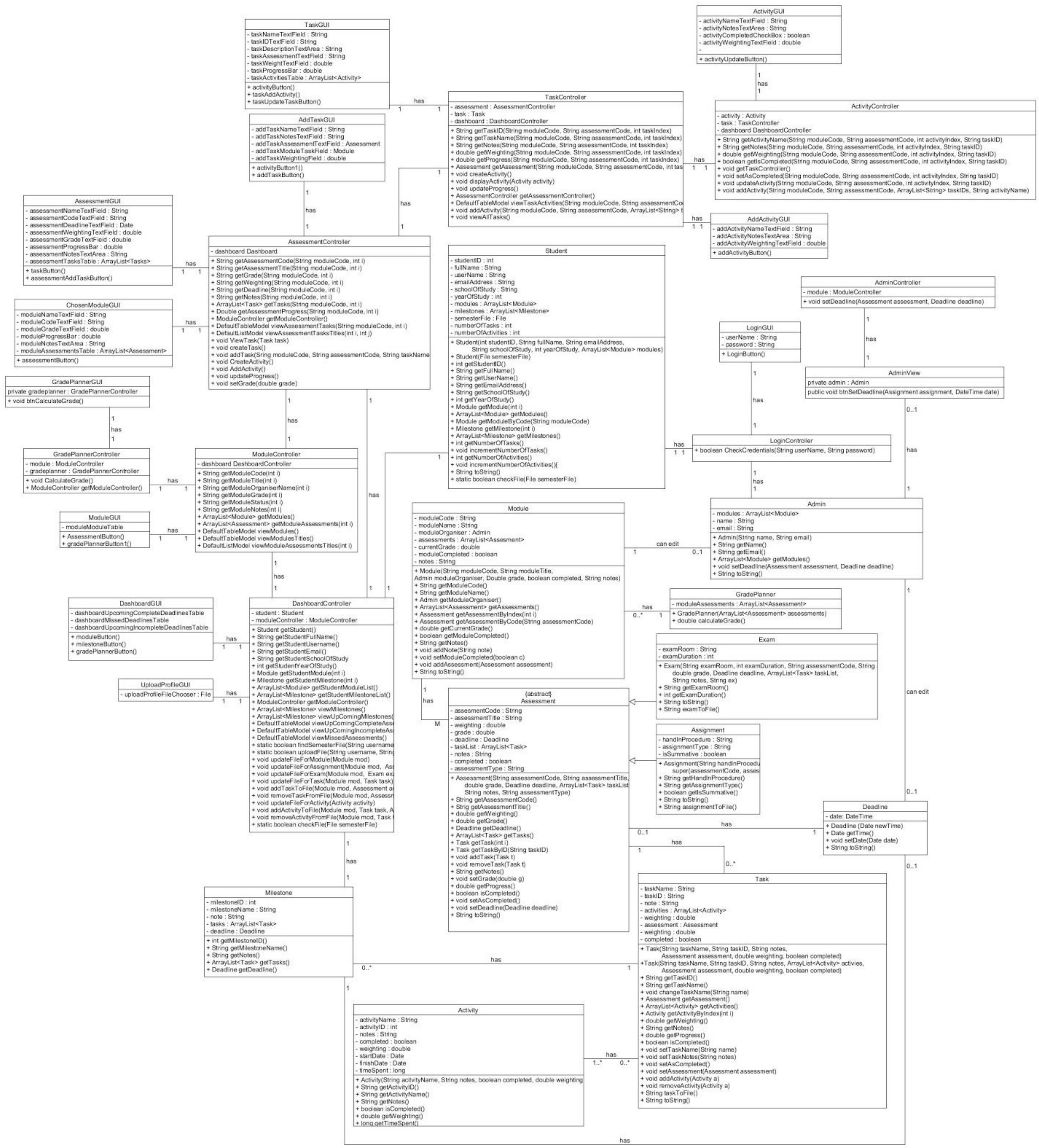
## Followed Method

We elected to continue using the SCRUM development methodology through the implementation stage of the application as this development approach served us well in the design phase, and completely changing our approach for the next phase in development would have caused unnecessary delay and complications.

