SYSC 3020: Introduction to Software Engineering

Group 7: Graduate Admissions Final Documentation

Korede Ogunyinka: 100860675
Josh Cohen-Collier: 100855222

Chosen Implementation Language: Java

Date Submitted: June 16, 2014

Professor: Alan Davoust
Head TA: Sunint Khalsa

Repository Location:
https://github.com/jaycarleton/sysc3020project.git

1.0 Implemented Subsystems

The project implements 3 distinct, but related subsystems, of the graduate application system documented in reports 1 through 3. The first system is the ProfileManager, the second is the ApplicationManager, and the third is the Matcher(also known as Matching subsystem).

1.1 ProfileManager Subsystem

The ProfileManager is responsible for the creation, maintenance, and destruction of all profiles that are created in the system. This includes two distinct classes, the StudentProfile, and the ProfProfile, both of with are subclasses of the abstract Profile.

The manager includes methods for creating both types of profiles, as well as accessor's for specific profiles(and both subtypes), as well as a method to destroy a Profile, once it is no longer needed (such as when a student accepts an offer, or a professor retires). The manager maintains a list of active profiles, which every method interacts with, either adding to, removing from, or accessing a member of.

The profile manager can interact directly with a related ApplicationManager, and Matcher, which are both included as instance variables. The ProfileManager itself does not invoke not include any of the methods that involve modifying a profile or application, only a way to access those profiles.

1.2 ApplicationManager Subsystem

The ApplicationManager is responsible for the creation, maintenance, and destruction of Applications attached to StudentProfiles. This functions in an identical manner to the ProfileManager, with regards to the Profiles. The manager similarly includes methods to create, destroy, and access Applications, each of with interacts with its own list of active Applications, in identical fashion to the ProfileManager. The main differences between the two sub-systems are that there is only one type of Application, as opposed to the 2 concrete Profile types, and that the relationship between the ApplicationManager and the ProfileManager is very one-way.

The ProfileManager is aware of the ApplicationManager, and can view its contents, and invoke methods at will. The ApplicationManager, however, is only able to access the profiles that own each individual Application, limiting their ability to view other Profiles, including professors. In a way, this almost functions like a server would to a client. This was done to both reduce the load on the subsystem, as it would be an unnecessary burden, and also for security reasons, as someone who hacked the ApplicationManager would have limited use of its database.

1.3 Matcher Subsystem

The Matcher subsystem is the most unique subsystem that was implemented. It is responsible for the construction and maintenance of all Administrators and Associates. The main difference, however, is that there is only one Administrator, and one Associate, per system. This means that any new Administrator or Associate automatically overrides the old one, in a sense "retiring" their older predecessor. It has methods that create Admins and Associates, as well as access them. Since it has only one instance of each Administrator and Associate, it has singular instance variables(as opposed to a list of Administrators and Associates).

The major method of the subsystem, however, is the matchApplication function, which takes an Application and matches it to ProfProfiles. This involves creating two lists of Professors, one for advising on the Application, and one for supervising it. Each professor is evaluated to determine if it has the same department as the owner of the Application. If it does, it alternates between being added to each list. If the profile becomes an advisor, the Applications list of advisors itself is updated. The Application has no list of supervisors, because they should only be able to contact their advisors.

The Matcher is also completely aware, and can interact mutually with the ProfileManager, and since the ProfileManager has knowledge of the ApplicationManager as well, it can access that too. Like the ProfileManager, and ApplicationManager preceding it, the Matcher does not directly interact with the Profiles or Applications, but merely access the Administrators and Associates, which do that internally.

1.4 Subsystem Interaction

In the implementation used, it was necessary to attach the ApplicationManager to the ProfileManager, and make the ProfileManager and Matcher completely aware of each other, in order to allow them to access private methods. In this way, to access Applications, the ProfileManager would invoke methods from the ApplicationManager, which it had as an instance variable. The ApplicationManager had no access to the Matcher, and only access to the individual owning StudentProfiles attached to Applications. Both the ProfileManager and Matcher had each other as instance variables, meaning they could interact with the other types of classes by invoking methods from the other subsystem instance.

Since the system was not complete, it was necessary to make these changes from the original documentation, since otherwise the subsytems wouldn't function at all. It was also necessary to add many accessor methods, since in full implementation the System class would have access to all private methods through its subsystem instances.

