

# Allocation Project Report

Kishan Maharaj | Ross Cathcart

202129516 | 202139210

MEng Electronic and Electrical Engineering

EE273 : Engineering Design For Software Development 2

March 2023

*“We hereby declare that this work has not been submitted for any other degree/course at this University or any other institution and that, except where reference is made to the work of other authors, the material presented is original.”*

**K.C.M**

**R.D.C**

# Contents

|           |  |           |
|-----------|--|-----------|
| <b>1</b>  | <b><u>Abstract</u></b>   | <b>1</b>  |
| <b>2</b>  | <b><u>Introduction</u></b>   | <b>1</b>  |
| <b>3</b>  | <b><u>Project Objectives And User Requirements Specification</u></b> | <b>1</b>  |
| <b>4</b>  | <b><u>Functional Specification</u></b>                               | <b>2</b>  |
| <b>5</b>  | <b><u>Program Design</u></b>   | <b>3</b>  |
| 5.1       | High level system architecture . . . . .                             | 3         |
| 5.2       | Design Patterns . . . . .  | 6         |
| 5.2.1     | Singleton Pattern . . . . .  | 6         |
| 5.2.2     | Strategy Pattern . . . . .   | 6         |
| 5.3       | Data members and methods . . . . .                                   | 8         |
| 5.3.1     | The <i>Abstract</i> User . . . . .                                   | 8         |
| 5.3.2     | The Student Class . . . . .  | 9         |
| 5.3.3     | The Supervisor Class . . . . .                                       | 10        |
| 5.3.4     | The Admin Class . . . . .  | 11        |
| 5.4       | Saving and Loading The Database . . . . .                            | 12        |
| 5.4.1     | Saving . . . . .   | 12        |
| 5.4.2     | Loading and The Builder Pattern . . . . .                            | 13        |
| 5.4.3     | Threading . . . . .  | 13        |
| <b>6</b>  | <b><u>Testing and Results</u></b>                                    | <b>15</b> |
| <b>7</b>  | <b><u>Discussion</u></b>   | <b>18</b> |
| <b>8</b>  | <b><u>Further Work</u></b>   | <b>19</b> |
| <b>9</b>  | <b><u>Conclusions</u></b>  | <b>20</b> |
| <b>10</b> | <b><u>References</u></b>   | <b>20</b> |
|           | <b>Appendix A User Guide</b>   | <b>21</b> |

## 1 Abstract

This report details the design, development and testing of a software application that provides an interactive method for students, supervisors and admin users to participate in project allocation.

The software design solution is implemented using a variety of object-oriented design facilities offered in the C++ language to minimise code duplication and produce an extendable class hierarchy that can be easily modified in the future, should project requirements change.

Inspiration is drawn from the Gale-Shapley Algorithm for stable matching to develop an efficient algorithm to maximise satisfied students receiving one of their top choices, whilst simultaneously minimising supervisors overburdened by a high number of allocated students.

Integration of the class hierarchy with a CSV database, using standard threads to save and load data has proven effective at maintaining the program state between launches in the testing phase of the project which subjected the code to comprehensive unit testing.

While the design solution currently has no graphic user interface and runs on the windows command console, it has proven to be robust at guarding against user-generated runtime errors. The program altogether functions as intended, matching the class hierarchy designated in §5.1.

## 2 Introduction

Manually allocating students to projects in such a way that student preferences are prioritised without overloading the supervisors is a challenge that faces teaching staff in academic institutions across the globe. As such, it would be economical to develop a system to automatically handle this feat.

With this vision in mind, the project detailed in this report will employ object-oriented design principles in an attempt to design a robust, maintainable, and extendable software application to solve this allocation problem. The universal modeling language (UML) provides a method of cataloging the program's constituent classes and how they aggregate to enable the program to allocate projects to students and supervisors alike. Threading will be applied to the larger methods of each class in order to provide a faster and more enjoyable user experience, independent of the size of the program's database. After describing these features in detail and implementing them in the C++ language, the code will be tested in order to meet a prescribed functional specification. Moving forwards, the efficacy of the implemented code -as well as the results of detailed testing- will be discussed at length in order to fully dissect the software's functional components and discover their shortcomings, and ways in which they may be improved.

## 3 Project Objectives And User Requirements Specification

As mentioned above, the general purpose of this design project is to develop software that implements some sort of project allocation system; students using the software are permitted to select these projects and rank them in order of preference. Additionally, each project within the database will be associated with a supervisor, who can also use the program to oversee the students in their

respective project groups. Administrative personnel should also be able to access this program and be granted executive privileges - being able to allocate a project to students from within each of their respective preference lists. The program should assist the admin user group in their decision-making- ensuring that a maximum number of students receive their first preference of project and that no supervisor may possess a project containing an excessive number of students. The user interface of this software must demonstrate several high-level functionalities in order to provide a satisfactory experience to all user groups:

1. The external interface of the program must be polymorphic depending on the end user (i.e., depending if students, supervisors, or admins are accessing the software).
2. Student users saving their project preferences should be able to log into a registration system- matching their choices to an entity in a database. This allows them to return at a later time and configure their choices.
3. All users who store or modify data within the program should be able to log out of the software system and have their data saved. This implies the existence of an external database outside of the main program.
4. All users should be able to view the database of projects but not directly edit its contents. Only users with administrative permissions should be able to directly modify the data pertaining to projects, and the students (or supervisors) allocated to them.
5. The database constructed at the launch of the program, and interacted with throughout, should preserve its own state across launches of the program. In other words, it should serialise any data which is changed and store this appropriately.
6. The algorithm implemented by admin users should optimise the number of students being allocated their most desired projects while minimising the maximum number of students in each project group.