Our team took on an iterative stratagem for the coding, as the initial problem was extremely complex and had many different parts to potentially conflict and cause issues further down the line, so breaking the program down into a number of iterations allowed us to keep abstraction within the program and ensure each component of the program worked by itself before its insertion into the broader application.

We allocated tasks mostly based on competence, and our members would usually be working on a task which corresponded to their personal coding skills, but the priority of each task was also definitely a factor when we were assigning tasks. Occasionally if a particularly crucial task was giving us problems, more than one team member would work on it to speed along the solution, but we avoided this when we could to ensure 3 different tasks were being done at any time.

## Detailed Class Diagram



## Program Outline

The Study Planner application enables a **Student** to plan their study time for university **Assignments** and **Exams** through planning and completing personal study **Activities** and **Tasks**. The **Student** can also view important details of **Modules**, **Assignments** and **Exams,** and track the **Deadlines** of each piece of work. All **Student, Module**, **Assignment** and **Exam** data is held in their semester file which is in the source directory of the project, which is loaded upon opening the application, and all **Activities** and **Tasks** are defined by the user of the program.

## Development Timeline

As we were employing SCRUM-based techniques, we agreed upon a timespan of 5 days for our first sprint, with our target being implementing the basic functionality and structure of the app, without the GUI or relevant view and controller classes. We began with developing the semester file, as we reasoned it would be difficult to test the hierarchy of the program without data already present in the system, and considering the semester file's position in the must-have section of the MoSCoW analysis, it was clear that designing and implementing the semester file was the first priority of the group. When the semester data was properly being loaded into the program, we began work on the classes that relied upon the semester data: **Student, Module, Assessment, Assignment** and **Exam.**

These classes form the backbone of the application, and once all these classes were instantiated with the relevant semester data, it became a lot easier for the team to process the overall structure of the program and recognize what tasks needed to be completed next. Work then began on the **Activity, Task,** and **Deadline** classes, which, while not being as important to the general program structure as the previously developed classes, provided important functionality for later class methods and also featured prominently in the MoSCoW analysis.

With these classes being completed, our first implementation stage was complete, and we reflected as a team on what the previous sprint had accomplished, and what we had left to complete in the time available. We evaluated what was left to do against the MoSCoW analysis, determined that we had the time to complete everything in time for the deadline, so we then prioritized the leftover tasks based on their reliance and dependencies on other tasks.

The View and Controller classes had been disregarded up to this point, so the next most important step was to create all the relevant view/GUI classes corresponding to each screen on the app we wanted to implement. These were **LoginGUI, ModuleGUI, MilestoneGUI, GradePlannerGUI, AssessmentGUI, TaskGUI, ActivityGUI, DashboardGUI, ChosenModuleGUI, UploadProfileGUI, AddTaskGUI** and **AddActivityGUI.** While it may appear this is a lot of classes to approach at once, most of the GUI classes were fairly trivial to implement compared to the other .java classes in the program.

Next in development came all of our data model classes, namely **ActivityController, AdminController, AssessmentController, DashboardController, GradePlannerController, LoginController, ModuleController** and **TaskController**. From this point onwards, all of the relevant classes had been created, and all that was left was to implement all leftover methods and eliminate any errors, which was no small task. We found it difficult at this point to use the same metrics to measure progress, as each task became smaller and thus more time consuming to inform each team member what we were working on, but after another day we reached a point where every team member was satisfied enough with our application to conclude development.

**Design Decisions**

Our first important design decision was which language to write the program in. We identified C++ and C# as potential candidates, but eventually decided to code entirely in Java because of our group's familiarity with the language, and the huge amount of online help available.

