A picture containing text, queen

Description automatically generated

Edge Hill University

The Department of Computer Science

# CIS2149 Object-orientated Programming

Level 5

Coursework 1 (CW1)

2019/2020

**Sean Carlin**

**24609650**

## **Table of Contents**

[CIS2149 Object-orientated Programming 1](#_Toc58107208)

[**Table of Contents** 2](#_Toc58107209)

[**Executive Summary** 4](#_Toc58107210)

[**Introduction** 4](#_Toc58107211)

[What is the Project? 4](#_Toc58107212)

[Planning 4](#_Toc58107213)

[Design 5](#_Toc58107214)

[Development 5](#_Toc58107215)

[Testing 5](#_Toc58107216)

[Potential Challenges 5](#_Toc58107217)

[**Requirements** 6](#_Toc58107218)

[**Design** 7](#_Toc58107219)

[Purpose of Object-Oriented Design 7](#_Toc58107220)

[Design Methodology 7](#_Toc58107221)

[UML Diagrams 7](#_Toc58107222)

[Class diagrams 7](#_Toc58107223)

[Hierarchical Inheritance – Staff, Academic & Tutor. 7](#_Toc58107224)

[Student 8](#_Toc58107225)

[Module 8](#_Toc58107226)

[Course 8](#_Toc58107227)

[Statistics 8](#_Toc58107228)

[Assignment 9](#_Toc58107229)

[Marked Assignment 9](#_Toc58107230)

[Applicant Exception 9](#_Toc58107231)

[Class Statistics 9](#_Toc58107232)

[Overall System Diagram. 10](#_Toc58107233)

[Activity Diagrams 11](#_Toc58107234)

[**Design Rationale** 13](#_Toc58107235)

[Functionality 13](#_Toc58107236)

[Functionality & Better User Experience 14](#_Toc58107237)

[Data Handling & Management 14](#_Toc58107238)

[Efficiency 16](#_Toc58107239)

[Error Handling / Validation 16](#_Toc58107240)

[**Design Principles** 17](#_Toc58107241)

[Robustness 17](#_Toc58107242)

[Adaptability 17](#_Toc58107243)

[Reusability 18](#_Toc58107244)

[Modularity 18](#_Toc58107245)

[Abstraction 18](#_Toc58107246)

[Encapsulation 19](#_Toc58107247)

[**Object Oriented Principles** 19](#_Toc58107248)

[Objects 19](#_Toc58107249)

[Interfaces 19](#_Toc58107250)

[Polymorphism 20](#_Toc58107251)

[**Summary** 21](#_Toc58107252)

[**References** 22](#_Toc58107253)

[**Appendices** 23](#_Toc58107254)

[Appendix 1 - Notes 23](#_Toc58107255)

[Appendix 2 – Splash Screen Image. 30](#_Toc58107256)

[Appendix 3 – LookAndFeelInfo Theme 31](#_Toc58107257)

[Source Code 31](#_Toc58107258)

[Comparison 31](#_Toc58107259)

[**With** *Nimbus* Theme 31](#_Toc58107260)

[**Without** *Nimbus* Theme 31](#_Toc58107261)

[Appendix 4 – Possible GUI Designs. 32](#_Toc58107262)

[Appendix 5 – JList, ArrayList & DefaultListModel Method 34](#_Toc58107263)

[Appendix 6 – Display Values replacing Iterators. 35](#_Toc58107264)

[Appendix 7 – Data In & Out 35](#_Toc58107265)

[Appendix 8 – Applicant Exception + Example 36](#_Toc58107266)

[Appendix 9 - Encapsulation 37](#_Toc58107267)

[Appendix 10 – Assistance 38](#_Toc58107268)

[JavaDoc API 38](#_Toc58107269)

[How to Populate for AutoGen ID. 38](#_Toc58107270)

[Opening the Project 38](#_Toc58107271)

## **Executive Summary**

This report will explain in detail about my final project, through its planning and prototyping stage, straight through to development and testing.

This project is based on client requirements for a Student Management System within the university department where it enables staff members to maintain and organise students, tutors, academics, modules and courses, all within a detailed system making it easier to find exactly what they need.

Upon loading the user is greeted with an aesthetic graphical user interface (GUI) that is formally address for what each section of the system provides, including icons relevant adding to aesthetics.

The users can add, edit and delete students or staff through the multiple sections the system has been divided into for ease of access. All data inputted from the user is written out to data files, so once the user boots up the system all data stored will be reloaded.

Users can also create assignments for students and even mark these assignments where all results relating to the specific students are stored within a data file, results are also written out to a separate report.

This same method also works for Student statistics, as the system automatically generates a student average once the desired student and assignment is selected.

## **Introduction**

### What is the Project?

Edge Hill University’s student services department struggle to maintain all their students, staff, courses and modules due to their continuous growth every year. Therefore, a new Object-Oriented system needs to be developed with Object Oriented Design principles to resolve this issue, beginning with 4 stages;

|  |  |
| --- | --- |
| 1. Planning | 1. Development |
| 1. Design | 1. Testing |

#### Planning

Throughout planning I had to ensure all *requirements* can be met for the project all within the timeframe to ensure it gets delivered on time. To overcome any issues down the line, I took notes(*appendices)* on what techniques for the class development to use, types of lists, maps, what fields of values will be included in all classes and how forms would potentially present themselves through a Graphical User Interface(GUI). This reduced development time if an error were to arise in terms of functionality.

#### Design

This system will be used by many people, so the appearance, layout and overall theme of the system with the use of a *Graphical User Interface* (GUI) was a key factor. This ranged from Fonts, text sizing, colouring and graphical presentation to the user, which was considered greatly as you can see in *appendix 2.*

With the implementation of icons relevant to their areas it helps add to the aesthetics, text sizing not too large and eligible. The colouring was kept minimal with a cyan blue/grey theme so it doesn’t affect the user’s eyes, and everything can be seen clearly.

To conclude, I implemented a theme that works alongside **Java Swing** to improve overall presentation of the *JButtons* and *JFrames etc,* called ***LookAndFeelInfo,*** a comparison of the theme including *source code* is at *appendix 3.*

#### Development

During development, there will be a time frame for each section of the system to be completed by, this is to ensure efficiency and delivery on time. For this to work affectively, all requirements need to be considered in advance to avoid any errors. The focus during development is functionality, robustness, adaptability and reusability to name a few. This ensures that university department gets an operational system that delivers speed and meets demands without any errors.

As mentioned previously, the system will be based off object-oriented designs and principles so functionality and validation requirements will be sufficiently met. Adding to functionality, we are going to implement a *Search* method that allows the users to search for a user, reducing time rather than having to scroll through lists of data.

#### Testing

The final yet most important stage of the project is to ensure the system works, this being functionality, validation and storage of appropriate data. During the final development stage, validation techniques will be implemented to validate user inputted data from *JTextFields & JTextAreas etc.* This data will be validated using *ApplicantException* that can be specified to validate how many letters, types and values of numbers are entered to ensure appropriate data stored. Once validated, it will be written out to a data file to be stored and when needed, that data can be pulled back into the system. Functionality testing will be done rigorously to ensure the system can withstand the workload.

### Potential Challenges

During development challenges are expected, and if needed a different pathway will be applied to ensure the system is delivered on time. For now, the main challenge is implementation of current student data within the university department.

## **Requirements**

See below a detailed requirement list for the Student Management system.

|  |  |
| --- | --- |
|  | Requirements |
|  | Application must have Java Swing-Based Graphical User Interface (GUI) |
|  | Must represent relevant areas within the University department – students, academics, tutors, overall staff. Followed by relevant information like gender, student supervised, position, office location and hours, expertise area (course they teach), employment status e.g. full time etc. |
|  | Users must be able to add, edit and delete students, academics, tutors, staff and modules. |
|  | Application must allow users to add students to modules through ‘Edit’ |
|  | Users must be able to display a specific category when needed (e.g. display tutors, modules or students) |
|  | Users must be able to add assignments |
|  | Users must be able to mark individual students based off a specifically selected assignment |
|  | Each assignment assigned must have date assigned, grade and feedback. |
|  | Marking assignments must include chosen student, with assignment name, mark, grade. |
|  | Application must automatically assign grade based on mark given to student. |
|  | Application must automatically generate student average based on mark against assignment total. |
|  | All data input to the system must be saved to an appropriate ‘.dat’ file and be written/saved to file upon application exit. |
|  | All data stored within the ‘.dat’ file must be loaded back into the system upon system start-up. |
|  | Application must allow users to search for a specific user from a specified category. |
|  | JDK package must be version 14+ |
|  | In relation to the GUI, Application must be aesthetically presentable, including use of graphics like icons, appropriate colours, text font and sizing. |
|  | Application must have Admin tools to pre-populate |
|  | Application must have help tabs to inform users when needed. |

## **Design**

### Purpose of Object-Oriented Design

The sole purpose of object-oriented design is to resolve software problems within a workplace. It involves planning a “*system of interacting objects [1]”* where object-oriented programming is applied to the application through *Design Principles*. The term ‘*object’* contains encapsulated data, grouped together through procedures to represent an entity. How this object interacts to solve software problems are defined through the object interface and its relating values. Another technical approach to applying object-oriented design to an application is by planning using *Object-Oriented Analysis and design (OOAD)*. As a result, I have decided to use object-oriented *design and programming* principles for the development of the student management system.