## 4 Functional Specification

Several modifications have been made to the original test and functional specification document (revised March 2023 - EE273 Week 7).

Firstly, the decision was made to **not** allow users to directly create accounts from the main menu. The reason for removing this feature is that it leaves the potential for any existing student user to simply create a new admin account and then have unrestricted access to areas of the database that are not appropriate. Instead, the creation of new users is delegated to an admin user who may create new students, supervisors, and projects. This more accurately reflects a real-life database system such as the model used by Strathclyde University; a student cannot create an account directly, forcing them to consult a trusted admin user to carry out this task on their behalf.

Additionally, as was indicated in section 5 of the functional specification, the decision was made to allow for the creation of only a single admin account. All admins share the same credentials when signing into the system. This simplifies the need to create multiple object instances of the admin class which are only superficially different, saving memory and eliminating the need to serialise the admin class.

## 5 Program Design

### 5.1 High level system architecture

For the purposes of this report, the majority of the project allocation program favours an *encapsulated* design principle. This technique offers object-oriented abstraction to each user (in the form of private classes, which are the principal method of restricting access in C++).

Each data structure represents an entity of a specific class; encapsulation hides the data members of this structure, making them *private* to external client code. Now, only the class methods can retrieve or mutate these data members as the programmer sees fit - forming a well-controlled interface around each complex data object in the program. The user does not need to care about the internals of these classes as the peripheral functions (methods) raise the level of abstraction, providing a simple API to interact with the internal class data.

Figure 1 shows the top-level UML class diagram for the group project allocation program. Note that this class diagram is incredibly detailed, showing all the associated class methods and data members. As such, Figure 2 simplifies the relationships from the larger diagram, highlighting only the key features of the program's general design - namely the object relationships.



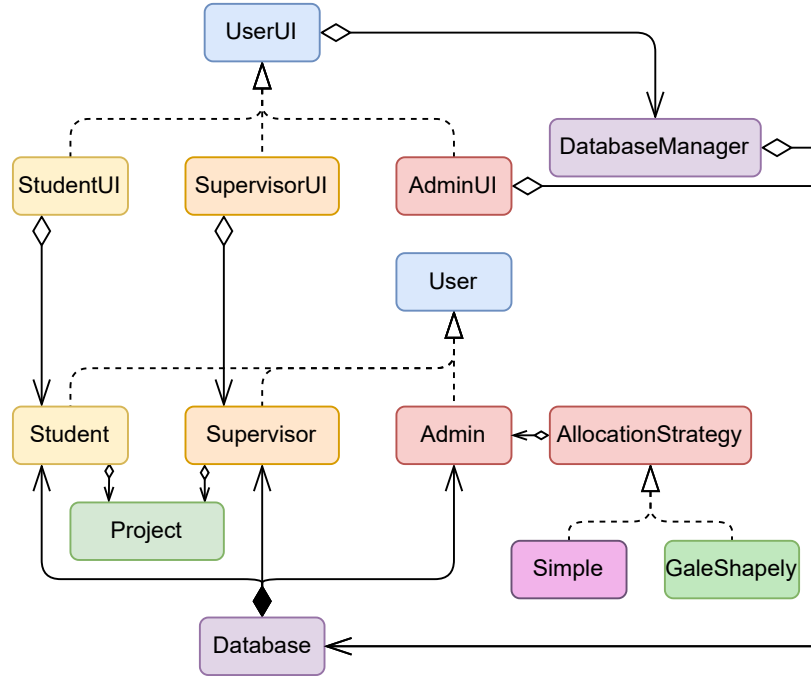


Figure 2: Simplified UML class-diagram of group project database objects

As previously mentioned, there exist three main user bases: students, project supervisors, and administrators. All of these groups may need to use the software for different purposes; as such, it would be prudent to provide different user interfaces (UIs) for each demographic. Each of these interfaces must be associated with the corresponding user (i.e, possessing a pointer to the associated user object), in order to enter and modify the user's information in a database. This database acts as a central storage facility for the objects, managing their lifetime and external access.

Just as each UI element is associated with a student, supervisor, or admin object, these entities are then associated with the database via a bi-directional relationship. Within the database class are smaller databases; vectors (dynamically allocated, resizable arrays), each holding ordered containers of associated students, teachers, and projects respectively. Note that these are merely *associated* with the vector (i.e, the vectors contain pointers to each object), as the memory for each of these elements is allocated from the heap. This is because the database must be loaded from a \*.csv file - from which it was previously saved to. In doing so, **new** pointers to each object must be created, as the objects themselves cannot be destroyed and must exist for the program's entire lifetime. The database class is responsible for managing the life cycle of its constituent classes. To comply with RAII principles means that the database must free all dynamically allocated memory when the program terminates, and this class is destroyed.

From the perspective of the user, another entity must facilitate the transfer of information between the database and the user's UI, acting like an API between the classes. This **Database Manager** entity shall provide utility functions to the user, allowing students to locate, view and select group projects. This functionality, however, must also be strictly controlled by the database manager - for example, students should not be able to directly access every project in the database. Rather, they should only have access to those projects which are members of their list of project preferences. The decision to define a manager class for the database was made to permit the decoupling of the core data and functionality of the database from the way in which classes interact with it. This

makes the code more modular and allows for easier testing.

Class inheritance, courtesy of object-oriented programming design principles, enables repeating data structures to be shared from *base* classes to *derived classes*: code can be recycled instead of simply repeated. All user classes have either some data members or methods in common such as a name, password, and the corresponding functions to set and retrieve these objects. These members can therefore be inherited from an *abstract* base class (one which has no instances, containing pure virtual functions used by instances of derived classes). This base class is the 'User' entity, from which the supervisor, student, and admin classes are derived. Similarly, the different student, supervisor, and admin UIs can be derived from an abstract `UserUI` class.

