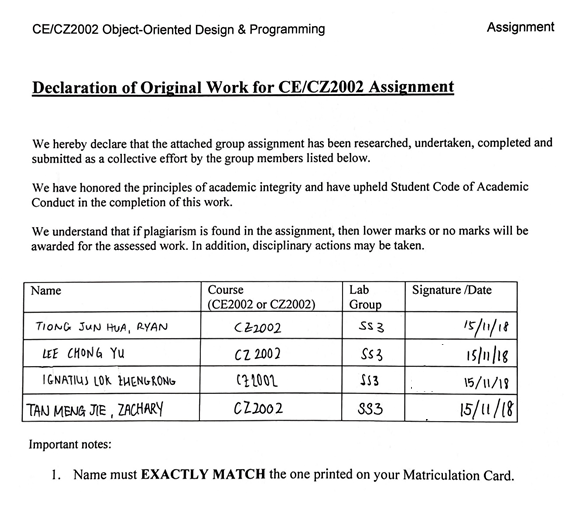
****

**Design Considerations and Object Oriented Concepts**

We designed SCRAME with the main objective of avoiding bad design principles such as rigidity, fragility and immobility. Each class is designed with a dedicated purpose to have high reusability, extensibility and maintainability.

1. Well-defined Cooperating Classes

By following the Entity, Control, Boundary modeling in the design phase, Entity Classes will contain persistent information tracked by the program while Control Classes will implement logic to realize use cases and Boundary Classes will be the interaction between the user and the program. This achieves the above aims and loose coupling as an effect. With loose coupling, any future changes to the program would be minimized.

2. Encapsulation

As the classes must work together cohesively, it is necessary to encapsulate the data within the classes so that each class does not need to know the implementation of the others. The classes working together are tagged with a data access level to prevent illegal access of other class attributes. As such class attributes are mainly declared as private together with method implementations. Constructors and get, set methods are declared as public to allow the cooperating classes to manipulate attributes in each other where required.

3. Polymorphism

The use of method overloading whereby a method can be called with different arguments is a form of static polymorphism, is done at compile time to decide which method to use. We used method overloading for the boundaries which may be called by the main controller or the sub controller to accept input or provide output to the user. As such we do not have to rename the different methods when they are generally doing the same function. This maintains the idea of the boundary still having a dedicated responsibility of providing input and output between the user and the program for a specific controller. Also, the use of dynamic polymorphism when implementing the serialization means that the compiler only decides which methods to use at runtime since different class may require different implementation of the serialization.

4. Maintenance

To achieve better maintainability, we attempted to simplify the responsibilities of the classes to be more focused. For example, we created a separate register for registration record (Register.java), instead of storing an arraylist of students in a course and vice versa. Not doing so would result in a bi-directional association that would allow students to hold outdated course objects which would not have the latest arraylist of students registered. As such the solution would be the use of a course register entity to store the registration record and result of a student to a course.

We chose to maintain a central arraylist in the main controller which can be used by all boundary and control classes that are called from there. By doing so, we will not have difficulty maintaining the data integrity of the arraylist as it is parsed to different controllers since the main controller copy of the arraylist is always updated after a boundary and control have completed their sequence of actions.