### Design Methodology

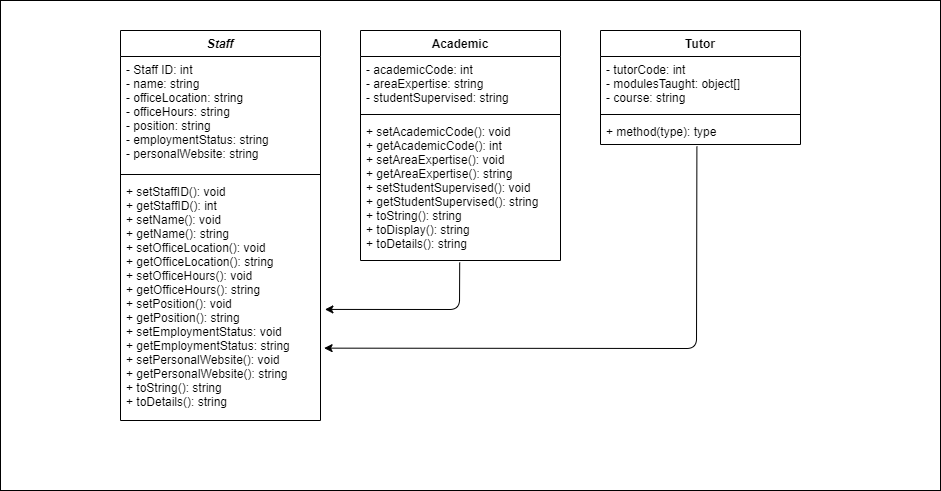
Starting development of the project with object-oriented design consisted of creating a graphical user interface that enabled the user to input data based on a specific area of the system, and what happened behind that interface were defined objects being implemented with that said data, relevant to their values.

Therefore, to ensure the GUI was user friendly, I started sketching possible designs to work with *(Appendix 4).* Followed by *Class Diagrams* that helped define each classes field of data and variables declared, programming became underway with the object-oriented approach.

### UML Diagrams

#### Class diagrams

##### Hierarchical Inheritance – Staff, Academic & Tutor.



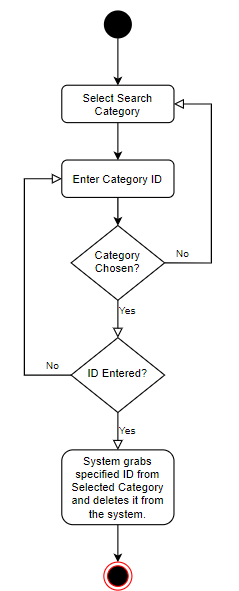
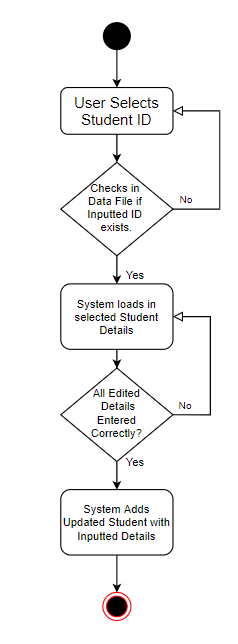
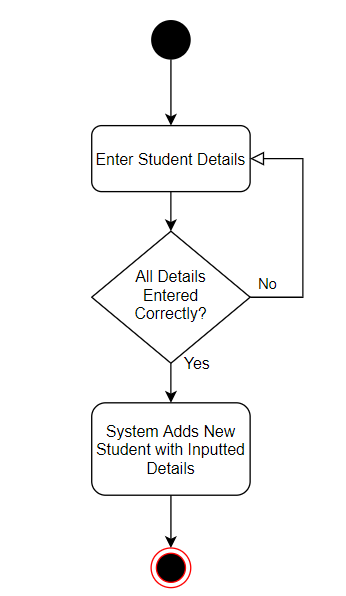
***Inheritance***

|  |  |
| --- | --- |
| Student | Module |
|  |  |
|  |  |
| Course | Statistics |
|  |  |

|  |  |
| --- | --- |
| Assignment | Marked Assignment |
| ***Inheritance*** | |
| Applicant Exception | Class Statistics |
|  |  |

##### Overall System Diagram.

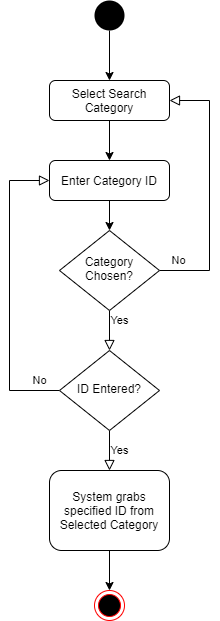
##### Activity Diagrams



***Add Student***

***Edit Student***

***Delete User***



***Search User***

### **Design Rationale**

Throughout the design period, many changes were made to improve:

|  |  |
| --- | --- |
| 1. Functionality | 1. Efficiency |
| 1. Better User Experience | 1. Error Handling/Validation |
| 1. Data Handling & Management |  |

#### Functionality

Improving the most important key factor of the student management system is functionality, thus improving it with the use of *Inheritance* within user classes for example. (Tutor, Student etc.) I decided to use ***Hierarchical Inheritance***when creating the following classes:

|  |  |
| --- | --- |
| * **Staff** (Super Class) | * **Assignment** (Super Class) |
| * Tutor – Sub Class | * Marked Assignment – Sub Class |
| * Academic – Sub Class |  |

My reasoning for this was to improve *Reusability* and to reduce repetitive code between each class that have similar attributes. Inheritance allows the capability of extending the super class values over multiple classes if required, where these values can be either added to or re-defined. Within the management system, a staff member can be security, a cleaner or in this scenario, A tutor and an academic both are technically ‘staff’ within the university. However, both have different areas they specialise in. So, both Tutor and Academic classes extend the ‘Staff’ *superclass.* Which allows values from Staff to be defined (overwritten in some cases) within Tutor or Academic upon user input, without having to duplicate these values within every class.

A basic diagram I created below (*Figure* *1)* represents Hierarchical Inheritance, or you can refer to the previous UML Diagrams.

Staff

Tutor

Academic

**Super Class**

**Sub Class**

**Sub Class**

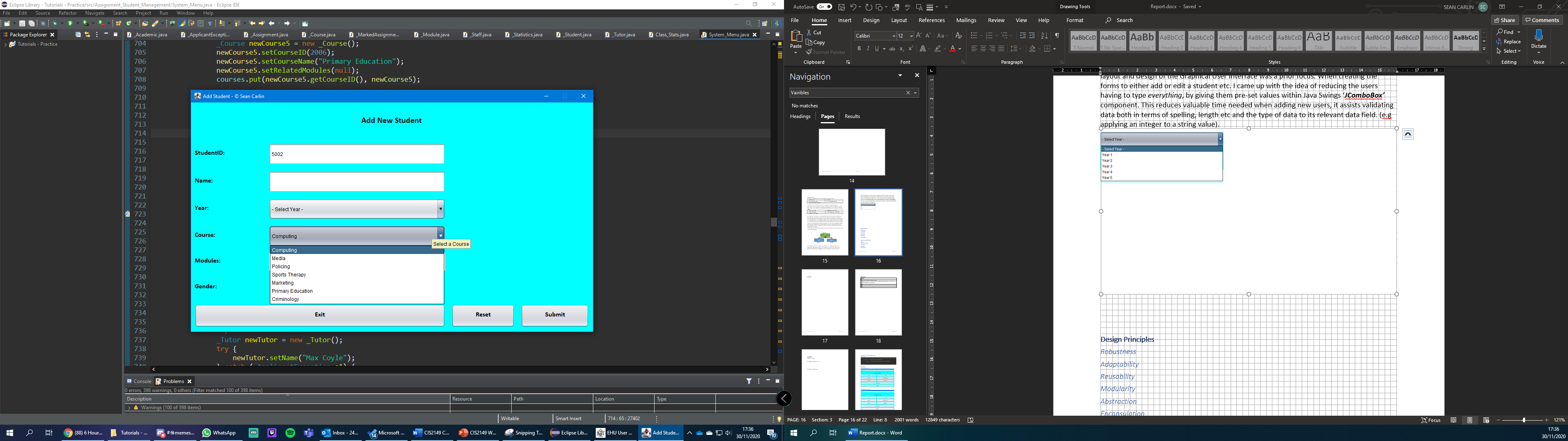
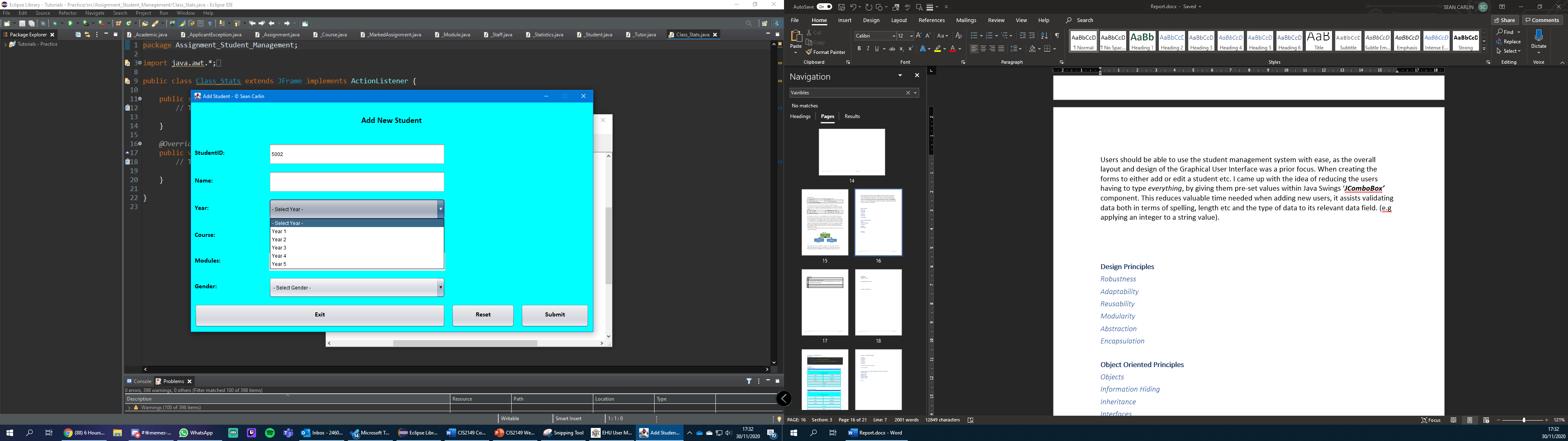
**Figure 1**