## 5.2 Design Patterns

When designing software, the engineer will often encounter repeating design challenges. For example, how can tightly coupled classes be separated? Or how can the programmer enforce the construction of only one instance of an object?

Fortunately, 'Design Patterns' exist to aid the programmer in tackling these issues. This code employs two of such patterns introduced in the book Design Patterns: Elements of Reusable Object-Oriented Software (1994). Namely, a modified singleton pattern and the strategy pattern.[1]

### 5.2.1 Singleton Pattern

The Singleton design pattern necessitates the creation of only a single instance of an object in a program at any given state. This is traditionally achieved by making both the copy and assignment constructors private members of the class, prohibiting the client code from unintentionally creating further instances or copies of the object. Static member functions allow the single instance to be accessed. For the Database class in this program, it would be ideal to enforce only a single instance of this to prevent dependent classes from modifying different copies of the Database, thus leading to synchronisation issues.

However, the standard implementation of the Singleton pattern has a major flaw in the context of this code: it essentially makes the core Database a global variable. This means all classes and objects can access, modify, and manipulate the underlying data with little to no restrictions imposed. This is a poor design choice and would defeat the purpose of the Database Manager class which provides *restricted* access to the Database. As such, a slight modification of the Singleton pattern was made. To eliminate the global scope of the Database, the static `getInstance()` method was removed. The copy and assignment constructors were still made private, but there was now no longer a way of forcing only one instance via regular construction - this was now left up to the responsibility of the programmer. This compromise still brings some of the benefits of the original design pattern.

### 5.2.2 Strategy Pattern

The brief for this project states that "you should consider in your implementation the possibility for additional algorithms being added at a later date" with regards to the allocation approach offered by the program.



Due to time constraints in the project, it is likely that despite the best efforts of the developers, the current selection of algorithms offered are not entirely optimised. The time complexity was considered when developing the Gale Shapely inspired algorithm, with an  $O(n^2)$  algorithm being achieved. However, it is possible that an even more efficient algorithm exists.

As such, the Strategy Design pattern was employed to make the code flexible and easily extendable in the future should further algorithms be added. This design pattern achieves this goal by separating the `Admin` class from the `AllocationStrategy` class. Each admin contains a pointer to an abstract base class of this type, from which concrete implementations such as `galeShapely` are derived. In such a way, the class has no knowledge of the implementation details for the allocation class - only the promise that one exists. This approach almost completely decouples the two classes and means that future algorithms can be added without needing to recompile or update the `Admin` user class. This relationship is illustrated in Figure 2. Adding a new algorithm would mean another class inherits from the `AllocationStrategy` class in this diagram. Visually, it is obvious that this would have very little impact on the rest of the classes in the hierarchy.

Code listing 1 shows the code implementation of the Strategy Pattern. The abstract base class `AllocationStrategy` contains a single pure virtual function, `allocate()`, which the concrete derived classes are forced to override. This means that a pointer to a base class object can polymorphically deduce the object type and call the correct `allocate()` override method.

```
1  class AllocationStrategy {
2  public:
3      enum Strategy {
4          GALESHAPELY,
5          SIMPLE,
6      };
7
8      virtual void allocate(Database* db) = 0;
9  };
10
11 class galesShapely : public AllocationStrategy {
12 public:
13     void allocate(Database* db) override;
14 };
15
16 class simpleAllocate : public AllocationStrategy {
17 public:
18     void allocate(Database* db) override;
19 };
```

Listing 1: Strategy Pattern in C++.

## 5.3 Data members and methods

### 5.3.1 The *Abstract* User

As previously mentioned, the `User` class is abstract. It is a base class containing both methods and data members common to all derived child classes. With this in mind, it would be useful to identify the common methods and attributes to all user types, then place these in the abstract class to provide the greatest level of code re-use.

Tables 1 and 2 detail the aggregate functions and variables of the user. To log in, all users must have a name and password. To further differentiate users with similar characteristics, an ID variable can also be implemented - this may help with quickly finding the user's information within the database, or reconstructing it when reloading the database from memory. These attributes are controlled by methods. Each student must have a default constructor. Additionally, another constructor can be utilised, taking argument variables and pasting them into the member fields of the created class object. For the purposes of encapsulation, it would also be useful to possess "interface" functions that set and retrieve these members; these are called *setters* and *getters*.

Table 1: Methods of the User base class.

| Function  | Description               |
|---|---------------------------|
| <code>User()</code>                                     | Default constructor       |
| <code>User(string name, string password, int id)</code> | Parameterised constructor |
| <code>~User()</code>                                    | Virtual Destructor        |
| <code>string getName()</code>                           | Retrieves user's name     |
| <code>string getPassword()</code>                       | Retrieves user's password |
| <code>int getID()</code>                                | Retrieves user's ID       |
| <code>void setName(string name)</code>                  | sets user's name          |
| <code>void setPassword(string name)</code>              | sets user's password      |
| <code>void setID(int ID)</code>                         | sets user's ID            |

Table 2: Data Members of the User base class.

| Data Member                   |
|-------------------------------|
| <code>string full_name</code> |
| <code>string password</code>  |
| <code>int myID</code>         |

Note that within this class declaration, all of the methods are made public, and the data members are private. In doing so, no entity can simply reach inside of an object and change its properties; the methods must act as a controlled interface to do so, adding protection and additional functionality.