2.0 Classes

Java is an object-oriented language, which means that each type of object, known as a "Class", needs to be defined in a separate document. Each class will have multiple types of relationships with other classes.

2.1 UML Class Diagram

This class diagram shows each of these relationships in standard UML format. It can be viewed in the "support" folder of the repository.

2.2 Class Roles

Each class fulfills a different role inside the implemented subsytems. They are described, in general, below.

Admin: Only one can exist at a time. Responsible for matching Applications to advising and supervising ProfProfiles. Can comment on Applications. Encapsulated in Matcher.

Application: Many, or none, may exist at once. The object of interest in the system, passed along between all users, in order to be evaluated for a decision of acceptance. Can be commented on, rated, ad have its CV upated. Each Application has an owning StudentProfile. Encapsulated in ApplicationManager.

ApplicationManager: The subsystem in charge of all Applications. One one is required per system, unless additional memory is needed. Encapsulates Profile(which is abstract), as well as its subclasses ProfProfile, and StudentProfile.

Associate: Only one can exist at a time. Responsible for determining the status of acceptance for Applications that have been passed to it. Encapsulated in Matcher.

Comment: A type of attachment to an Application. Many, or none, may exist for each Application. Has an author, as well as actual date/time stamping. Created by ProfProfiles, Administrators, or Associates. Encapsulated in ApplicationManager, as it is not active, but is attached to instances of Application.

Matcher: The subsystem in charge of all Administrators and Associates, as well as matching Applications to Professors. Only one should exist per system, unless the system is too large, and needs to be segmented. Encapsulates Administrator, and Associate.

Profile: The instance that is used by professors and students to interact with the system. Each system can have many. Superclass to ProfProfile, and StudentProfile, and is itself abstract. Encapsulated in ProfileManager.

ProfileManager: The subsystem responsible for managing all active Profiles. Each

system should have only one ProfileManager, unless there are so many Profiles that the system is overwhelmed, and needs to be segmented. Encapsulates Profile, ProfProfile, and StudentProfile.

ProfProfile: The class used to interact with the system by Professors. Can be many at once in system. Each ProfProfile will be passed an Application. It can then comment on it, rate it, invest in it, and then decide whether it is complete enough to pass on to the Associate for review.

Rating: A type of attachment to an Application. Done on a scale of 0-5 stars, anything else will invoke an error. Many, or none, may exist for each Application. Has an author, as well as actual date/time stamping. Created by ProfProfiles, Administrators, or Associates. Encapsulated in ApplicationManager, as it is not active, but is attached to instances of Application.

Reference: A type of attachment to an Application. Many, or none, may exist for each Application. Has a written reference, as well as actual date/time stamping. Created by StudentProfiles to enhance their application. Encapsulated in ApplicationManager, as it is not active, but is attached to instances of Application.

StudentProfile: The class used to interact with the system by students. Can be many at once, and each one has an attached Application. Can add references to profile, contact any advising Professors by email, and has the final say on any offers made on the Application by the Associate. Encapsulated in ProfileManager.

3.0 Usage Instructions

The program can be run on any computer in the SYSC department, on one of two IDEs for Java. They are both fairly simple, and easy to use. Although both are perfectly acceptable, to run test cases, Eclipse is probably better, as its terminal doesn't get rid of old messages once the screen is filled up(which is common for the test cases). BlueJ is reccommended for simply playing around and examining the program, as individual instances of any type of object can be created, inspected, and then invoke methods.

3.1 Eclipse IDE Instructions

To run the project in Eclipse, you can manually download the files, drag an drop into a folder, then set that folder as the directory. However, to simplify the process, the project can be imported easily from github.com. The instructions to do so are as follows.

-Open Eclipse for Java
-Right click in the empty space of the leftmost pane, Project Explorer. If it is not there, click on the button as shown...