When adding a new Tutor or Academic to the system, the user would input those independent class values. However, not all values from the staff super class must be defined, only specifics like the *Staff ID* or any other relevant to the type of member.

#### Functionality & Better User Experience

Users should be able to use the student management system with ease, as the overall layout and design of the Graphical User Interface was a prior focus. When creating the forms for add or edit a student etc. I came up with the idea of reducing the users having to type *everything*, by giving them pre-set values within Java Swings ‘***JComboBox’*** component *(Figure 2).*  This reduces valuable time needed when adding new users, it assists validating data both in terms of spelling, length etc and the type of data to its relevant data field adding to the *Robustness* of the system.(e.g. applying an integer to a string value).

Whilst implementing the combo boxes, It gave me the idea of automatically filling the courses pulled from the ‘.dat’ file. This meant that whenever a new course is added, it would automatically be added to combo boxes linked to its ***ArrayList****.* *(Figure 3).* This ensures all courses selected by the user are genuine and cannot be misspelt.



**Figure 2**

**Figure 3**

cmbCourse.setToolTipText("Select a Course");

Small Details adding to the user-friendly GUI

#### Data Handling & Management

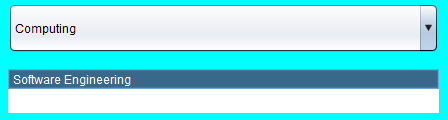
When the user selects a course, they are not prompted to enter a module, as not everyone can remember so many modules. Therefore, dependant on the *Course selected* through the ***JComboBox****,* *Modules related* will be displayed onto a ***JavaList (JList).*** Whilst programming I realised that Java Lists have no methods to add or delete items once they are initialized. The proposed solution I first attempted to fix this with was:

lstModules = new JList(moduleArray.toArray());

This approach did not work. So, I declared an instance of ***DefaultListModel***. Created it and then populated it with the values of modules that were pulled from *module data file*. I then used a ***for Loop*** to retrieve that data from the ***TreeMap*** which proceeded to store the data within an ***ArrayList,*** this method is also used throughout the system with Courses JComboBoxes etc (*Appendix 5*)***.*** However, the task planned was to display modules that are *related to the selected Course*.

***Continued Overleaf…***

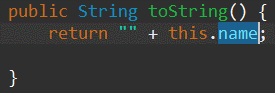
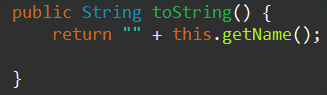
I created another for loop that went looped through the ‘*relatedModuleDATA’* ArrayList, pulled the ‘relatedCourse’ attribute from the module class and matched it with the course selected from the JComboBox. Resulting in the *DefaultListModel* ‘*lstDataArray’* displaying these values within the JList, enabling the system to change the display values when needed (**Figure 4)**. *Source Code Appendix 5.* Overall, this approach improved the systems validation and reduced the amount of error handling required, giving the users easy navigation and the university department valid data.



**Figure 4**

When designing the system, my original plan was to use ***HashMap’s,*** to store the user data, then further down development, ***Iterators*** would’ve been used to display this data appropriately. However, HashMap’s have no sorting algorithm or any sorting methods as standard, and my plan was to create a method that automatically grabs the last entered value, and increment it by 1 to automatically create the new ID (Staff ID, Tutor ID etc).

To do this properly, I had to switch to ***TreeMaps*** which are *slightly* sorted. This meant I could no longer use iterators properly either, which were replaced with a for loop and appended the values to the display (*Appendix 6).* I then had to overwrite the ***toString*** method within all classes. See example **Figure 5**.



**Before**

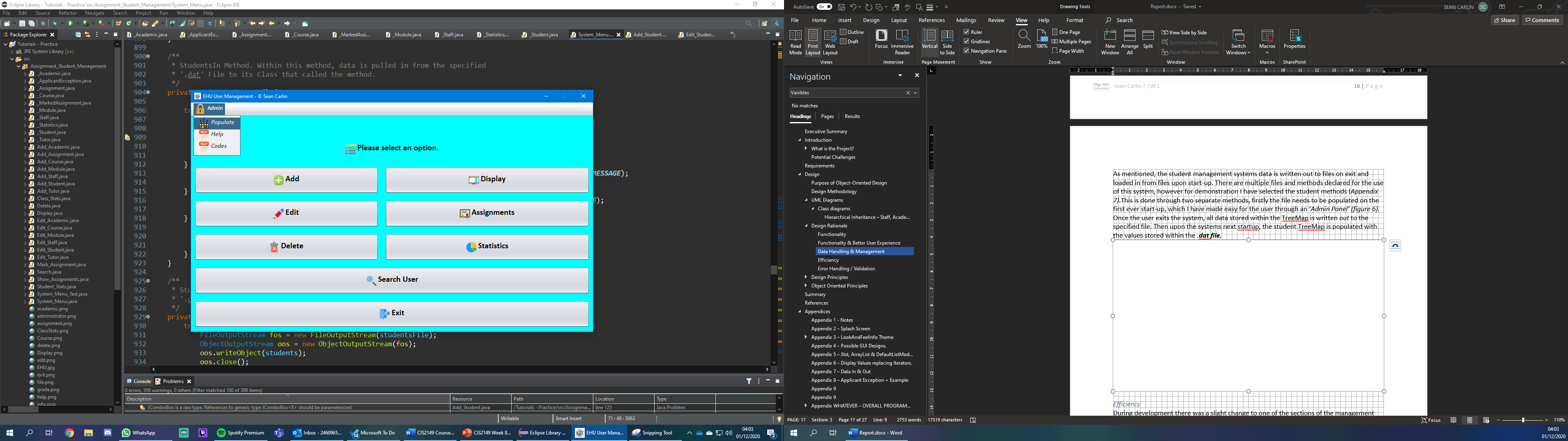
**After**

**Figure 5**

***Continued Overleaf…***

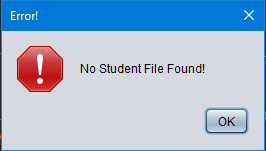
As mentioned, the student management systems data is written out to files on exit and loaded in from files upon start-up. There are multiple files and methods declared for the use of this system, however for demonstration I have selected the student methods (*Appendix 7)*.This is done through two separate methods, firstly the file needs to be populated on the first ever start-up, which I have made easy for the user through an ‘*Admin Panel’* *(figure 6).* Once the user exits the system, all data stored within the TreeMap is written *out* to the specified file. Then upon the systems next start-up, the student TreeMap is populated with the values stored within the ***.dat file.***

If there’s an error with the files, an error message will appear, *figure 7.*



**Figure 6**

**Figure 7**



#### Efficiency

During development there was a slight change to one of the sections of the management system and that was the student class lists. I decided to abandon this section and focus more on the search engine for the users and *Statistics* per student. Personally, I seen the search engine being a more important tool for the university department, but its reasoning was more so the time schedule. I still stuck by the Object-Oriented approach when making the search engine as well as the statistics form.