### 5.3.2 The Student Class

The Student class is derived from the User class and therefore contains all of the data structures detailed in tables 1 and 2. Of course, this class must contain its own unique methods and data -illustrated in tables 3 and 4; as is the purpose of the program, each student possesses a vector of pointers to projects they would like to be allocated to. The decision to avoid a composition relationship was made so that the Student class does not manage the lifetime of the project. Additionally, an association allows many students to associate with the same project.

Only one project can be allocated, so students also contain a pointer to this project; if it has yet to be allocated, then the pointer is *null*. It is also practical for the class to contain the name of the student's degree for differentiating between similar members of the database.

Table 3 provides the getters and setters for the student's degree, allocated project, as well as the project vector. There are new miscellaneous functions here, however. Some of these methods add and remove project pointers to the array of preferred projects, while others locate each project within the vector.

The Serialise function is very important, as it stores all of the student's data in string form: now this aggregate object can be represented as memory outside of the program. This string can be inserted into a file (CSV, XML, etc.) when the database of students is saved. Similarly, one constructor for the student class takes a string as an argument; this is to reload the saved string into the database.

Table 3: Methods of the Student class.

| Function   | Description              |
|--|--------------------------|
| Student(string name, string password, int id, string degree) | User constructor         |
| Student(const string& cvslne)                                | CSV constructor          |
| virtual ~Student()   | destuctor                |
| string Serialise()   | save to string           |
| void displayAllocatedProject()                               | print project            |
| void displayMyProjectChoices()                               | print choices            |
| vector<Project*> getMyProjectChoices()                       | return project vector    |
| string getDegree()   | return degree            |
| void setDegree(string degree)                                | sets degree              |
| Project* getAllocatedProject()                               | returns project pointer  |
| Project* findProject(Project* to_find)                       | return pointer in vector |
| Project* findProject(string project_name)                    | find pointer by title    |
| void addProjectToPreferences(Project* project)               | push back vector         |
| void setAllocatedProject(Project* to_allocate)               | set allocated            |
| void removeProjectFromPreferences(Project* to_remove)        | erase from vector        |
| void removeProjectFromPreferences(string to_remove)          | erase by title           |
| bool hasProject(string project_name)                         | check for project        |
| int getAllocatedIdenifier                                    | get associated ID        |
| vector<int>& getPreferenceIdentifiers                        | get vector IDs           |

Table 4: Data Members of the Student class.

| Data Member                        |
|------------------------------------|
| vector<int> preference_identifiers |
| int allocated_identifier{ 0 }      |
| string degree                      |
| Project* allocated{ nullptr }      |
| vector<Project*> projects_choices  |

Note that the student has a vector of identifiers used to locate projects in the database.

### 5.3.3 The Supervisor Class

The Supervisor class contains similar methods (table 5) to the Student class; this is because their data members (table 6) are similar, but, are used for different purposes in the user interface. These supervisors are also associated with a vector of projects which they oversee. They also belong to a department - the name of which is contained in a string. To access this class, the necessary getters and setters for these objects are listed in table 5, alongside some more exotic functions. As with the student, there are CSV-string constructors and CSV-string serialiser functions for saving and loading the supervisors into non-volatile memory. There are also additional methods that are used to add and remove projects from their respective vectors.

Table 5: Methods of the Supervisor class.

| Function  | Description        |
|---|--------------------|
| Supervisor(string name, string password, int id, string department) | user constructor   |
| Supervisor(const string& csvline)                                   | CSV constructor    |
| virtual ~Supervisor()   | destructor         |
| string Serialise()  | save to string     |
| void setDepartment(string department_name)                          | assign department  |
| void addProjectWorkload(Project* project_to_add)                    | push back vector   |
| void removeProject(Project* to_remove)                              | erase from vector  |
| string getDepartment()  | return department  |
| vector<Project*> getProjectsOversee()                               | return vector      |
| vector<int*> getProjectIdentifiers()                                | return project IDs |

Table 6: Data Members of the Supervisor class.

| Data Member  |
|--|
| <code>vector&lt;int&gt; project_identifiers</code>   |
| <code>string department</code>                       |
| <code>vector&lt;Project*&gt; projects_oversee</code> |

As with the student class, the supervisors also have a list of project identifiers used to locate them when the database is reconstructed from system memory.

#### 5.3.4 The Admin Class

The primary function of an administrator is to control the group project allocation process: this user can view the database and make edits to the data held by other class objects. As such, the Admin class itself possesses only one primary data member. This member is a utility class containing the preferred functions for allocating projects between the students. The methods of this class (table 7) are simply the getters and setters for this data.

Table 7: Methods of the Admin class.

| Function   | Description |
|--|-------------|
| <code>Admin()</code>   |             |
| <code>Admin(std::string name, std::string password, int id, -<br/>AllocationStrategy::Strategy strat)</code> |             |
| <code>virtual ~Admin()</code>  |             |
| <code>void setAllocationStrategy(AllocationStrategy* strategy)</code>  |             |
| <code>void setAllocationStrategy(AllocationStrategy::Strategy strategy_type)</code>                          |             |
| <code>AllocationStrategy* getAllocationStrategy()</code>   |             |
| <code>string getStratID()</code>   |             |

Table 8: Data Members of the Admin class.

| Data Member   |
|---|
| <code>string strategy_identifier</code>                       |
| <code>AllocationStrategy* allocate_strategy{ nullptr }</code> |

## 5.4 Saving and Loading The Database

### 5.4.1 Saving

When saving the database, all existing entities and their complex relationships must be serialised into a form of information that can be stored in a non-volatile file type. This program uses the CSV file format. To serialise the data, each student, supervisor, and project are converted into strings. Each string contains the attributes of the class object separated by commas. Examples for the class serialise strings are shown in Table 9 :

