[CSC8404 Coursework]
Student Name: Moaz Adnan
Student Number: 200747376

Aim

Design a set of interfaces and classes that can be used by the university's management system to regulate all the students that are currently registered to a program provided by the university. The students can be one of three types: A Post Graduate Taught Student (PGT), a Post Graduate Research Student (PGR), and an Undergraduate Student (UG). The main distinction between the students is the number of course credits that they can apply for; the PGT and UG students can apply for 180 and 120 credits-worth of courses, respectively, while the PGR students can only be assigned a supervisor. A student can only be of one type. A record of the modules and supervisors is maintained by the university in the form of files which it can access to update a student's information if needed.

The university should be able to interact with the students via the following methods:

1- A method that allows the university to register a new student onto the system, and assign a smart card and a student ID.

2- A method which can amend the student's data, such as register the student for a new course, or assign a supervisor if required.

3- A method that can terminate the student by removing it from the university's student records.

4- A method that informs the university about the number of currently registered students of a specific type.

The assignment of the smart card has the following caveats:

1- An undergraduate student must be at least 17 years old.

2- A postgraduate student must be at least 20 years old.

3- A student can't be issued multiple smart cards.

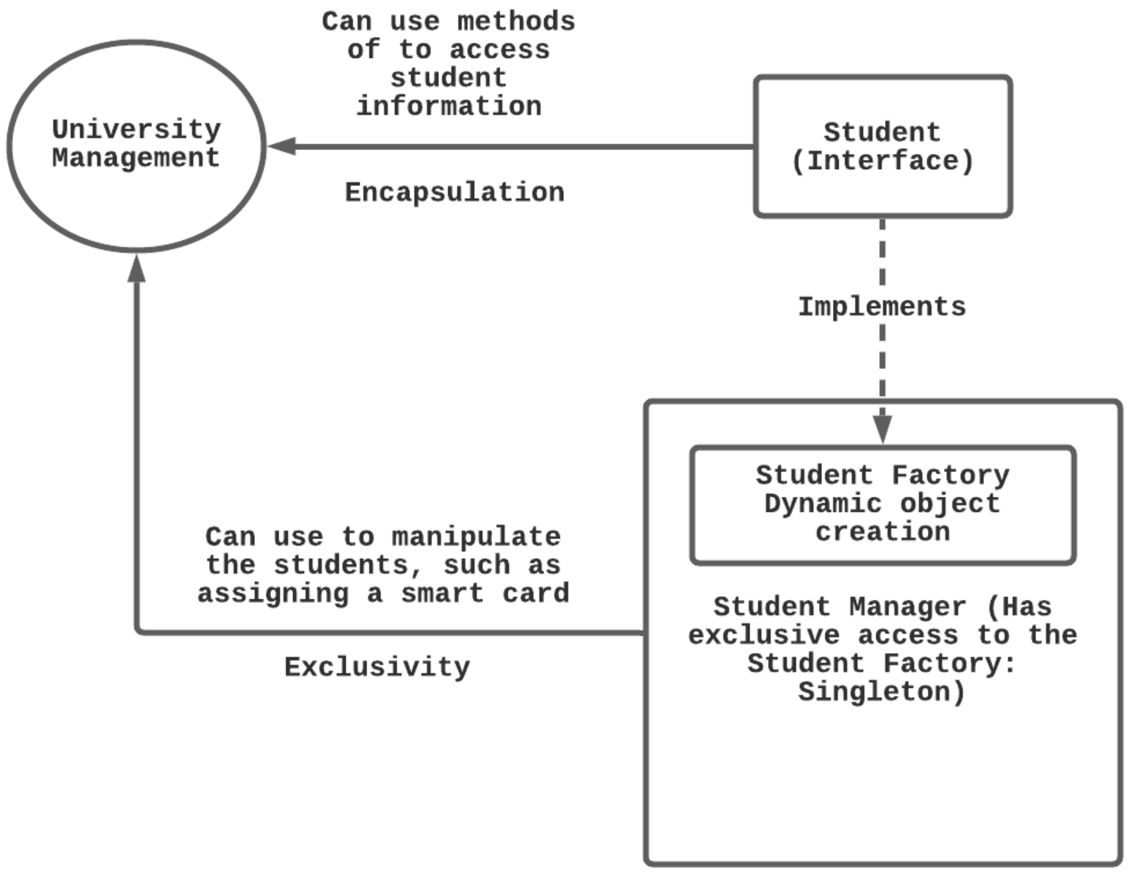
Design Choice

As there are three types of students, the program needs to be able to generate the requested type in accordance with the specified parameters. The generated student should then be able to retrieve the information assigned to it for later use by the aforementioned university methods. Since assignment is involved, exclusive access and uniqueness is required by methods which entail it.

With all this in mind, a hybrid approach pertaining to the singleton and object factory design patterns is employed. The former guarantees exclusivity, while the latter can provide dynamic object creation based on the input from the user.

