

**SC/CE/CZ2002: Object-Oriented Design & Programming**

**ASSIGNMENT**

*Building an OO Application*

**2022/2023 SEMESTER 2**

**SCHOOL OF COMPUTER SCIENCE & ENGINEERING**

**NANYANG TECHNOLOGICAL UNIVERSITY**

## Declaration of Original Work for CE/CZ2002 Assignment

We hereby declare that the attached group assignment has been researched, undertaken, completed and submitted as a collective effort by the group members listed below.

We have honored the principles of academic integrity and have upheld Student Code of AcademicConduct in the completion of this work.

We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work. In addition, disciplinary actions may be taken.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Course  (CE2002 or CZ2002) | Lab Group | Signature /Date |
| Ong Sheng Da (U2220668C) | SC2002 | Z51 | ShengDa  15/04/23 |
| Branson Chang (U2121641L) | SC2002 | Z51 | 15/04/23 |
| Babu Rishe (U2121918L) | SC2002 | Z51 | 15/04/23 |
| Dennis Chen Chee Hao (U2220058a) | SC2002 | Z51 | 15/04/23 |

Design Considerations

A look at our class UML diagram, perhaps, you might find it shocking, there seem to be several dependencies in our UML diagram. We see some classes having many dependencies pointing to them. It seems to violate one of the fundamental design principles, which is to produce a program with little coupling, reducing the dependency of our classes as much as possible.

There is a reason why we design in such a way. Perhaps, we should have a quick overview of how our program works and highlight how certain OOPD concepts are applied. *(Note: The UML diagram that we proposed only includes the interactions between the classes as the question suggested. Some of the codes we wrote like reading and overwriting the CSV files, some of the menu displays for each user etc are not shown in the UML we proposed).*

1. We read data from the CSV files
2. We create a list class for each of the object classes (student, supervisor, coordinator, request, project). These list classes get the data read from the CSV files and append them into lists for the use of the program. This is under the method “initialise” for each of this list classes.
3. We will then authenticate the user base on their inputs of user ID and password. We will do a string comparison here, upon having the same string, the user will be able to log in. The display they will see depends on the circumstance they are in.
4. Student: They will see the different display:
5. If they do not have a project, they can see and execute the following:
6. Change password
7. View Available Project
8. View Request History and Status
9. Request to register for FYP
10. Logout
11. If they have a project, they can see and execute the following:
12. Change Password
13. View Registered Project
14. View Request History and Status
15. Request to Change Title
16. Request to deregister for FYP
17. Logout
18. Supervisor: They will see the following:
    1. If they do not see a new request:
19. Change Password
20. My Project
21. Request
22. Request to Transfer Student
23. Logout
    1. If they see a new request:
24. Change Password
25. My Project
26. Request (New)
27. Request to Transfer Student
28. Logout
29. Coordinator:
    1. If they see a new request:
       1. Change Password
       2. My Project
       3. Request (New)
       4. Logout
    2. If they have no new request:
       1. Change Password
       2. My Project
       3. Request
       4. Logout

Subsequent displays will not be shown here in the report but what we wanted to emphasize here is there are similar processes for the 3 kinds of users, which are changing password and viewing of requests. We can choose to create a parent class, which is what we did here. We created a *User* class and have *Student* Class and *Supervisor* Class inherits from it. Since Coordinator is a Supervisor, we will inherit from the Supervisor. This is to fulfil SRP in the Design Principle, so that not 1 class (*User)* handles all kinds of functionality.

An interesting thing observed in the *Coordinator* Class is that apart from inheriting the from *Supervisor*, the methods only consist of Constructor. This is again done after much consideration. Looking at the coordinator’s job, we can see that apart from being a supervisor, what distinguishes it from the other user type is it is that it is able to process the request. However, we had already created *Request* Class, which will be discussed later. We realised that we could handle the processing of this in a different class, such that suppose there are any changes to be committed in the future, such as a new request type for the coordinator to process, we will not need to edit explicitly the *Coordinator* Class, we just have to create new classes for these requests. This is to fulfil the OCP Design Principle, where we open to extension and close to modification, or to modify as few codes as possible in our design.