Table 9: Possible CSV strings

| User       | String   |
|------------|--|
| Student    | name,password,ID,degree,allocated,[preference list]              |
| Supervisor | name,password,ID,department,[overseeing list]                    |
| Project    | title,module code,description,capacity,supervisor,[student list] |

Listing 2 Shows how these strings are saved into their respective CSV files: (note that this is a rudimentary saving method, and is to be improved in §5.4.3).

```

1  void Database::saveDBtoCSV() {
2      std::ofstream StudentStream("Student.csv");
3      std::ofstream ProjectStream("Project.csv");
4      std::ofstream SupervisorStream("Supervisor.csv");
5
6      for (auto& n : this->getStudents()) {
7
8          StudentStream << n->Serialise() << "\n";
9      }
10     for (auto& n : this->getProjects()) {
11
12         ProjectStream << n->Serialise() << "\n";
13     }
14     for (auto& n : this->getSupervisors()) {
15
16         SupervisorStream << n->Serialise() << "\n";
17     }
18 }
19

```

Listing 2: Saving to CSV.

The data of Students, Supervisors, and Projects must be sorted into three different \*.CSV files: as such three different output file streams are opened to the respective *Student.csv*, *Supervisor.csv* and

*Project.csv* files. A for-range loop is run through the database’s vectors of Students, Supervisors, and Projects; each of these class objects is serialised and fed into the correct output stream.

### 5.4.2 Loading and The Builder Pattern

To load the database, the strings must be deserialised into regular class objects. But how is this achieved? All of the database classes are associated: for a project to be created, it must be associated with students and supervisors which do not yet exist. Similarly, new students may be linked to non-existent projects. The *builder* design pattern [1] solves this dilemma. By fragmenting the construction of objects in the database, these objects can be first instantiated, and then their associations can be mapped when all other database entities have also been created. As with the saving method, input file streams are created from each CSV file. As loops iterate through each file, the CSV string constructor is called for each class, and they are added to their respective vectors in the database. This constructor simply creates each object with non-pointer data types (such as the registration number). When all objects have been constructed, a second loop is commenced for each class, reading the second part of each string pertaining to the name or ID of their associated projects, students, or supervisors. These objects are then located in the database entity and pointers are “linked” to the correct objects.

### 5.4.3 Threading

It is clear that saving and loading the database in this fashion is very time and resource intensive. Additionally, the program must wait for all of the related sub-processes to execute sequentially. A degree of parallelism would allow each file to be loaded into each vector concurrently; the C++ `std::thread` library supports threading to run functions at the same time, merging them into the primary thread after they execute and continuing with the main function. Listing 4 provides a sample of parallelism within the program. The database is once again saved. This time, however, a thread is created for each loop. The `std::thread` class takes a function pointer in its constructor. This function is then executed within the created thread. At the end of its lifetime, the thread joins back into the “pool”. The `.join()` function waits for each thread to terminate into the pool before continuing.[2] Instead of pointing to a named function, a standard lambda function of each loop takes the input argument of each thread. This simplifies the code, as the thread can just run the already existing code. Listing 3 shows an example:

```
1 ([&](args){doSomething;});
```

Listing 3: a Lambda function

The square brackets ([ ]) describe the lambda’s capture clauses (what is externally passed into the function); using an ampersand (& ) as a capture statement passes all external variables into the function by reference, allowing objects from the above scope to be modified. [3]

```

1  void Database::saveDB_THREAD() {
2      std::ofstream StudentStream("Student.csv");
3      std::ofstream ProjectStream("Project.csv");
4      std::ofstream SupervisorStream("Supervisor.csv");
5
6      std::thread saveStudents([&] {
7          for (auto& n : this->getStudents()) {
8              StudentStream << n->Serialise() << "\n";
9          }
10         }
11     );
12     std::thread saveProjects([&] {
13         for (auto& n : this->getProjects()) {
14
15             ProjectStream << n->Serialise() << "\n";
16         }
17     });
18
19     std::thread saveSupervisors([&] {
20         for (auto& n : this->getSupervisors()) {
21
22             SupervisorStream << n->Serialise() << "\n";
23         }
24     });
25
26     saveStudents.join();
27     saveProjects.join();
28     saveSupervisors.join();
29 }

```

Listing 4: Threaded-Saving Method

An identical approach is taken to loading the database- although its intricacies are too great to be illustrated in this report. Three threads run in parallel to create the base instances of each object. The threads are then joined: in simple terms, the program waits for each student, supervisor, and project to be constructed before continuing. After this, three new threads are created to map associations between the objects, before being joined again. After then the program has reached the end of its lifetime, it must be saved, and the threading process above is reattempted to save each object to its respective CSV file. Figure 3 shows the control flow for these processes.