The student’s information that will be required by the university needs to be both contained, and only accessible by a set of predefined methods that the university can call upon without directly influencing the student’s details, i.e., the university should not be able to change the student’s personal information, such as the name or the smart card data, **once it has been assigned**. Hence, a student interface is created to expose a group of methods for the university to use.

Putting it all together:


Can use methods of to access student information
University Management
Student (Interface)
Encapsulation
Implements
Student Factory Dynamic object creation
Can use to manipulate the students, such as assigning a smart card
Student Manager (Has exclusive access to the Student Factory: Singleton)
Exclusivity

Exclusivity - Student Manager

Exclusivity describes the methods which can modify the student object inline with what is delineated under the ***Aim*** section. Methods that can create, terminate, and update a student object fall into this category. Direct manipulation of the student's record, which is required in all of the above-mentioned functions, can potentially lead to some indeterminate state when multiple users have the option to modify it. This can occur when the changes made by one user aren't preserved across every instance of the student's record being modified. To emulate a **semblance of persistence** in the management system, an intermediary class is created to implement the functionality defined under the ***Aim*** section - the ***Student Manager*** class.

This class follows the singleton design pattern. There is only one instance of the class that is available to the university management personnel. It keeps track of all students who are registered onto the system by acting as the gatekeeper for manipulating their data. All university personnel have to go through the student manager in order to utilize the functionalities presented under the ***Aim*** section. This approach ensures that all changes made to the student's details are carried over to the next time someone tries to access them. Another benefit of this persistence is that duplication can be dealt with by adding checks and balances in the student manager class.

A brief description of the class' methods:

1- ***createStudent():*** This method creates a student object by exclusively using the student factory. All requests for a new student have to pass though the student manager.

2- ***getInstance():*** This method returns the singular instance of the student manager, regardless of the number of invocations.

3- ***registerStudent():*** This method registers a new student onto the system by first generating a student ID, assigning the smart card, and making a new entry in the student records.

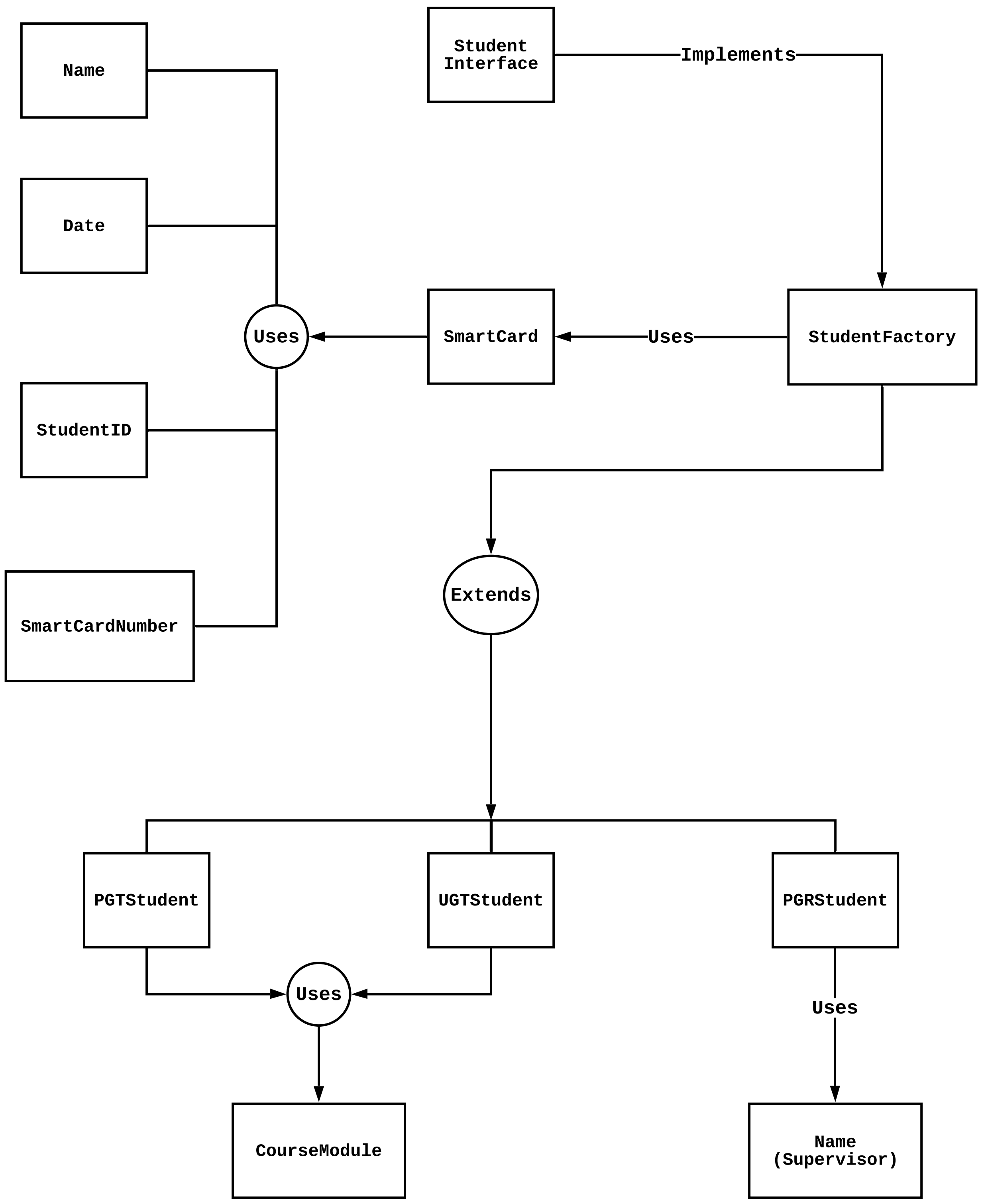
4- ***noOFstudents():*** This method provides the number of registered students of a given program.