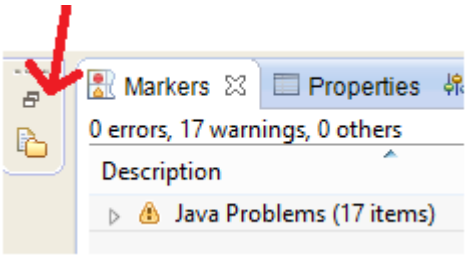

Figure 1: Project Explorer Button

-Select Import->Import...
-Select Git-> Projects from Git
-Clone URl
-Enter the info as shown in the pic below:
-URL = https://github.com/jaycarleton/sysc3020project.git
-User= sysc3303
-Password= summer2014

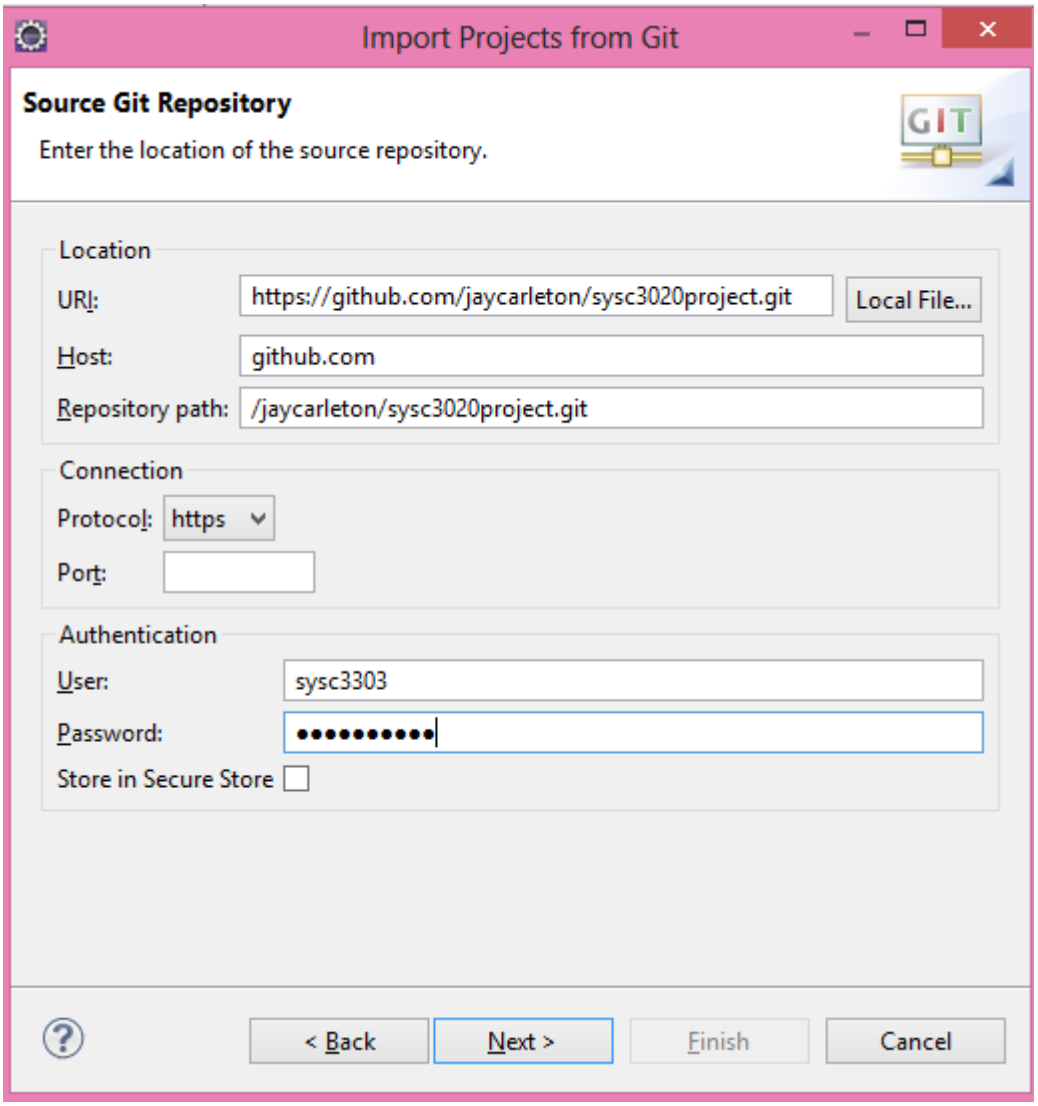

Figure 2: Import Dialog Box

-Select Next, then Next again
-Choose an empty directory, if not already chosen
-Select Next
-Select Import Existing Projects
-Select Next, then Finish

Once the project is imported, to operate the Unit Tester, follows the next steps.