#### Error Handling / Validation

As mentioned previously, auto filled combo boxes with pre-set values help to avoid invalid data input. However, not everything is inputted via combo boxes or lists, users must enter values within ***JTextFields*** throughout the system. This is done through the ***ApplicantException*** class where I can set the minimum and maximum values of strings, integers etc. (*Appendix 8)*

### **Design Principles**

#### Robustness

As mentioned previously, I have implemented many **JComboBoxes** to reduce input from the user and have pre-set values instead. However, the main technique to maintaining least user error is the ***Applicant Exception*** method. I created a class with the exception error message that extends Javas own ***Exception class.*** My abstract classes then extending applicant exception, allowing me to apply error handling to specific values. For example, if a user were to enter an assignment mark with the value ‘95’(Min 0 – Max 80), it would deny the user to submit that marked assignment, display an error message stating the error and then enable the user to fix their issue. Source code & example *Appendix 8.*

#### Adaptability

Throughout design and development, I decided to use multiple collections to help manage and store data that was stored by users into the defined abstract classes. Abstract classes were used to define specific areas within the university department, like a *Student, or tutor* etc. The Collections used were ***ArrayList’s****,* ***TreeMaps*** *(sister to* ***HashMap’s****, but with a bit of sorting),* ***JavaLists*** and ***LinkedList****.* I did also use the ***DefaultListModel***technique, however, list models do not support collections, so this was used to help with specific tasks.

TreeMaps were mainly the heart of data handling within the system, other collections mentioned assisted handling that said data into their relevant areas throughout (*Figure 8)*. Using the LinkedList was to combine 2 Maps together to result in a final output. Refer to *Appendix 5* to see an example of some collections mentioned above being used.

**Figure 8**

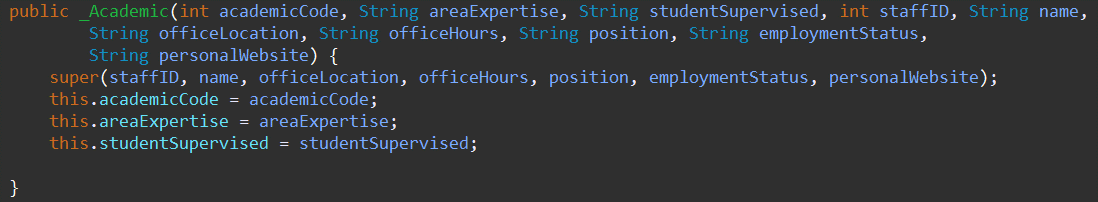


#### Reusability

Within my system I have used a *Hierarchical Inheritance* scheme based off 3 abstract classes. Staff being the *superclass,* with Tutor and Academic being the *sub-classes(children/child).* I went with this principle approach as these three classes all carry the same key value, that being a ***Staff member*** within the university department. So, to prevent repetition, the inheritance scheme was implemented and the different values for academic and tutor were declared within those abstract classes. You can see an example of the staff values being *extended* within the *Academic Class* in *figure 9.*

This enables the user to add a new tutor or academic without having to create a staff member first, as it goes, “*to kill two birds with one stone.” [2].*

**Figure 9**



#### Modularity

Within the student management system, not much modularity was used in terms of ***packages.*** The system is currently running off one package that contains the entire source code for every class. Personally, I didn’t see a need to separate classes as the system runs efficiently without. However, if I had more time in the future, I would use this method as standard to improve system modularity and overall tidiness.

#### Abstraction

During the system development, the classes created didn’t exactly have any relevant methods to return values. Therefore, the *superclass* used for our hierarchical inheritance approach isn’t ***abstract classed,*** as I didn’t see this useful.

#### Encapsulation

One of the 4 fundamentals of Object-Oriented Programming concepts, within all classes used to describe a type of person or subject in the management system, incorporate *encapsulation*. What encapsulation does to my classes is it basically wraps the data(variables) and code acting within that data(methods) together as one. These variables will be hidden from other classes (unless inherited), this is also known as *Data Hiding.* To Encapsulate data, I have had to declare my variables as ***private*** and create public ***Getters & Setters*** where they can modify and view the values when initialized in other classes. I see this as a great approach to developing the system has it keeps all data managed and separate from each class effectively. For source code and example refer to *Appendix 9.*

### **Object Oriented Principles**

This section includes principles *I have not* already discussed through the *Design Principles* section.

#### Objects

Within Java, everything is associated with classes and objects with their attributes and methods. Classes are basically *blueprinting* for the outcome of an object.

For example, a plane has attributes which could be its colour and weight, methods would then be deploy landing gear, start engines etc.

Overall student management system is based off various classes and methods that all work effectively together. Without them, the program wouldn’t know how to handle any code written. *Figure 10* shows objects/Classes being declared.



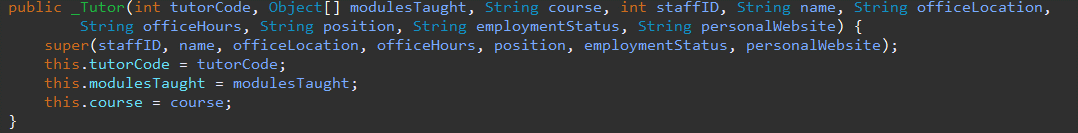
**Figure 10**

#### Interfaces

I haven’t used interfaces as I didn’t see a major difference between a regular class and them. Terms of functionality not much difference either, so I decided not to include this in the design stage. In terms of classes, all work efficiently with Java Swing and pass through data effectively. With *implementation*, the only thing implemented within these classes is ***java.io.serializable.*** Which is used to help the **ObjectOutputStream and ObjectInputStream** when writing in and out to files. Refer to *Appendix 7* for an example.

#### Polymorphism

Another Object-oriented fundamental, Very similar to inheritance, where two sub-classes inherit from the same *superclass* and you can perform a single action in different ways. I did this with Staff, Tutor and Academic classes. Where Academic and Tutor both inherit from Staff. When creating a new instance of either academic or tutor, those sub-classes would pull the methods from Staff to *override/overloading* the method needed to store its value. So, from the management system, if I were to add a new Tutor, the tutor class would *pull* the Staff ID method (and others if needed) to the new tutor being created, tutors also have their own ID independently. The Staff ID is used as tutors are still classed as staff, hence the inheritance, the same goes for Academics also. In *figure 11* you can see the Staff attributes being passed into the Tutor class through **super** constructor.



**Figure 11**

## **Summary**

As I was developing the student management system, I was very strict with the requirements and made sure almost all were met. However, there were a few errors along the way, beginning with Student classes. Student classes would have required a lot of time to implement especially as it encompasses roughly 4 abstract classes, therefore, I made the tough decision to abandon that area of the project completely and focus more on *Student Statistics.* Within student statistics there is a *generate report* option that prints out all the data for the selected student, to a data file named after said student. I personally see this as an excellent decision because having a method that automatically generates student average based off marks, is better than simply viewing what class they’re in.

During development a last-minute decision using Maps was to change from HashMap’s to TreeMaps, reasoning for this was because of sorting. HashMap’s have no sorting methods at all whereas TreeMaps do a little bit, with the help of ArrayList and for loops of course. This last-minute decision improved the overall functionality of the system and kept the project within the time scale given.

If I were to make improvements to the student management system, I would firstly make the search engine more advanced. Now, the search engine currently pulls from a selected category. In the future I would like to improve this and make it, so the user doesn’t have to select a category and instead enter the values they want to find, letting the engine do the rest. As mentioned above, I abandoned the student class lists for time priority over other areas, if there was more time allocated, I would implement this form and have classes allocated to each student added to the system with a relating course. Another minor issue I encountered was displaying selected modules through a ***JtextArea,*** and the toString for the text areas alone. If I had more time, I would fix this, however, its only when it needs displayed, the data is still stored otherwise. To conclude, sometimes the Staff ID doesn’t automatically auto-gen due to an unknown bug.