This brings us to our *Request* Class. As we can see, the *Request* Class is an abstract class. Why do we use abstraction in this case? Because we will want to reduce the number of dependencies on this class itself. By using abstraction, we are not only able to fulfil SRP, and OCP as mentioned above, but we also have achieved to achieve DIP, inverting the dependencies from these classes most importantly, by doing this step, we managed to cut down many interactions (dependencies and association) between these classes! More will be shared in the reflection below about our struggles, quarrels and ultimately achieving this feat.

Perhaps, the last interesting thing to be discussed will be why there are so many dependencies for our *List* Classes. A look and we can see severe dependencies between *ProjectList Class, and some form of dependencies for StudentList* and *SupervisorList* Class. We would like to point out that this is not done without consideration. We do know it looks as if we violate some form of fundamentals in design principles. But we design it as such because we are keeping the goal of OCP in mind. How so? These classes act as a mediator that mediates between the reading of the CSV files, and the processing of data for the functionality of the program. It is because that the data is a list of instances of objects, we can ensure we read what we want from these data files. While we did not include this due to it being out of the requirement, we could easily create an interface class for reading different types of data files (say in the future, one might want to read from json, xml, txt etc), then, we create the different classes to process the data file type. We would then force them to be stored inside these list arrays, our program will still run as fine. Perhaps, a slight modification will be needed in these mediator classes (the *initialise* method will need some modification as now we only take CSV files as our only input, we did not consider other filetypes input and I/O design), but the rest of the program will not be changed. In this way, even extensions made in the I/O will have little effect on the executability of our program, achieving OCP, and our fundamental design goal, low coupling and high cohesion.

UML Class Diagram

Chart, box and whisker chart

Description automatically generated

Figure 1: Overview of our UML Class Diagram: Reference to Z51-grp3-UML Class Diagram.vpp for a clearer visualisation.

Test Cases

We have conducted many test cases but we will only cover 8 test cases. This report will cover the brief steps undertaken to conduct the test cases. Detailed results can be obtained from “testcases.pdf”.

Test Case 1 - User Login with wrong User ID

Steps:

1. Login and enter a wrong user ID

Result: User unable to login to system

Test Case 2 - User Login with wrong password but correct user ID

Steps:

1. Login

2. Enter a correct User ID and password

Result: User unable to login to system

Test Case 3 - User Login and request change of password and logout

Steps:

1. Login

2. Choose option 1: change password

3. Change password

4. Log out and log back in

Result: User able to login with his new password

Test Case 4 – Student (YCHERN) Login and register for FYP + Coordinator (ASFLI) rejects request

Steps:

1. Login to YCHERN account

2. Choose option 4: register FYP and register for project 5

3. Logout from YCHERN account

4. Login to ASFLI

5. Choose option 3: Request (New)

6. Choose option 1: Pending request

7. Choose option 2 to reject request

8. Logout from ASFLI account

Result: YCHERN request rejected

Test Case 5 – 2 students (YCHERN and KOH) register for the same supervisor project (BOAN) + Coordinator (ASFLI) approve requests.

Steps:

1. Login to YCHERN account

2. Choose option 4: register FYP and register for project 5

3. Logout from YCHERN account

4. Login to KOH1 account

5. Choose option 4: register FYP and register for project 6

6. Logout from KOH1 account

7. Login to ASFLI account

8. Choose option 3: Request (New)

9. ASFLI approves both YCHERN and KOH1 FYP

10. Logout from ASFLI account

Result: YCHERN and KOH1 FYP approved. Project 5 and 6 becomes reserved and project 7 becomes unavailable

Test Case 6 – Change project title by YCHERN (student) and supervisor (BOAN) approves

Steps:

1. Login to YCHERN account

2. Choose option 4: request to change project title

3. Change project title to OOP

4. Logout from YCHERN account

5. Login to BOAN account

6. Choose 3: Request (New)

7. Approve the request

8. Logout from BOAN account

Result: Project title changed successfully. This information will be reflected in student, supervisor and coordinator app