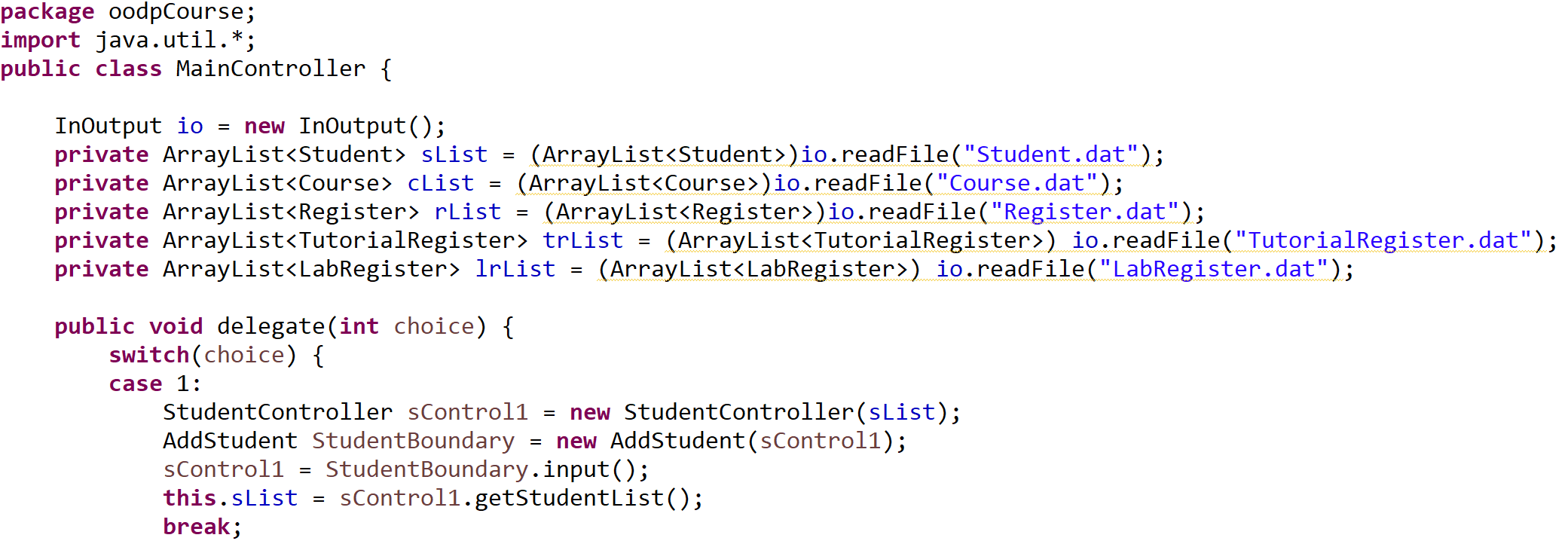


Figure 1: Declaration of ArrayList in Main Control with updating before case break

5. Error Checking

There was a need to ensure that the code would be easy to debug if there are any errors. The use of error checking through basic if-else, do-while loops and try-catch exceptions help to prevent mistakes in user input that would conflict with the program requirements. Should there be an exception that occurs, we are able to narrow down the error to a certain part of the code, making the debugging process more efficient.

**Design Principles**

1. Single Responsibility Principle

Each class is given one responsibility be it as a boundary, control or entity. By spreading the responsibilities among the classes, we avoid having a “god” class and over-reliance on a single class which would also make it harder to resolve errors since the entire responsibility of the class has to be checked. For example, a student is created as an object of the entity class (Student). These student objects will contain the attributes of the student such as the student name and matriculation number which is unique. The student objects will then be stored in a controller class (StudentController). The controller would create students and assist in the logic of the program between the boundary and entity class such as checking if a student exists. Finally, the boundary class (AddStudent) acts as the connector between the user and the program by getting user input for the details of the student and output basic user feedback for the user actions.

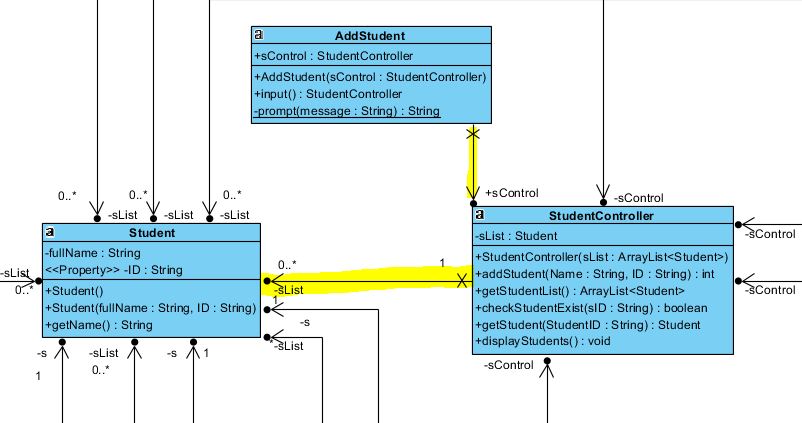


Figure 3: Entity Control Boundary Example

2. Open-Closed Principle

The key to the Open-Closed Principle is the use of abstraction. This makes the class open for extension but closed for modification as the base abstract methods have been declared and must be implemented in the concrete classes. For example, a course type interface would allow us to declare different types of courses (lecture, lecture with tutorial, lecture with tutorial with lab) without having to worry that any of the derived classes do not have sufficient method implementations. If there comes a requirement for more types of courses such as lecture with lab, the course can be easily extended from the base method to include getter and setter for lab list.

3. Dependency Injection Principle

The dependency injection principle allows the separation of an object creation from its usage. This promotes low coupling yet high cohesion as the high level modules can be reused easily for extension without losing the low level modules. For example, when a student is registered to a course, a course register object is created by the registration manager. In the process, the registration manager acts as a Context Dependency Injection Container to inject a result object for each coursework component that exist in the course, into the course register. When there are more types of courses, they can be extended from the base course class and have a resulting course register object with results for each component.