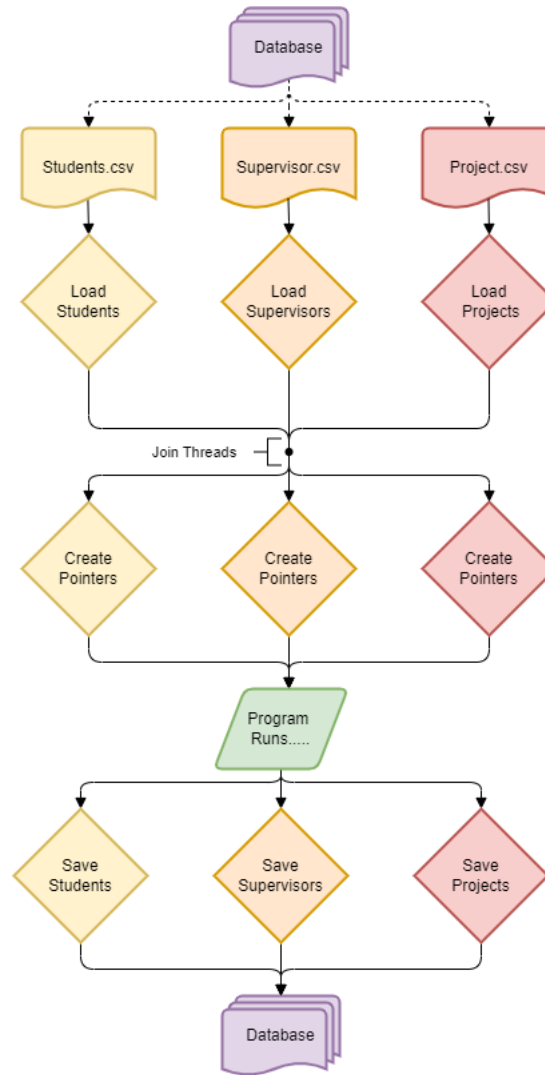


Figure 3: Database Control Flow

## 6 Testing and Results

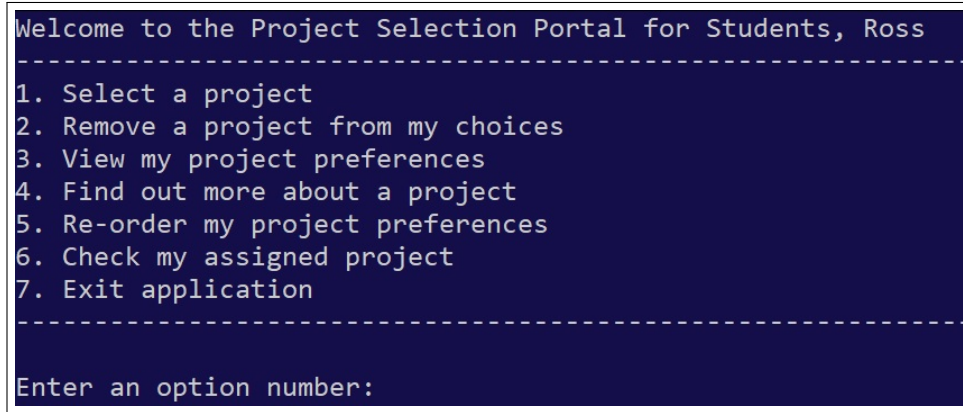
Referring back to the test specification completed in week 7, four key areas for testing were identified.

1. Logging in as an existing user
2. Managing appropriate permissions for each user
3. Efficacy of sorting algorithms
4. Maintenance of program state

To test area 1, a number of mock accounts for each user type were created. By following the login steps detailed in Appendix B, it was confirmed that the program is able to successfully retrieve the data for each user and match this up with the correct derived class type. The

`authenticateAndFindUser()` method searches each user type database for a matching user and returns a pointer to this type if it is found or a `nullptr` if there is no matching user. This strategy ensures that the login portal also successfully *prevents* those without valid credentials from progressing.

To tackle area 2 identified in the test specification, the program naturally manages the permissions the user has by offering a menu similar to the one shown in Figure 4. For each of these options, the user interacts with the **Database Manager** class rather than directly with the **Database** class. This API class acts as a bridge between the sensitive underlying data in the **Database** and the user of the software system, restricting access so that users cannot modify unauthorised data.



```

Welcome to the Project Selection Portal for Students, Ross
-----
1. Select a project
2. Remove a project from my choices
3. View my project preferences
4. Find out more about a project
5. Re-order my project preferences
6. Check my assigned project
7. Exit application
-----
Enter an option number:

```

Figure 4: Access control by enumerated options

Area 3 of testing ensures that the sorting algorithms written fulfill the specification of maximising the number of students receiving one of their preferred project choices whilst minimising the workload of supervisors. To test this functionality, a vast assortment of mock projects, supervisors, and students were created.

The student csv strings were appended with project IDs to replicate the scenario where every student has selected  $n$  projects, indicating their preferred choices. Following this, the system was accessed as an admin user to perform automatic allocation. The list of allocated projects was then compared to those given in the preference list of each student to ensure that the algorithm was accurately distributing projects to reflect the preference of students and leaving no students without allocated projects.

In addition to this, it was verified that the algorithm was not overfilling projects (in other words, the maximum capacity attribute of the projects was successfully preventing overflow of students), as this would cause overburdening of supervisors.

The given project allocation for each student can be viewed from the admin interface, producing an output similar to the one shown in Figure 5. This makes it easy to spot students who may not have any allocations. Furthermore, the allocations can be edited manually by admins by interacting with a desired student object to swap out the allocated project attribute. The admin can run the allocation process many times after manually seeding through to pre-set a student's allocated project, offering a degree of flexibility and control.

```

Current Allocations
-----
Kelly(Politics) : Projects and Students
Ross(EEE) : Projects and Students
Benjamin(EEE) : Projects and Students
Arthur(EME) : Chess Engine
Thomas(EEE) : Chess Engine
George(EME) : Chess Engine
Harry(CES) : Chess Engine
Isabel(BME) : Machine Learning
Jake(Maths) : Machine Learning
Liam(English) : Machine Learning
Megan(Law) : Stock Analysis
Nancy(Chemistry) : Stock Analysis
Olivia(EME) : Stock Analysis
Peter(CES) : Stock Analysis
Quentin(BME) : Stock Analysis
Rachel(Maths) : Social Network
Sarah(Politics) : Social Network
Thomas(English) : Social Network
Ursula(Law) : Language Translator
Victor(Chemistry) : Language Translator
William(EEE) : Language Translator
Kishan(EEE) : Language Translator
Ethan(Biology) : Language Translator
Ava(English) : Language Translator
Charlotte(Philosophy) : Language Translator

```

Figure 5: Sample console output with allocation

This process was iteratively carried out for each allocation strategy to ensure the correctness of operation for all algorithms provided.