-In the Project Manager tab, open up Iteration 5, then src, then Default Package
-Right click on UnitTester.java
-Select Run as->Java Application
-Double click on the newly opened Console window to make it full screen
-Follow UI Instructions

3.2 BlueJ IDE Instructions

Operating on BlueJ is very simple. To import, simply download all files into the same folder, then double click on the file labeled "package.bluej". The unit testing can be done by simply right clicking on the UnitTester class symbol, then clicking on "Main(String args[])". This will open the UnitTester interface, and from here simply follow the instructions.

The main advantage to this IDE however, is that you can interact with the system in real-time, and inspect the contents of every objects instance variables, which often include lists of other classes. This can simulate a simple, easy to navigate playground, and is beginner-friendly for new Java users.

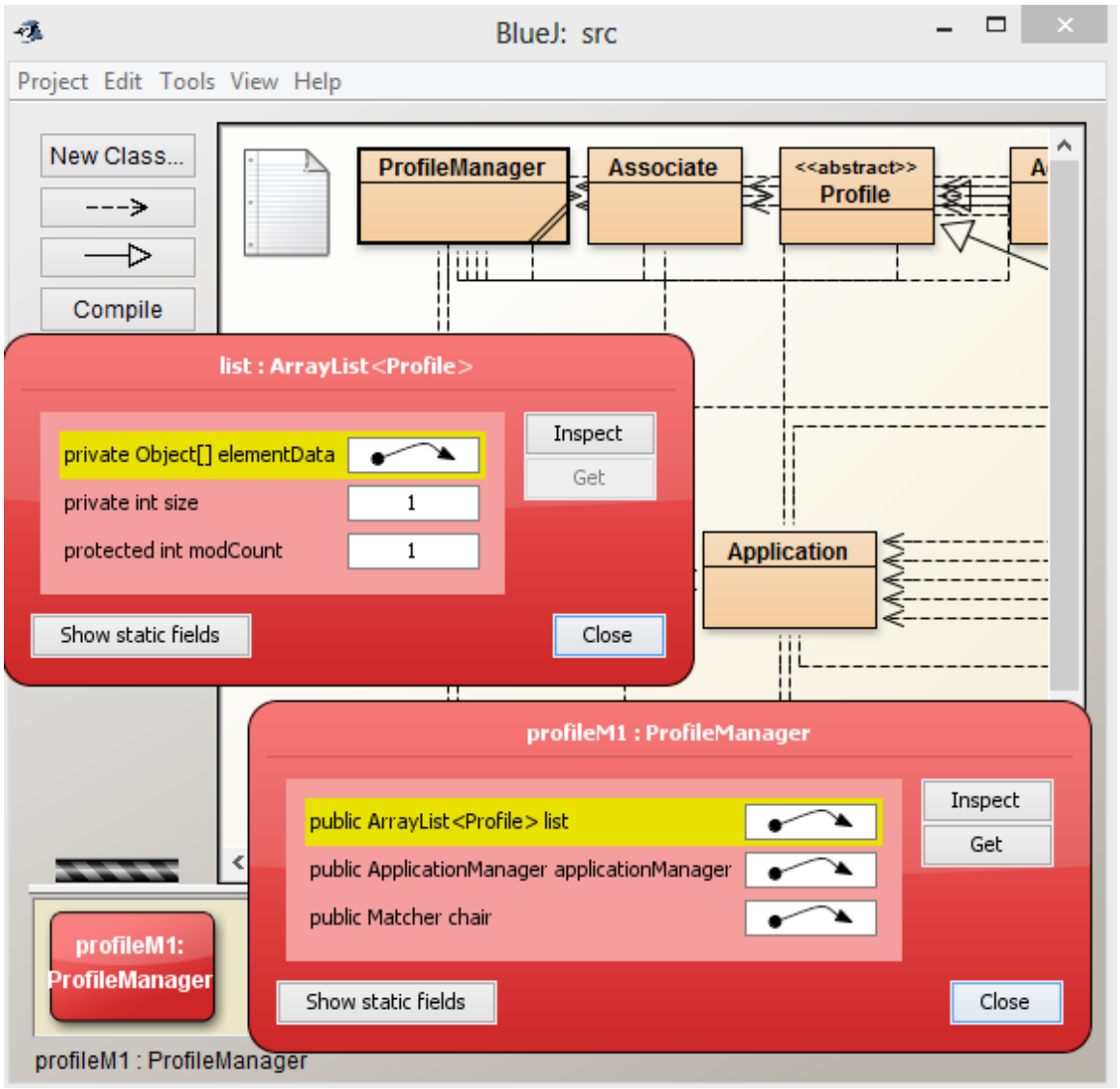

Figure 3: BlueJ Example Usage

4.0 Testing

A UnitTester class, along with a variety of methods that involved test scenarios, were used to easily and efficiently test the functionalities of the system. The tests, and the tests they fulfill, are below. Detailed, step-by-step explanations of each test are viewable in the java code.

changeApplication():
-Creating StudentProfile, ProfProfile
-Updating CV
-Commenting, Rating
-Allocating Funds
-Accessing Profiles via ProfileManager

-Accessing Application via ApplicationManager
-Printing/Viewing Applications

updateStudent():
-Creating StudentProfiles
-Editing StudentProfile name, email, id, and department
-Viewing StudentProfile info

updateProfessor():
-Creating ProfProfiles
-Editing ProfProfile name, email, id, and department
-Viewing ProfProfile info

withdraw():
-Create StudentProfile, ProfProfile
-Creating offer of acceptance
-Viewing accepted/declined Application

fullCheckup():
-Creating Administrators, Associates, StudentProfiles, ProfProfiles
-Commenting
-Matching Applications to ProfProfiles
-Viewing Application advisors
-Making Associate decision
-Viewing accepted/declined Application

5.0 Summary of Usability

The subsystems implemented follow their contract interface exactly, excepting methods that involved subsystems that were not implemented. Each subsystem function was implemented with the exact same signature(same set of input parameters, output returns), and methods were only added as accessors to compensate for the lack of other subsystems. Someone who read the subsystem design, and interface contract, would be well equipped to use the program.