5- ***terminateStudent():*** This method removes a student from the university's system by removing the entry from the student records.

6- ***amendStudentData():*** This method registers a student for a new module if the student isn't registered for enough credit hours. Only the PGT and the UG students are assigned modules, the PGR students are assigned supervisors.

Encapsulation & Dynamic Object Creation - Student (Interface)

Student UML Diagram



Encapsulation is achieved by providing an interface to the university to utilize whenever a user wants to view the assigned information of a given student. Logically speaking, once the student has been given a smart card with all of his/her information in tow, all the changes that the university can follow-up with will be dependent on said information. Fields such as the smart card info, the student's name and the program type will act as permanent identifiers for the duration of the student's stay.

Some of the methods provided in the interface were mandated by the assignment. All the other methods were created to support the structure mentioned under the ***Design Choice*** section and in the ***UML diagram***. Each method can be used by both the university management, and by the student manager. The former can use them to retrieve the student's data, while the latter can use them as checks for assignment of modules and supervisors.

The interface exposes the common behavior amongst all students. However, the differentiators of the student types, such as the number of credits and type of module assignment, necessitate some specialized methods for each student. All the shared behaviors are collected in an abstract class called the ***Student Factory***. This class acts as the creation point for all the different student objects. Access to it is regulated by the student manager, which provides it with the university management's request for the type of student to be created. All the guards for the information required to create the student can be placed here to ensure a streamlined operation.

Just like with the student manager, the student factory needs to keep track of all the types of students that have been requested by the university's management system. However, unlike the student manager, the factory uses the student's name as the identifier to distinguish between them. It doesn't use the student's ID as the student has not been assigned one during the creation process. A student only receives his/her ID once it has been registered with the university's system, i.e., createStudent() happens before registerStudent().

As for the student-specific functions, the student factory is extended into three classes which cater to the Undergraduate Student, Postgraduate Taught Student, and the Postgraduate Research Student, respectively. These classes override the student interface methods which were mandated by the coursework assignment, and add their own methods that are required for more particular operations.

Any common data members, or has-a fields, follow the same arrangement as the methods. The smart card is expected to be with every student, whereas only the UG and PGT students require course modules. Therefore, the smart card ends up in the student factory, while the course modules appear exclusively in the extended classes

Note: There is a deviation from the UML diagram due to how create and register student operations work. As they are separate, all students need to store their names and dates of birth somewhere before they are registered. So to get around this issue, both the smart card and the student factory have a Name and Date field. Once the student has been registered, the name and date of birth of the student are shifted to the smart card (some duplication).

A brief overview of the interface's methods:

1- ***getStudentID():*** This method retrieves the student's ID. *(used by the student manager when the university personnel either wants to remove or update the student's info).*

2- ***getStudentType():*** This method returns the type of the program the student is registered to *(used by the student manager when the university personnel either wants to remove or update the student's info).*

3- ***listModules():*** A student-specific method which is overridden in the extended classes. It returns all the modules that the student has opted for *(used by the university personnel).*

4- ***enoughCredits():*** A method which tells whether a student is registered with enough course credits *(used by the student manager when the university personnel wishes to register a student to a new course).*

5- ***returnName():*** A method which returns the student's name *(university personnel use).* Used to assign a student with a smart card for the first time *(student manager use).*

6- ***returnBirthDate():*** A method which returns a copy of the student's date of birth *(university personnel use).* Used to assign a student with a smart card for the first time *(student manager use).*

7- ***getExpiryDate():*** A method that returns the expiry date of the smart card *(used by the university personnel).*

8- ***hasCard():*** A method which tells whether a student has a card or not *(used by the student manager to detect duplication).*

9- ***getStudentInfo():*** A method that returns all the student's information present on the smart card, the modules, and the type of program *(mainly used by the university personnel use).*