Our second major design decision came when we had to decide between using an existing file type for the semester file, or designing a bespoke new file system to handle the semester file and its data. If we decided to opt for a preexisting file type such as a database, xml or json, it would come with its own set of advantages and disadvantages, depending on our choice. The obvious benefit of using a pre-existing file would be time saving, as all the time we would spend developing our own custom semester file format we could spend on other areas of the program. Another benefit would be the ease of use for other developers after we passed the project onto them, as if they were familiar with the preexisting format already it would be trivial for them to begin entering their own semester data. This also raised a potential issue, as a pre-existing file format may not meet the requirements for the semester file and might need further code and effort to extend it.

However we elected to go with our own semester file format, which was a .txt file populated by semester data and separated by delimiters. This came with its own set of advantages and disadvantages: the main advantages were that it would give us much more control and flexibility for future changes, and would also remove the risk of any conflicts between any 3rd party applications and the Study Planner application. Unfortunately it resulted in a lot of extra work and testing to ensure all the correct data was populating the correct classes, but in the end it was definitely worth it as it enabled us a huge degree of flexibility in our approach, and made future tests a lot easier.

Another design decision we had to make was how many controller classes to write for the program. For simplicities sake, one controller class would have been able to manage the whole GUI side of the program, and would have been easier to keep track of than many different controller classes. However we felt that having many different controller classes, and having one to correspond to each GUI class, would make each individual controller class a lot easier to understand, and give our application a better degree of abstraction.

## Detailed Class Breakdown - Model Classes

### Student

This class manages the student currently logged in, and has possesses all the relevant student attributes, such as their full name, email address, year of study, and list of Modules they belong to. The class also has 2 different constructors which are used in different situations: the first creates a new student and populates its fields with the data from the constructor's parameters, while the second takes a semester file as its parameter, breaks it down into its relevant data tokens, then populates the student object and its related assessments with the data from the file. The first constructor is used in situations where the semester file has already been loaded, and a new student needs to be added without loading another semester file. The second one is used at the start of the program, and creates a new student, along with assessments, tasks and activities from the semester file, then associates them with the newly created student.

A switch statement within a loop manages this by choosing between case ‘E’ or case ‘A’: E being the first letter of an examination, and A being the first letter of an assignment. We designed the semester file around the file reading functions so for example, as soon as the function sees an E, it knows the next 8 tokens are going to be associated with an exam object, and a new exam object should be instantiated using that data. By ensuring the format of the semester file always stays the same, the length of the file or the number of exams or assignments is irrelevant, as the function will always fetch all the relevant data by iterating through the switch statement.

Another important function is getModuleByCode(String moduleCode) which takes a moduleCode as an argument then searches through the **student’s** modules until it finds the module that matches the given moduleCode, which is then returned. The other functions in student are fairly self-explanatory, with most of them being basic get() functions.

### Module

This class models a **Module**, and contains all the fields relevant to a **Module**, including the moduleCode, the name, the current grade and the ArrayList of the assessments that belong to it. Along with the usual get() methods, there are a few noteworthy methods: getAssessmentByIndex(int i) and getAssessmentByCode(String assessmentCode) gives the user two different ways to search for assessments within a **Module**; either by passing in an integer and searching by the index of the list, or by looping through every value within assessments until a match with the given assessmentCode is found. addNote(String note) simply concatenates the note parameter with the notes attribute of **Module.**

### Assessment

This is an abstract class to model a general assessed piece of work for a **Module**. Both **Assignment** and **Exam** inherit from this class, and thus they share many of its attributes and methods. The classes are modelled this way because **Exams** and **Assignments** are inherently very similar, and making them both children of **Assessment** makes for more efficient code and simplifies the structure of the program. **Assessment** only possesses fields relevant to both **Assignments** and **Exams**, such as weighting, grade, deadline, and the associated tasks. It also has an extra field called assessmentType, which is a string used to specify which child of **Assessment** to construct. Along with the usual get() and set() methods, there are methods for adding and removing tasks from the **Assessment**, and finding an exact task by a given taskID.