The final major area to test was the maintenance of the program state between launches. This means that any data entered by one user should be saved when the program closes, and changes they make to centrally stored data in the database should be reflected in the UI when another user launches the program in the future. To provide an example, if an admin user signs in and edits the name of a pre-existing project then when a student later logs into the system they should now see the changed project name when viewing the list of available projects. This was proven true by signing into the database as each different type of user and modifying some piece of data, then checking that this modification was both:

- Visible to the other user types (via their respective interfaces)
- Visible to the mutator of the data upon signing back into the database

Both of these criteria were satisfied for all classes so this test area can be concluded to have passed with its correctness of operation verified to a high degree of confidence.

Lastly, at a lower level than the areas identified in the test specification, testing was conducted to ensure validation of user input at every opportunity the user has to provide `cin` data to the program. This was handled by writing `getValidInteger` and `getValidString` functions to ensure that the data entered by the user is of the correct type, clearing the buffer if a fail flag is raised thus preventing bad data from entering the database. A sample validation function for retrieving integer data is shown in Listing 5.

```

1  int getValidInteger(const std::string message) {
2      int input;
3      bool valid_input = false;
4      while (!valid_input) {
5          std::cout << message;
6          std::string strinput;
7          std::getline(std::cin, strinput);
8          std::stringstream ss(strinput);
9          if (ss >> input) { /
10             valid_input = true;
11         }
12         else {
13             std::cout << "\nInvalid input. Please enter a valid integer."
14             << std::endl;
15         }
16     }
17     return input;
18 }

```

Listing 5: A validation function for integer data

## 7 Discussion

The overall design pattern of this program offers a plethora of versatility and code reusability. The encapsulation and polymorphism between each user class allowed their behaviour to be separately and independently managed by a number of routine subprograms, minimising the chances of erroneous code while obfuscating insignificant, low-level data. Constructing the relationships between these classes as pointers and references ensured that no copies of an object were created, minimising memory wastage.

Furthermore, creating an abstract interface between both users and the database, and administrators with their preferred allocation strategies provides a means of extending, modifying or refactoring code without directly affecting these classes. Additional features of this database greatly improve its ability to grow at a large scale. Holding the program data in nonvolatile memory such as a CSV file allows gigabytes of data to be stored and quickly accessed. Dynamically loading this information into the heap further improves this access time- although this must be done with complex and extensive functional programming. Here, the functional methods to both extract and save database information have been designed very effectively.

Traversing the database using range loops provides a satisfactory  $O(n)$  time, although using algorithms from the standard template library may offer improved time complexity. Decomposing steps of this process into threads may improve the program's efficiency; many processes run simultaneously without waiting for more resource-intensive subprograms to execute. However, there exist difficulties and inconsistencies when using high degrees of parallelism. As the database grows in size, the file size of each CSV may grow, and other CSV files may be necessary - with more additional threading. There is an issue here: as many threads interact with a shared (or even *global*

variable simultaneously, the probability of random behavior occurring increases drastically. This is called a data race and is a common problem of multi-threading with databases. The `std::atomic` library introduces new thread-safe data types capable of exhibiting precise and well-defined behaviour in the condition of a data race; these atomic types may be useful if the database grew very large. [4]

The use of an almost exclusively OOP style in this project brings its own problems - there is significant coupling between the user and respective UI entities. This is due to the size of the interface between each module. The implication of high coupling is that the codebase as a whole will be difficult to modify when errors occur in one of the two primary classes, making the code less reusable. Coupling is a reality of object-oriented programming, but its impact could be reduced by further modularising the UI classes. This would increase the cohesion between the UI and user classes while decreasing the interface size of each entity. [5]

## 8 Further Work

In developing this application, the decision to not use a graphical user interface (GUI) was made to allow more time to be spent on the back-end stability and in designing an effective database model, rather than making the software look visually appealing.

As such, the aesthetics of the program (and user experience) suffer due to a lack of familiar features such as buttons and field forms for entering data. An obvious improvement to the project is therefore to use a library such as QT to develop a GUI for the application; the modularity of the existing code developed and avoidance of tight coupling should make this task feasible without too much refactoring to the existing code.

In addition to the graphical limitations of the program, the algorithms used to perform the allocation process may not be completely optimised. If more time was available then the program could be improved by adding further algorithms to offer better performance. To accommodate for this, the Strategy Pattern was used to provide easy extendability to the code. As was previously mentioned, the only major change which would need to be made to implement any new algorithm is to create a new concrete implementation of the “Allocation Strategy” class. This makes the code easy to modify in the future to support new updates.

Another potential improvement that could be made to the project is to connect to an external database tool, such as Azure SQL, to store the data (such as student and supervisor records). This offers the benefits of being more rigorously tested than the bespoke database class written in this project. Additionally, Azure being a cloud-based storage solution means that the scalability of the database would be far better than the current implementation which uses *csv* files to store the metadata between launches. Using a tool like SQL means that the inbuilt functions such as `SELECT` can be used instead of the custom `find` functions written in this program. Once again, the SQL functions are likely better tested than custom code developed to replicate this functionality.

## 9 Conclusions

Designing, developing, and debugging this project has provided valuable experience in the entire software lifecycle and has emphasised the importance of writing code in a maintainable style.