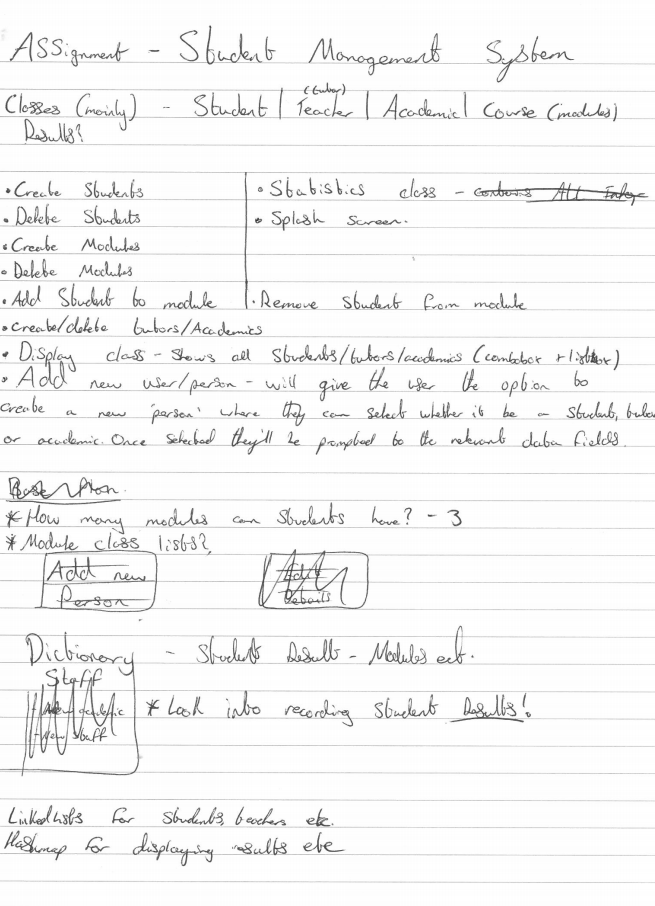
Overall, the university department now have a fully functional student management system, that comes equipped with a user-friendly graphical user interface (GUI) with colours and icon aesthetics that improve navigation. The users of this system can easily add, edit, display, delete and search for all its available sectors. This system also comes with the data file approach where all data inserted will always be available for the user to retrieve when needed. In terms of code for the university’s IT management team, most of the code is ***JavaDoc*** Commented so everything is eligible.

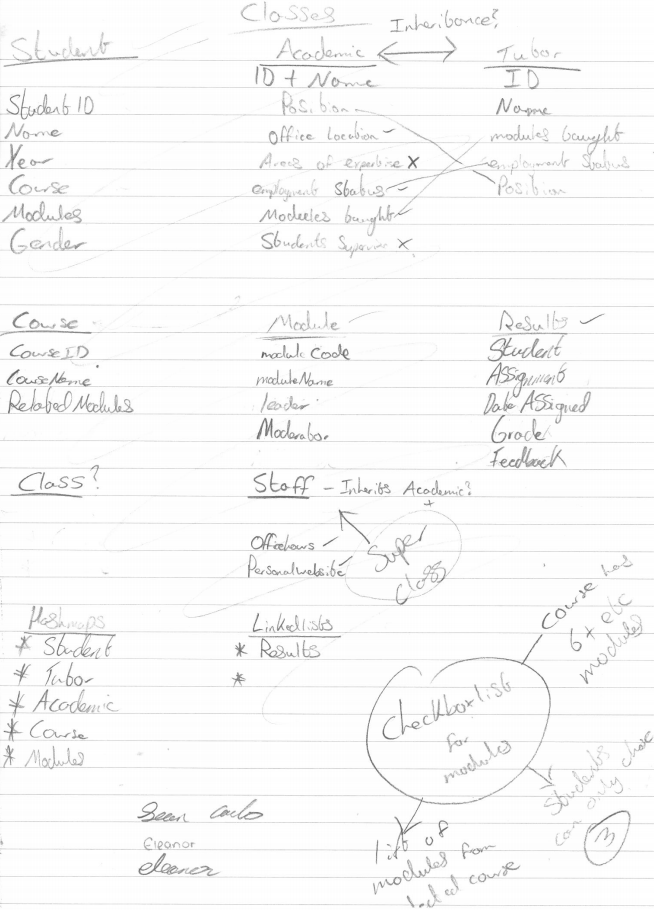
## **References**

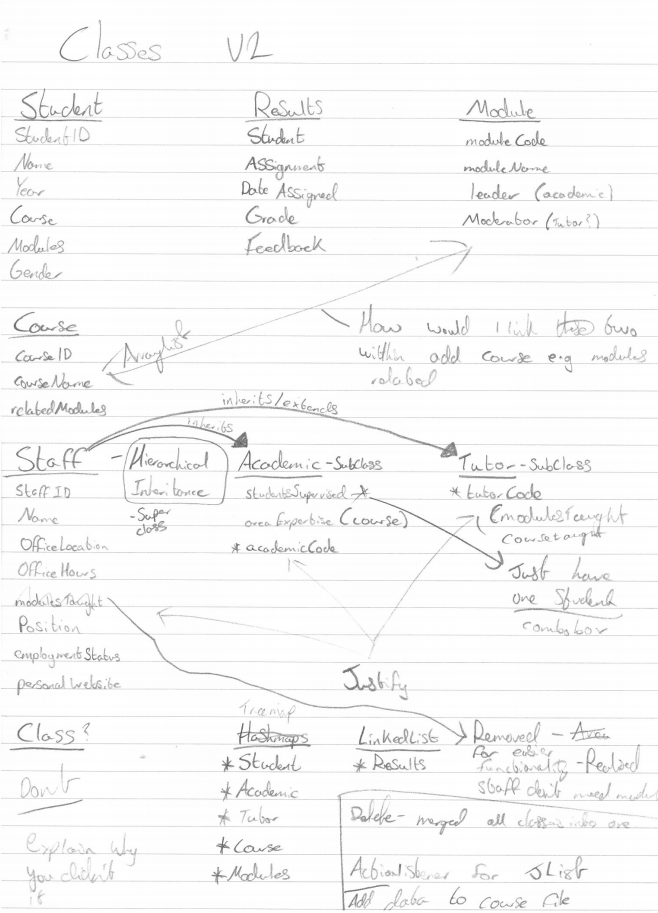
|  |  |
| --- | --- |
| # | Link |
|  | Wikipedia (02/11/2020) *Object Oriented Design,*Available at: *https://en.wikipedia.org/wiki/Object-oriented\_design#:~:text=Object%2Doriented%20design%20is%20the,one%20approach%20to%20software%20design.* (Accessed: 26/11/2020). |
|  | Cambridge Dictionary (2018) *Two birds - One Stone,*Available at: *https://dictionary.cambridge.org/dictionary/english/kill-two-birds-with-one-stone* (Accessed: 28/11/2020). |

