Super Turtle Hackers Design Document

SOLID

Single Responsibility

The CRC cards outline each of the classes and were implemented such that each has a single duty. For example, the user class is only concerned with the user entity that stores all the data pertaining to the user such as the profile, unique identifier, and current matches. The profile class contains the user's information which includes name, year of study, program of study, courses for a term, interests, and contact information. The responsibility of storing the information in the profile class was not given to the user class in an effort to adhere to this single responsibility principle. The same idea can be extended to the course class containing a course's code, lecture time, and tutorial time in a separate class from the profile.

Open-closed principle

Our design is open for extension because currently, we have not added features like report and chat for users, but we plan to implement those features in Phase 2. Those features can be extended without changing the code. It is closed for modification because its basic function, which is to match users, stays the same.

Liskov substitution principle

This principle is demonstrated by the interchangeability of the user and admin classes. These classes

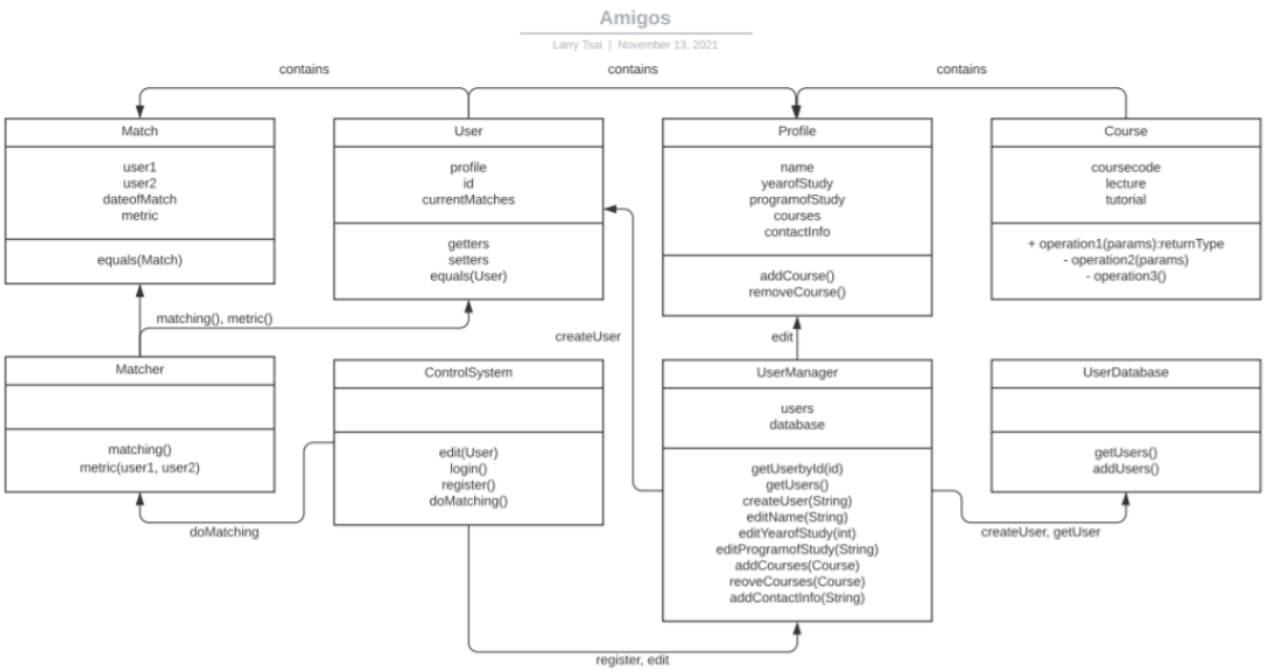
Interface segregation principle

The userdatabase and controlsystem interfaces split the duties of the organization of the users and the actions of the users. Most actions for the client do not require the userdatabase so therefore there is a division between the two interfaces to allow for more client-specific interfaces.

Dependency inversion principle

There is an admin class that gives certain users permissions to see information that is hidden to other users. The users, profile, course, and match entities have private variables that are only specific to the objects in question. It is the structure of these objects that are public such as the methods but the literal private information such as ids, courses, and interests, are all kept private.

Clean Architecture



The CRC was implemented with a clear design of entities, use cases, interface adapters, and frameworks/drivers. In the center, the entities are separated from the other layers and do not have any knowledge of any of the other classes. For example, the user class has no information about the userdatabase or the usermanager. ControlSystem, an interface adapter, takes the input and requests the UserManager, a Use Case, to make a new User, Profile and one or more Course objects, all of which are entities. This is a clear path from the surface to the core of clean architecture. The UserManager will also communicate with the UserDatabase to log in the new User. UserManager is a Use Case and UserDatabase is a database, so UserManager depending on UserDatabase is a violation of Clean Architecture because an inner layer is depending on an outer layer. However, this violation is an exception because UserManager has to interact with the database to add and edit users. In lecture, it was mentioned that this can be accepted. Similarly, once a day, the ControlSystem will request a list of User objects and send it to the Matcher, another use case. The Matcher will generate a hashmap of matches, called match hashmap based on each User's courses and interests, and return it back to ControlSystem. The match hashmap has a key of a User object and the value would be that user's list of matches. This match hashmap is then forwarded to UserManager by the Controller. Lastly, UserManager takes the match hashmap and allocates the list of matches to each User.

The Dependency Rule is followed throughout the code except for the violation mentioned above. A concrete example is the interaction with the UI. The UI only displays information and gets input from the user. After that, the ControlSystem gets the input from the UI and communicates with the Use Cases to carry out functions.

Design Patterns

Currently, we are planning on implementing the factory design pattern. This can be implemented in our project for the abstract class Report. We are planning on having two kinds of Reports – HarassmentReport and DistressReport. Both take slightly different information and return different reports. We can create the interface Report with the method “createReport()” and leave the instantiation up to the subclasses. Now, the HarassmentReport will implement the method “createReport()” and DistressReport will implement the method “createReport()”. Both will return new HarassmentReport() and DistressReport() objects (or products) respectively.

This design pattern is particularly helpful in this case as we can create different types of reports in the future, say ReportUserProfile and not have to worry about changing the implementation of Report. We can create a new subclass and define ReportUserProfile individually. Only Admins are allowed to resolve these reports and they don’t need to worry about what information is common and how things change if new report types are added. They can also create different ways to resolve or deal with each report and not worry about how this impacts the base class.

Progress Report

Thus far, our group has diligently worked on implementing phase 1 of this project. The app was implemented as a webapp using Spring. The frontend team (Lawrence and Adrian) used the designs from Phase 0 to create the webapp structure using HTML and CSS. Akshat and Rue worked on creating the database for the storage of the data of the users. The backend team (Tony and Dien) build upon the existing skeleton code in coding the additional classes that were not implemented before. The design document and UML was created by Lawrence and Dien. For the next phase, everyone will be focused on finalizing the remaining classes mainly in the backend.