### Assignment

This is a child class of **Assessment**, and thus inherits most of its attribute and methods from **Assessment**, apart from 3 fields unique to **Assignment:** handInProcedure, assignmentType, and isSummative, which all apply to **Assignment**, not **Exam**. An important method of **Assignment** is assignmentToFile(), which returns a string matching the representation of that **Assignment** in the semester file's format, so that it can subsequently be compared to the actual file.

### Exam

This is a child class of **Assessment**, and thus inherits most of its attribute and methods from **Assessment**, apart from 2 fields unique to **Assignment:** examRoom and examDuration. It also possesses an examToFile() function, which works exactly the same way as it does in **Assignment**. However due to the differing number of fields between **Assignment** and **Exam**, unique ToFile() functions were needed for each respective class.

### Task

This class models the individual tasks that make up an **Assessment**. Its class variables include the usual name and ID, the weighting, the **Assessment** it belongs to, and the ArrayList of **Activities** that belong to the **Task**. Along with the standard get() and set() functions, it possesses a ToFile() function similar to the other previous ToFile() functions.

### Activity

This class defines the **Activities** that make up each individual **Task**. Because **Activity** is at the bottom of the file hierarchy, it is a fairly simple class that essentially only contains its members, the relevant getters and setters, and a ToFile() function, to enable easy comparisons against the semester file’s format.

**//GradePlanner**

This class performs basic mathematical operations on grades that are passed into its methods. It only has one member variable, an ArrayList of **Assessments**. The predictedAssessmentGrade(double predictedGrade, Assessment assessment)

### Deadline

This class is probably the simplest in the whole program, as it is essentially just a wrapper for the Date member variable. The only methods are a get() and set() function, and a toString().

## Detailed Class Breakdown - Controller Classes

We tried to make all of our Controller classes only have 1 member variable: a **DashboardController** object named dashboard. We did this to reduce the dependency on **Student** throughout the program, as most actions within the Study Planner involve the **Student** class in some form and passing the current **Student** into every function as a parameter wasn't an option. Model-View-Controller was also very useful during our group work, as it meant 3 of us could be working on different parts of what was essentially the same class simultaneously.

### ActivityController

This controller class enables the **ActivityGUI** and **Activity** classes to interact with each other, whilst ensuring that they do not conflict with each other. This class comes with the usual array of get() and set() functions for **Activity**, but with the lack of an **Activity** object to reference, all references to the members of **Activity** are done through the **DashboardController**, then **Student**, **Module**, **Assessment**, **Task**, and finally **Activity. ActivityController** also contains the function updateActivity(String moduleCode, String assessmentCode, String taskCode, int i, String name, String notes, boolean completed, double weighting), which takes all the necessary data to construct a new **Activity** as a parameter and updates an already existing **Activity** with that information.

### AssessmentController

As with the other Controller classes, the only class variable is the DashboardController, and this class focuses on controlling and manipulating the Model's (**Assessment)** data and displaying it to the View (**AssessmentGUI**). There is a corresponding get() and set() function for each of Assessments' class variables, with all relevant attributes accessed through the **DashboardController** instead of **Assessment** as in **ActivityController** and the other Controller classes. For example, the body of the getAssessmentCode(String moduleCode, int i) contains:

```
dashboard.getStudent().getModuleByCode(moduleCode).getAssessmentByIndex(i).getAssessmentCode()
```

Which accesses the **DashboardController,** then **Student, Module** and finally **Assessment.** This class also contains the viewAssessmentTasks(String moduleCode, int i) function, which loads every **Task** associated with the current **Assessment** and returns a DefaultTableModel to the GUI, where the table will be populated with the relevant **Tasks**.