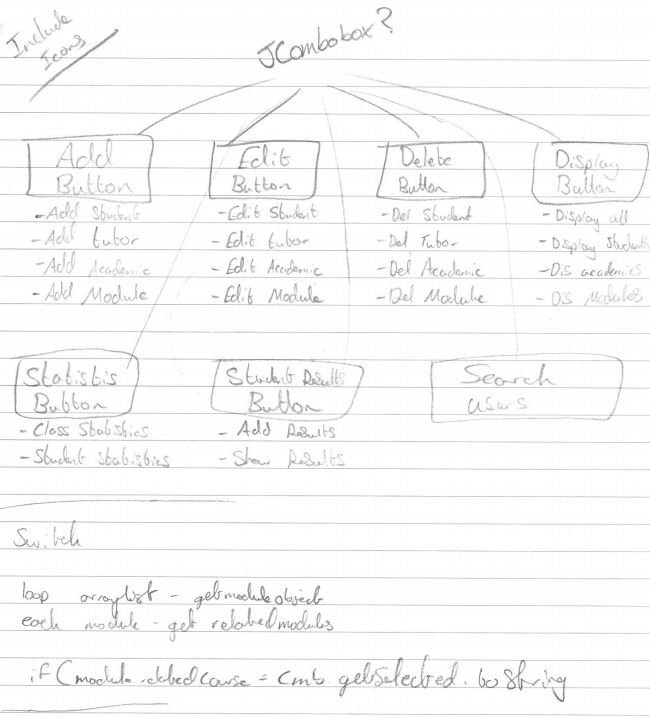
## **Appendices**

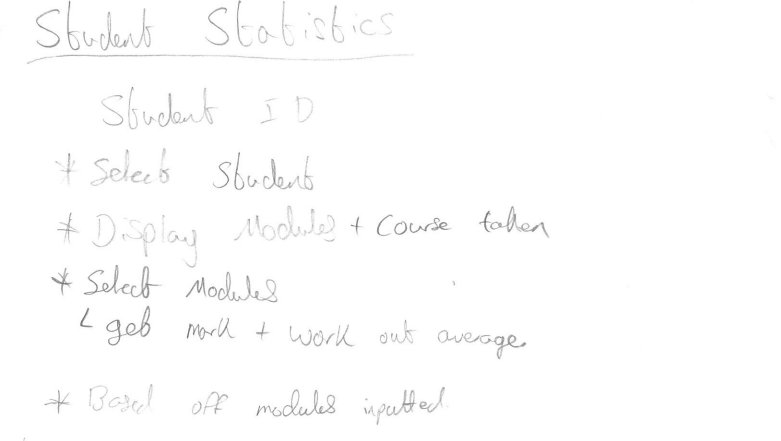
### Appendix 1 - Notes

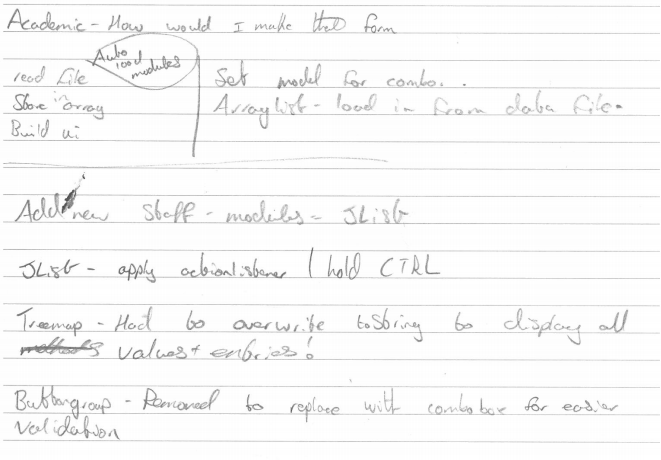


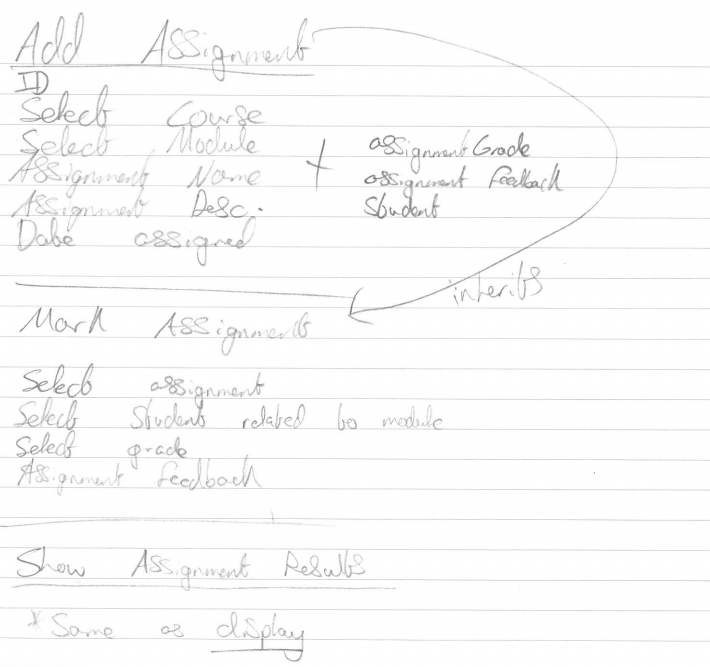


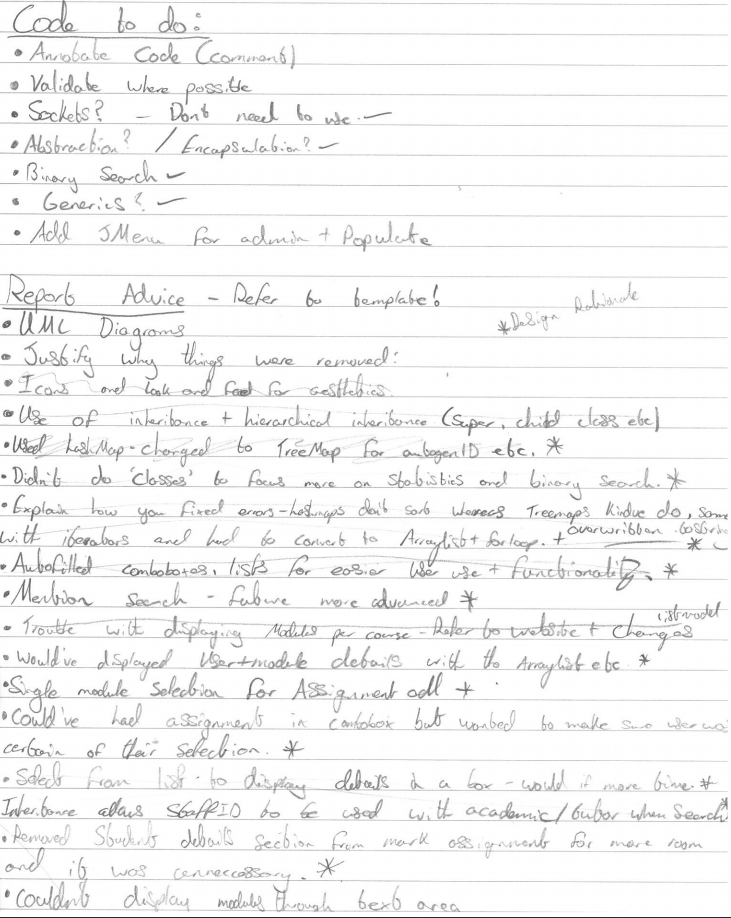












### Appendix 2 – Splash Screen Image.

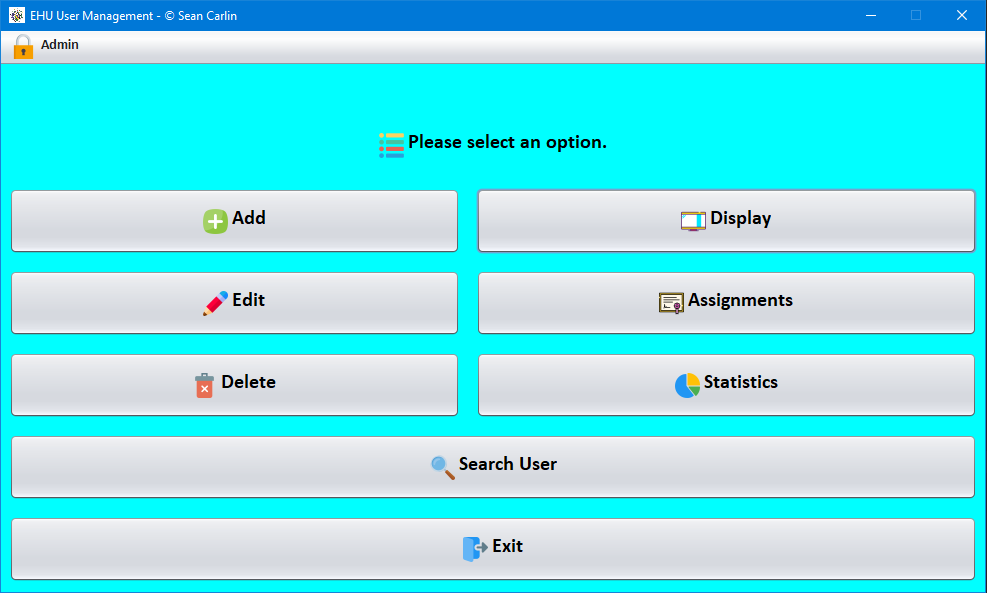


### Appendix 3 – LookAndFeelInfo Theme

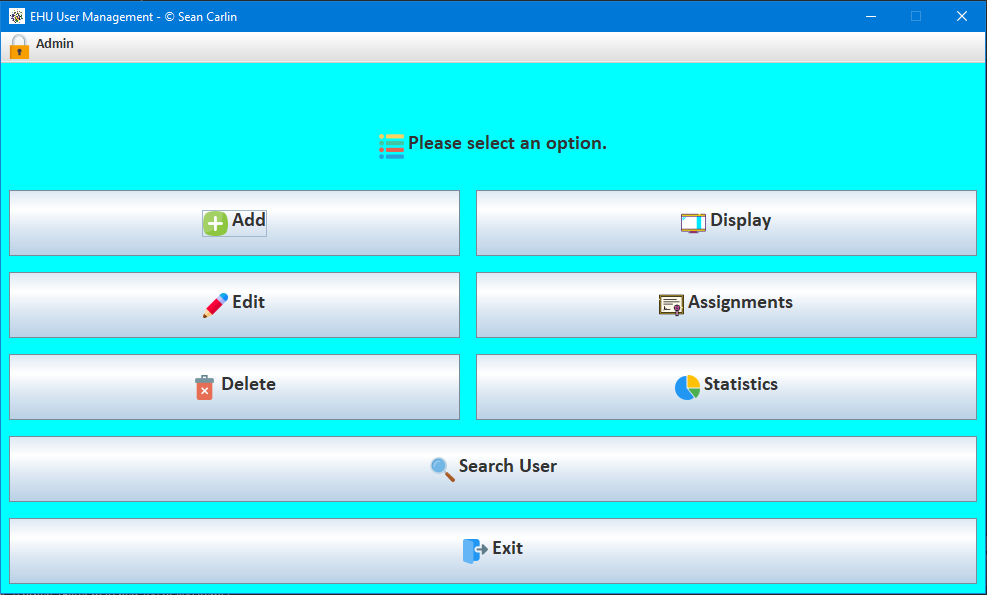
#### Source Code

#### Comparison

##### **With** *Nimbus* Theme

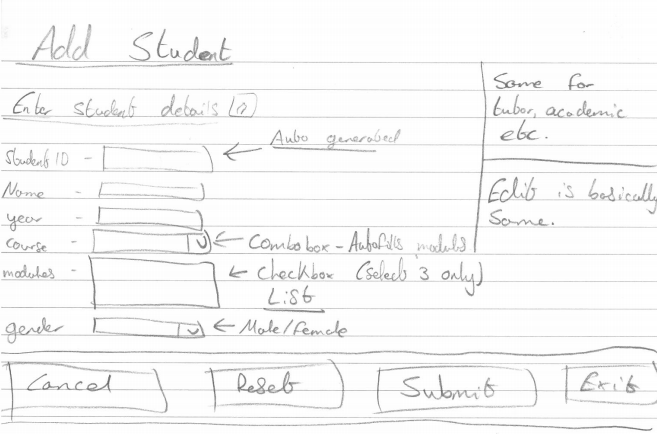