Test Case 7 – Transfer student (BR015) from YPKE to ASFLI to YPKE

1. Login to BR015 account

2. Choose option 4: register FYP and register for project 16

3. Logout from BR015 account

4. Login to ASFLI account

5. Choose option 3: Request (New)

6. ASFLI approve YPKE request

7. Logout from ASFLI account

8. Login to YPKE account

9. Choose option 4: Request To Transfer Student

10. Transfer BR015 from YPKE to ASFLI

11. Logout from YPKE account

12. Login to ASFLI account

13. Choose option 3: Request (New)

14. Approve request to transfer student from YPKE to ASFLI

15. Go back to main menu

16. Choose option 4: Request to transfer student

17. Transfer BR015 from ASFLI to YPKE

18. Go back to main menu

19. Choose option 3: Request (New)

20. Approve request to transfer student from ASFLI to YPKE

21. Logout from ASFLI account

Result: Successfully transfer project from YPKE to ASFLI and back to YPKE

Test Case 8 – Deregister FYP (YCHERN)

1. Login to YCHERN account

2. Choose 5: Request to Deregister FYP

3. Logout from YCHERN account

4. Login to ASFLI account

5. Choose option 3: Request (New)

6. Approve for YCHERN to deregister for FYP

7. Logout from ASFLI account

8. Login to YCHERN account

Result: Student is unable to login to system as he has already been deregistered for FYP

Reflections

It was in early week 9, when one of our team members Dennis, said that he has completed the program. He has tested most of the program and it works just fine. Free from bugs, it almost looked like a submittable work, we almost celebrated with happiness. Except, none of us has any design knowledge then. However, the joy did not last long, we did not know we committed a major mistake in our program. Upon learning the 5 Design Principles, we re-examined back our program. It was horrendous, we violated every single design principle. Our first draft only consisted of 5 classes in our program (*Student*, *Supervisor*, *Coordinator*, *Project*, *MainApp*). They are all dependent on each other, with each of the classes “overworked”.

Immediately, we changed our codes and separate them. Focusing on SRP in our mind, being the first principle in SOLID, we immediately make changes. The most obvious was to separate out the request, into 4 subclasses, so that each class does its job. We also separated out the lists we hard-coded inside our program. All seems good, each class has its own job now and the program is running fine. Until Sheng Da steps in and said “Horrible”. Well, Dennis was unhappy at that time, being the main coder fixing most of the changes, his effort was just shot down by another of his team member. They argued, where Dennis said he fulfilled SRP and the program was working just fine. But Sheng Da retorted that it violated OCP. There are too many dependencies which cause the program to be non-extendable.

Eventually, a new design was proposed, which we will name Draft 3. While there were still some forms of dependency which seems unnecessary, at the very least, some form of inheritance was used, removing some dependencies. While we were slightly happier, little did we realise we read a requirement wrongly. We misunderstood the instruction to *use the data directly.* We thought that we could hard-code the data directlyinto our program. But it meant using the excel file. Our program now will also need to include the I/O. We will also need to think about committing changes to the data files, ensuring that the changes made to the data were not lost. These led to the design of Draft 4 and Draft 5.

Draft 4 was based on the design of directly editing the CSV files such that it removed away the *lists* Classes. Draft 4 also focuses on using the idea of abstraction to reduce dependencies as much as possible. However, there was a slight logic error while doing this. Draft 4 has abstracted too much such that the *projectID* will be seen as redundant for accessing the *Student*’s project*.* It basically loops the whole list to find a matching project for the student. The return type was a list of projects for students, containing only a single project in that list. But a student only owns a project, so it makes no sense to return it as a *list<Project>* type rather than a *Project* type.

While Draft 5 was based on the design of using a list as a *mediator*. It focuses less on dependency issues. A look at our final class diagram and instinctually we might reject such a design. Again, quarrels occur, ultimately, we decided to go with merge draft 4 into draft 5, as we eventually agreed that forcing the data into the list will perhaps reduce the ambiguity of the effect on unknown filetypes used to obtain the data, which may affect the executability of our program upon extension to other file types.

Merging draft 4 into draft 5 also managed to reduce our dependency on the *Request* Class, which we make it abstract. In this way, we managed to reduce and invert some of the dependencies to the *Request* Class, achieving DIP in the process.

Perhaps, an improvement to be made is what is mentioned earlier in the report, using interfaces to create a *ReadFile* Class so that our program can read from other file types too other than CSV. We did not do this in this project because it is not part of the class diagram and is outside of our consideration. Efficiency-wise in our program itself is quite poor as it basically loops through all the in our search method. Since we had the project ID, perhaps a Hashmap may be a better choice in this case, which will improve the search time of our program from O(n) to O(1).

*This is the end of the report, thanks for reading.*