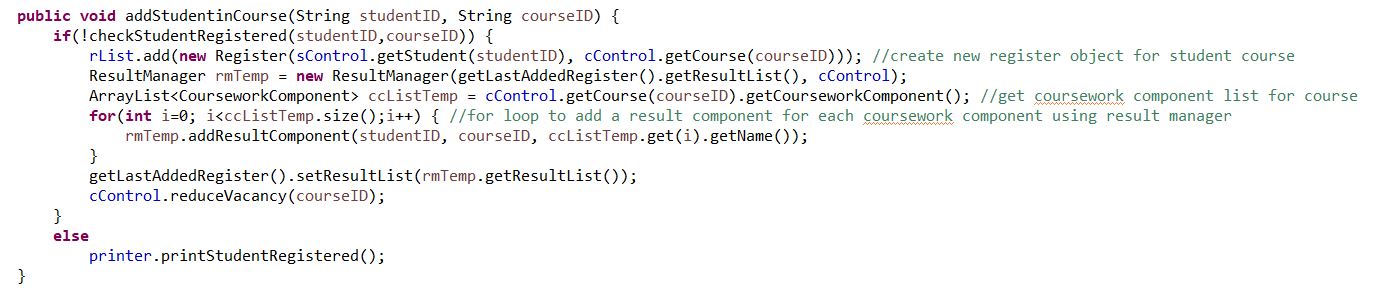


Figure 4: Injecting of result object into register for each coursework in course

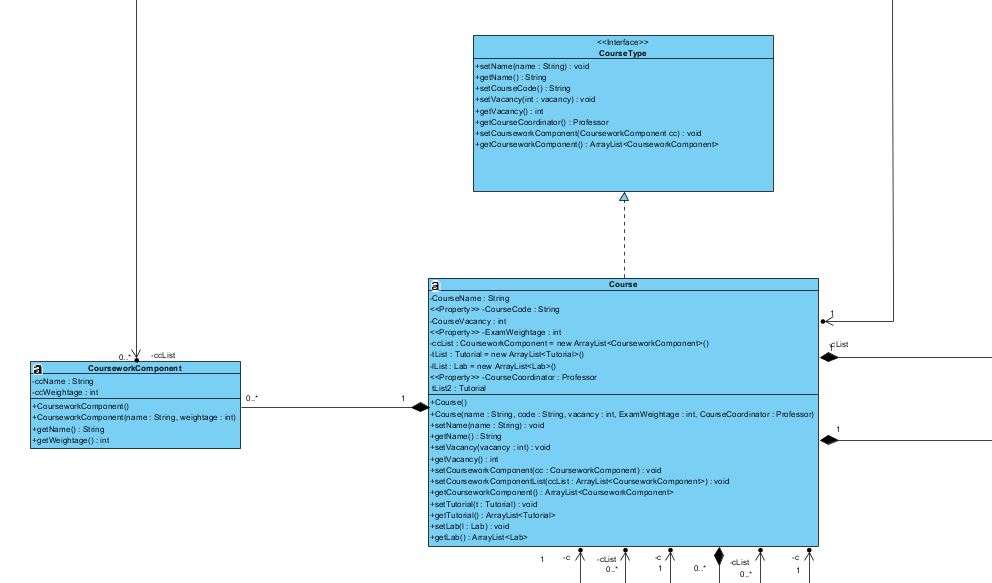
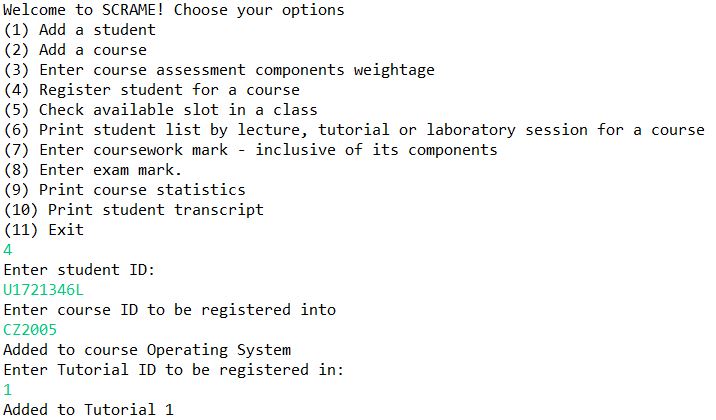


Figure 5: Course implementing CourseType Interface

**SCRAME Menu**



**Test Cases with Results**

|  |  |
| --- | --- |
| Test Case | Output |
| Invalid data entry - Case-sensitive |  |
| Registering for already registered course |  |
| Invalid data entry - Beyond marks limit |  |
| Invalid data entry - Beyond component weightage |  |
| Student Transcript Example |  |
| Print students in course when no student is registered |  |