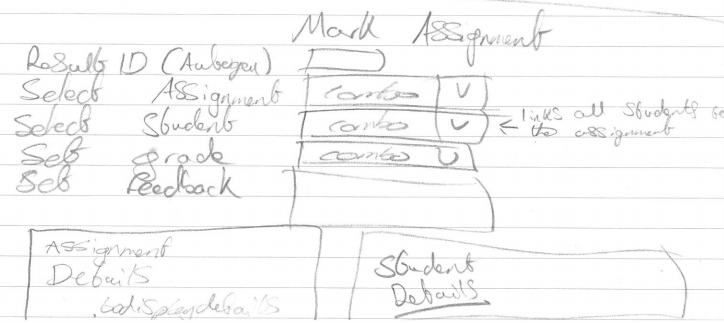
##### **Without** *Nimbus* Theme



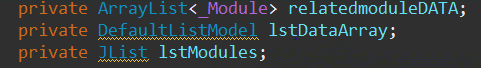
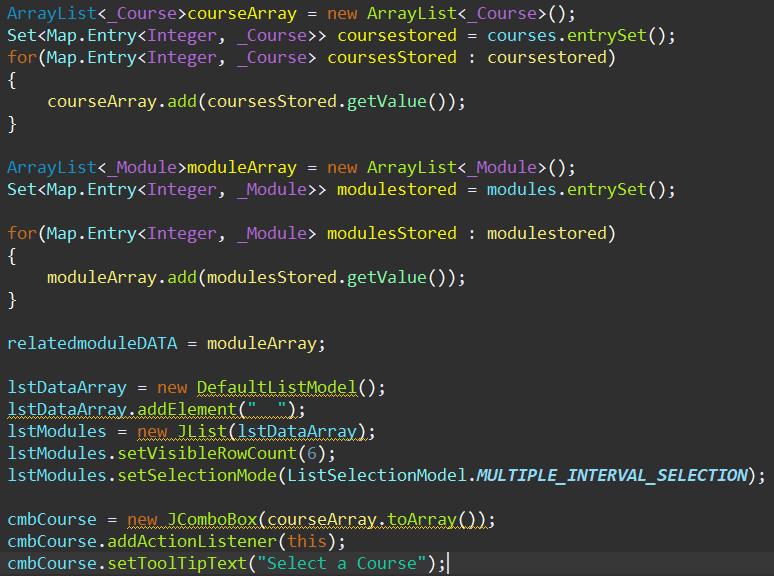
### Appendix 4 – Possible GUI Designs.



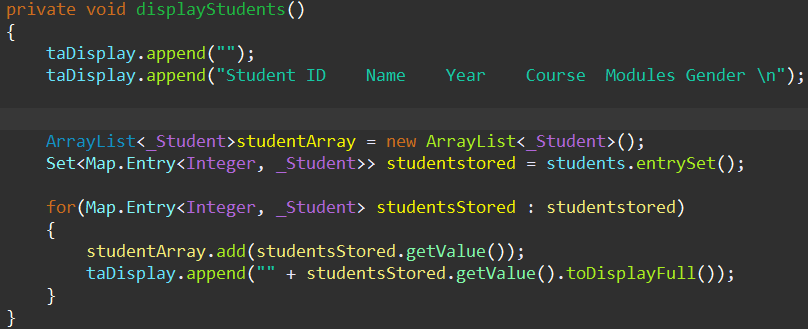




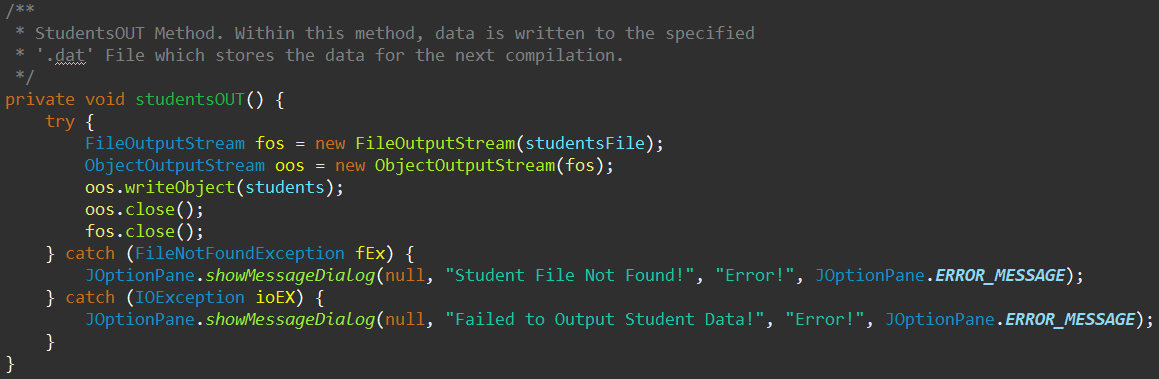
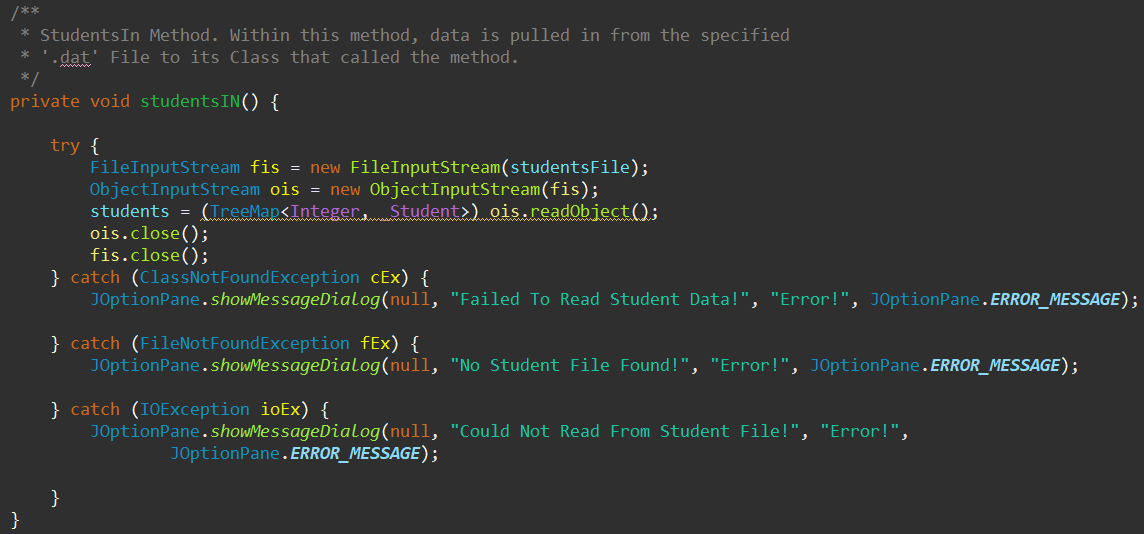
### Appendix 5 – JList, ArrayList & DefaultListModel Method



### Appendix 6 – Display Values replacing Iterators.



### Appendix 7 – Data In & Out

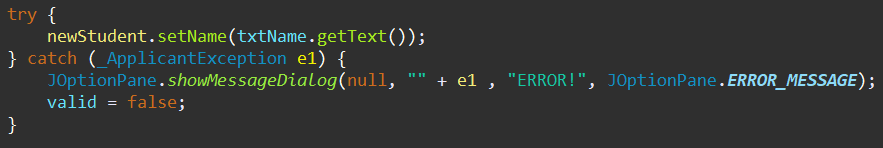
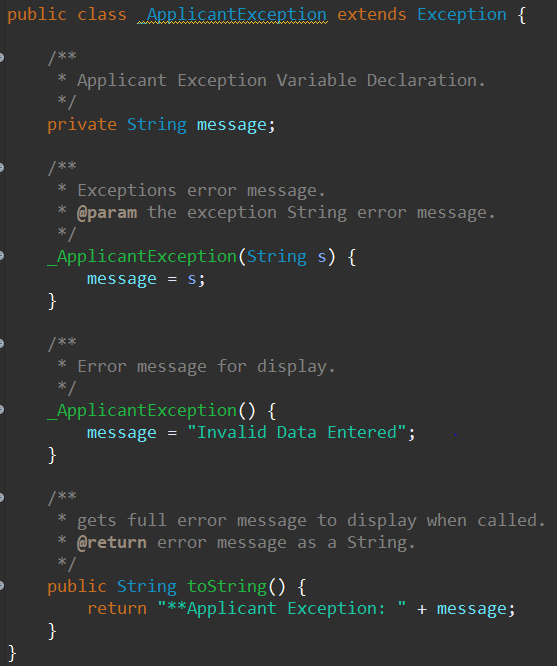