### DashboardController

**DashboardController** is the single most important class in the project, as it is the only class with a reference to **Student,** where most of the semester data is stored. The constructor for this class calls new **Student**(semesterFile) as soon as this class is instantiated, so a new **Student**

object is created from the data in semesterFile, and the **Student's Module** information including all **Assessments** is also loaded into memory.

**DashboardController** also contains several functions crucial to the operation of the program, such as viewUpcomingCompleteAssessments(), which fetches all **Assessments** that have been completed and where the deadline has not elapsed.

```
!today.before(previousWeek) &&
today.before(student.getModule(i).getAssessmentByIndex(j).getDeadline().getTime())
```

The above logic is equal to: if the assessment is complete and it is within 7 days of the due date. The nested for loop in the function loops through every **Module** and **Assessment** and displays every **Assessment** that meets the criteria. Another important function in **DashboardController** is viewMissedAssessments, which returns a DefaultTableModel with every **Assessment** where the deadline has elapsed and the **Student** did not finish it. The operation is similar to viewUpcomingCompleteAssessments(), but the comparison is very different:

```
today.after(student.getModule(i).getAssessmentByIndex(j).getDeadline().getTime() && !added
```

The above logic is equal to: if the **Deadline** of the **Assessment** is after today, and if that **Assessment** hasn't been added to the table, then add this **Assessment** to the table for the **DashboardGUI** to display the results to the jFrame. Next is the findSemesterFile(String username) function, which is a boolean function which looks at the current **Student's** username, and checks if they've uploaded a semester file, and returns true if they have and false if they haven't. This method is called when a user logs in, and the returned boolean determines what happens next. If the user has not uploaded a file, the function uploadFile(String username, String source) is then called, which brings up the upload file window. If the user has uploaded a semester file, the Dashboard opens as usual and the user is presented by their missed deadlines, incomplete assessments and completed assessments.

Then we have updateFileForModule(Module mod), updateFileForAssignment(Module mod, Assignment assignment), updateFileForExam(Module mod, Exam exam), updateDeadlinesFromFile(File deadlineFile), and updateFileForActivity(Activity activity) which are all custom functions to manage updating the five classes from a provided file. Due to the differences in structure of the five classes, and the differing amounts of parameters, it was necessary to split the updating from file methods into bespoke functions that cater to each class exclusively.

The addTaskToFile(Module mod, Assessment assessment, Task task) method adds a **Task** passed in as a parameter to the semester file. The method then decides between adding the **Task** to an **Assignment** object or an **Exam** object by evaluating the first letter of the assessmentCode, for an ‘A’ for **Assignment** or ‘E’ for **Exam.**

Finally, the checkFile(File semesterFile) function provides error checking and attempts to match the semesterFile parameter to a regular expression. This function is a boolean which returns true if the semesterFile passes all the checks.

### ModuleController

Once again, the only member of this Controller class is a **DashboardController,** so all relevant **Module** information is fetched by referring to the **DashboardController,** then the **Student,** then after **Module** is referenced all relevant **Module** information is readily available e.g:

```
dashboard.getStudent().getModule(i).getModuleCode()
```

*In which i is an int representing the index of the desired Module*

**ModuleController** also contains the viewModules() function, which, like the other view functions, returns a DefaultTableModel containing all the **Modules** belonging to the currently signed in user, which is then loaded into a table by the **ModuleGUI** class.

Additionally, the viewModuleAssessments(int i) function uses a for loop to iterate through every **Assessment** in a chosen **Module** (given by the index i), while j is incremented every iteration of the loop to allow for the fetching of every **Assessment** related to the **Module**.