The code was commented exhaustively in standardized javadoc format, with explanatory comments on nearly every line of code. Plain, simple english terms were used to ensure that anyone could look at the code and determine what each line does, right away.

The User Interfaces provided upon each test case, and every printing function ensured that the user would always have clear instructions on what to do. Any invalid input is met with an error message explaining what was incorrect(eg ID number not following format 100xxxxxx). Someone who has never used this program, or even java itself, should have no problem operating the basic tests, as long as they read this accompanying document.

6.0 Future Aspirations

Because of the fact that the system was not fully implemented, and was done in a short period of time, there are many improvements that could be made to future iterations, both for future group projects to work on, and for actual real-world applications.

Firstly, it would be ideal to implement the entire system, as it would eliminate many discrepancies between the compact theoretical system, and the larger, more accommodating incomplete system. Next, it could be beneficial to incorporate a message system for profiles, such that actions like receiving an Application, or personal message could be performed.

Ideally, the complete system would implement a free-roam mode, in the form of a detailed user interface. This would allow the user to experiment in detail, and view the entire functionality of the system themselves, not just in test cases.

7.0 Work Done

Java Code
-Comment: Josh Cohen-Collier
-Matcher: Josh Cohen-Collier, Korede Ogunyinka
-Profile: Josh Cohen-Collier
-ProfileManager: Josh Cohen-Collier
-ProfProfile: Josh Cohen-Collier
-Rating: Josh Cohen-Collier
-Associate: Korede Ogunyinka, Josh Cohen-Collier
-ApplicationManager: Josh Cohen-Collier
-Application: Josh Cohen-Collier, Korede Ogunyinka
-Admin: Korede Ogunyinka, Josh Cohen-Collier
-StudentProfile: Mohammed Zen-El Din, Korede Oguyinka
-UnitTester: Josh Cohen-Collier

Support
-Documentation(this one): Josh Cohen-Collier
-Code commenting, formatting: Josh Cohen-Collier
-UML Class Diagram: Iteration 4: Josh Cohen-Collier, Mohammed Zen-El Din
-UML Class Diagram: Iteration 5: Josh Cohen-Collier
-Readme file:Josh Cohen-Collier
-Automated Testing(with "Emma" plugin): Mohammed Zen-El Din
-Iteration 4 Documentation