**Students In Method**

**Students Out Method**



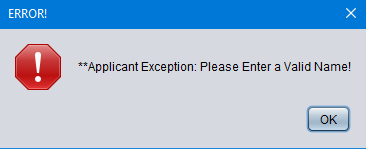
### Appendix 8 – Applicant Exception + Example



**ApplicantException Class**

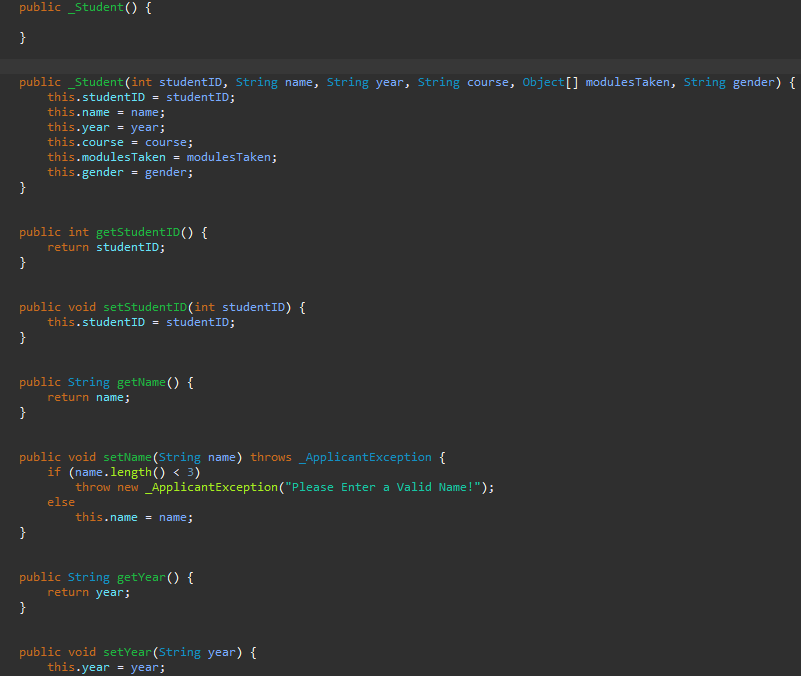
**Student Class**

**Validated within Add Student Form**



**Error Message if error is caught**

### Appendix 9 - Encapsulation



**Partial Capture of Student class**

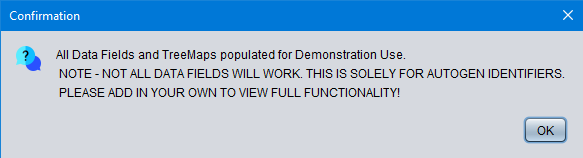
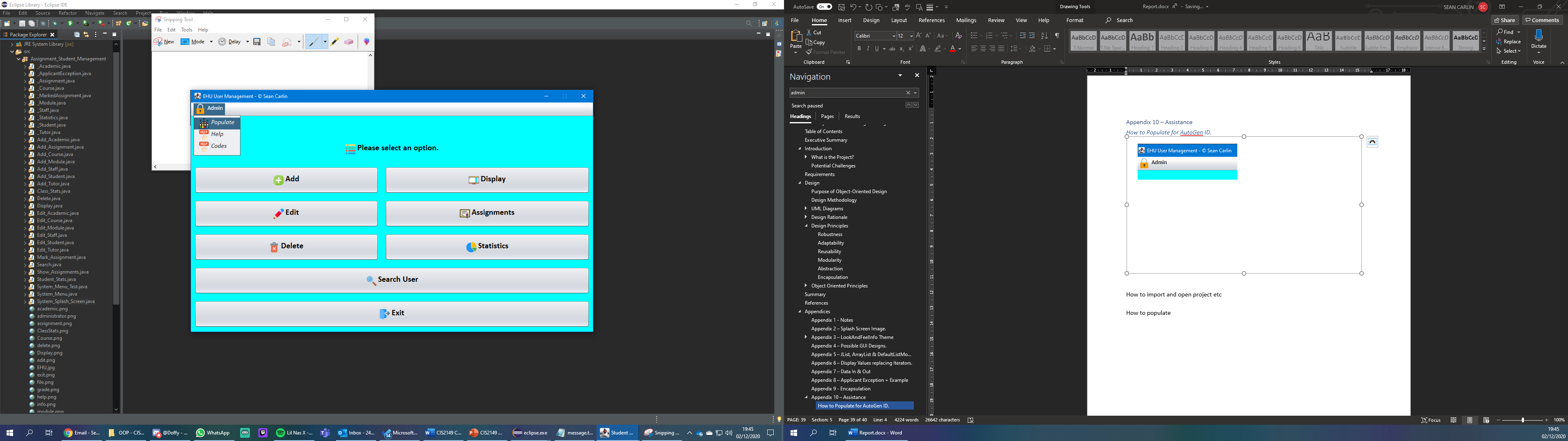
**You can see the class being called and initialized here. Also view the defined attributes getting values assigned.**

### Appendix 10 – Assistance

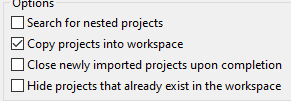
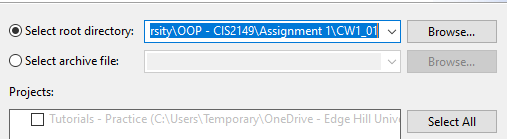
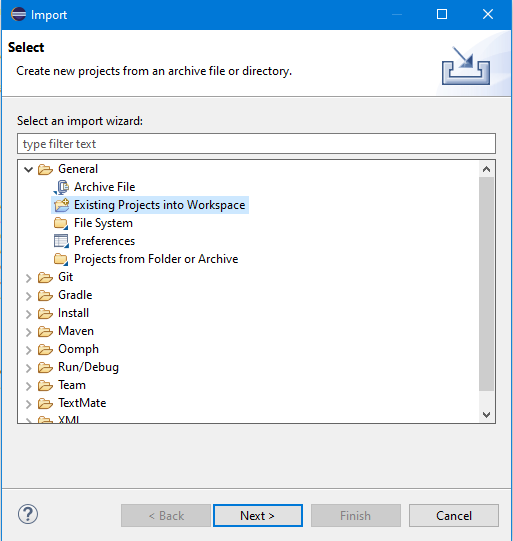
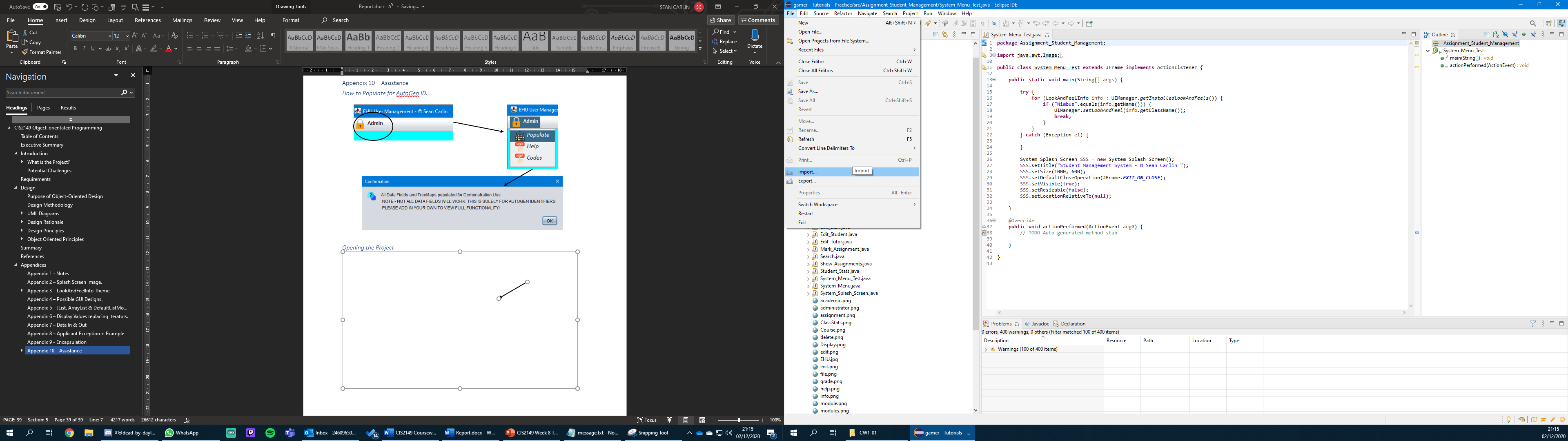
#### JavaDoc API

This project also has a Javadoc API generated. You will find this in the source folder.

#### How to Populate for AutoGen ID.



#### Opening the Project



**Main Class to run = System\_Menu\_Test**

**Upon start-up all data files wont exist, hence the populate method!**