### TaskController

Finally, TaskController possesses a DashboardController which it uses within the get() and set() functions to access all the class variables of Task, via

```
dashboard.getStudent().getModuleByCode(moduleCode).getAssessmentByCode
(assessmentCode).getTask(taskIndex).getTaskID();
```

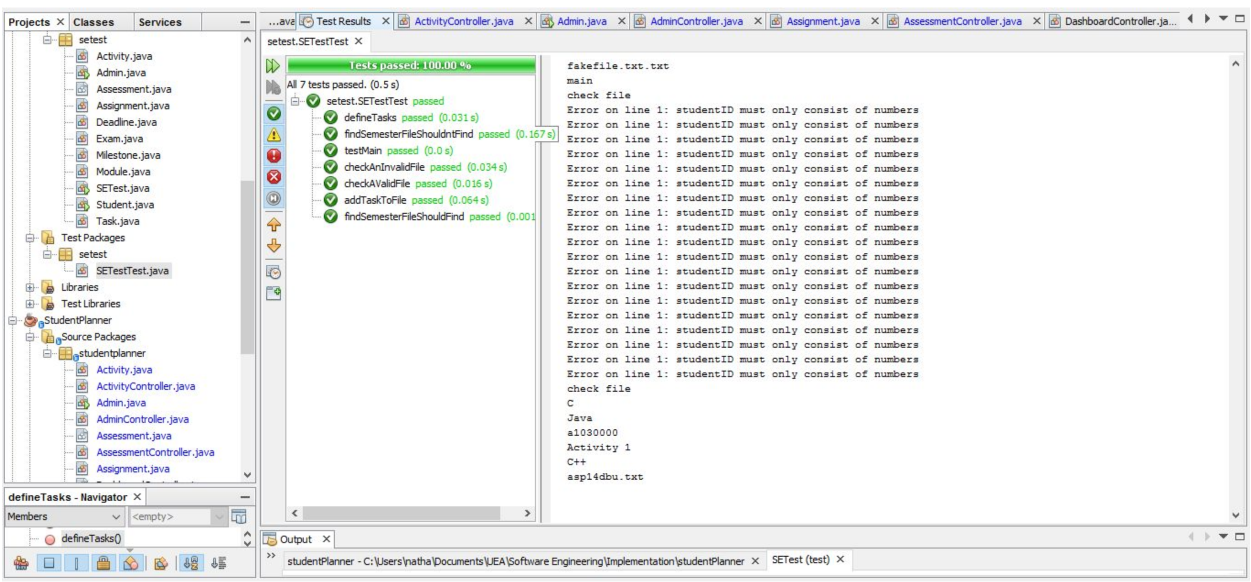
It also contains the updateTask(String moduleCode, String assessmentCode, int i, String name, String notes, double weighting) function, which takes in updated Task data as parameters, then calls the previously mentioned updateFileForTask(Module m, Task t) in the **DashboardController** class to write the modified Task to the semester file.

### GUI Classes

The view or GUI classes are extremely simple compared to the model or controller classes, as they usually just consist of a few text fields to be referenced by the controller classes.

## Testing

We tested our project constantly
throughout development via the use of main methods in classes, separate projects and by running the study planner to test the methods we developed were correct, we also used Junit testing to ensure our methods gave the result we were expecting. This is the test driven development (TDD)approach. Test driven development was a good choice of development approach for this assignment for a number of reasons. TDD keeps the programmer focused on the task at hand and stops them from getting distracted by other issues as they are focussed on getting the correct result from the test. TDD also decreases the number of bugs in the code as when testing is being undertaken bugs are constantly being discovered.




Junit test results

| | Requirements | | |
|---|---|---|---|
| Test Case | Upload Semeste | Store tasks | Define Tasks |
| findSemesterFileShouldntFind | x | | |
| findSemesterFileShouldFind | x | | |
| addTaskToFile | | x | |
| checkInvalidFile | x | | |
| checkValidFile | x | | |
| defineTasks | | | x |
| | | | |

Coverage Matrix

## Assumptions

We made the assumptions that both the semester file and the user file are provided by the HUB.

## Links to Git and Trello

**https://github.com/semipeeled/studentPlanner**

**https://trello.com/b/A5vOveFJ/se2-coursework**