By carefully considering object interactions and relationships, this project has highlighted the practical implications of composition vs aggregation with respect to the management of an object's lifetime. It was critical in this application for the Database to have a *composition* relationship to its internal data, but for all other classes to *associate* with the database itself. This prevents these other classes from destroying the database when they go out of scope which would be detrimental to the operation of the system.

Furthermore, the challenges faced when designing the class hierarchy to best support future modifications raised the usefulness of applying pre-existing software design patterns to tackle these challenges. The Strategy Pattern was employed to allow the program to support the easy addition of other allocation algorithms in the future, and a modified Singleton pattern was also used to manage the dangers of copying the central database. Without re-inventing the wheel, these patterns help software developers to employ tried and tested solutions to address common design issues.

In designing algorithms to save and load the database, questions regarding efficiency were raised. The initial algorithms sequentially loaded/ saved each user base which was slow and glaringly inefficient. To tackle this flaw, multi-threading was employed to parallelise the control flow and allow these actions to occur concurrently, thus drastically improving their efficiency.

On a final note, the development of the code has been beneficial in fostering good software-development habits by introducing the developers to the use of the industry standard tool for version control - Git and Github. Whilst the learning curve was steep initially with an abundance of merge conflicts, the merits of using such a tool became evident when both developers could seamlessly switch between working at home and in the lab on the computers there. Additionally, it made keeping up to date with current changes in the codebase easy and encouraged better collaboration.

Overall, this design project has solidified key OOP concepts which were introduced in semester one and demonstrated the usefulness of C++ in designing general-purpose software applications.

## 10 References

### References

- [1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
- [2] B. Stroustrup, "*std::thread*." <https://cplusplus.com/reference/thread/thread/> (2000). Accessed: 11-03-2023.
- [3] Microsoft, "*Lambda expressions in C++*." Available at <https://learn.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp> (20/02/2023). Accessed: 11-03-2023.

- [4] N. McNie and B. Baumann, “*std::atomic*.” Available at <https://en.cppreference.com/w/cpp/atomic/atomic> (2004 - 2008). Accessed: 17-03-2023.
- [5] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall/Pearson Education, 1997.

## Appendix A User Guide

When the program is initially launched, the user is presented with the screen shown in Figure 6. The process to log in as a student, supervisor, or admin user is identical. The program will deduce the correct user type and then display the corresponding UI.

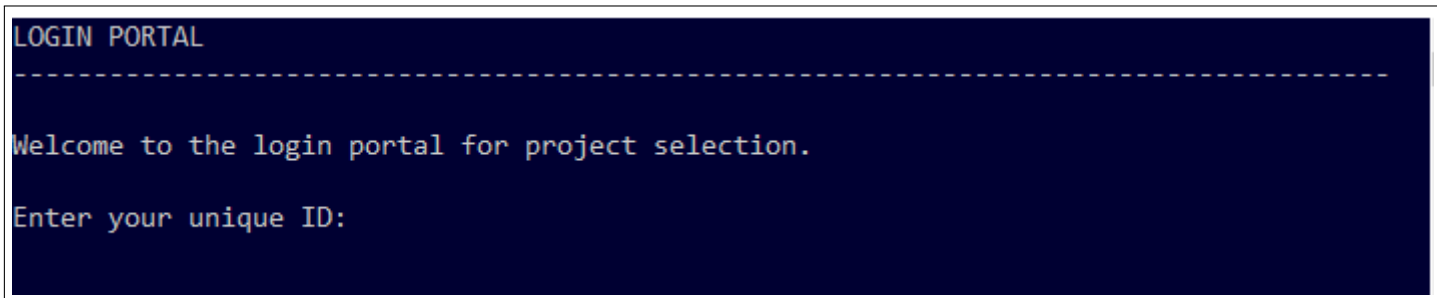


Figure 6: Login portal for users

When prompted, enter the unique user ID and associated password. The program will then search for the user in the database and retrieve the object if it exists - otherwise, there will be an informative error code displayed to let the user know that the user could not be found with matching credentials.

If the login is successful, then the program will branch to display a different UI for each user type. These reflect the different permissions of the user types. The user can then enter the number corresponding to the action they wish to perform. The login screen for the Admin user type is shown in Figure 7.

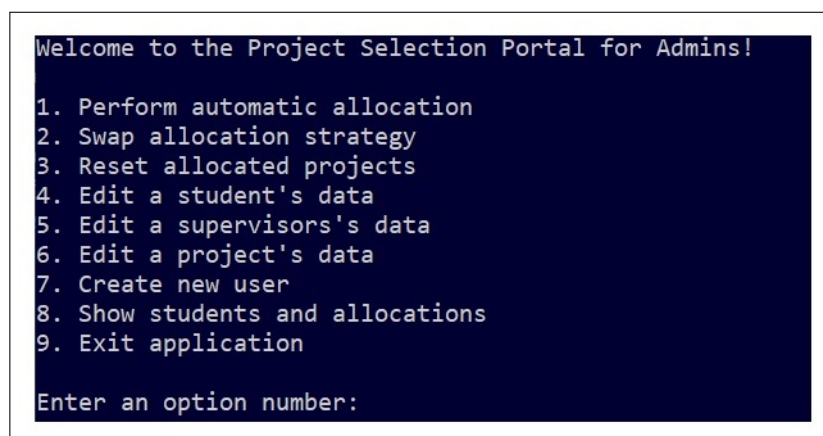


Figure 7: UI menu (Admins)

Each action will then clear the screen to provide another set of options to the user (again, using enumerated choices) to guide the user and receive valid input to guide the flow of the program. Once again, entering a valid input (which is checked using a `getValidType()` function in the utility library) causes the program to branch and display further choices to the user.

This repeats until the user selects the option from the main menu to exit the program. This saves all current data, which is then reloaded on subsequent launches.