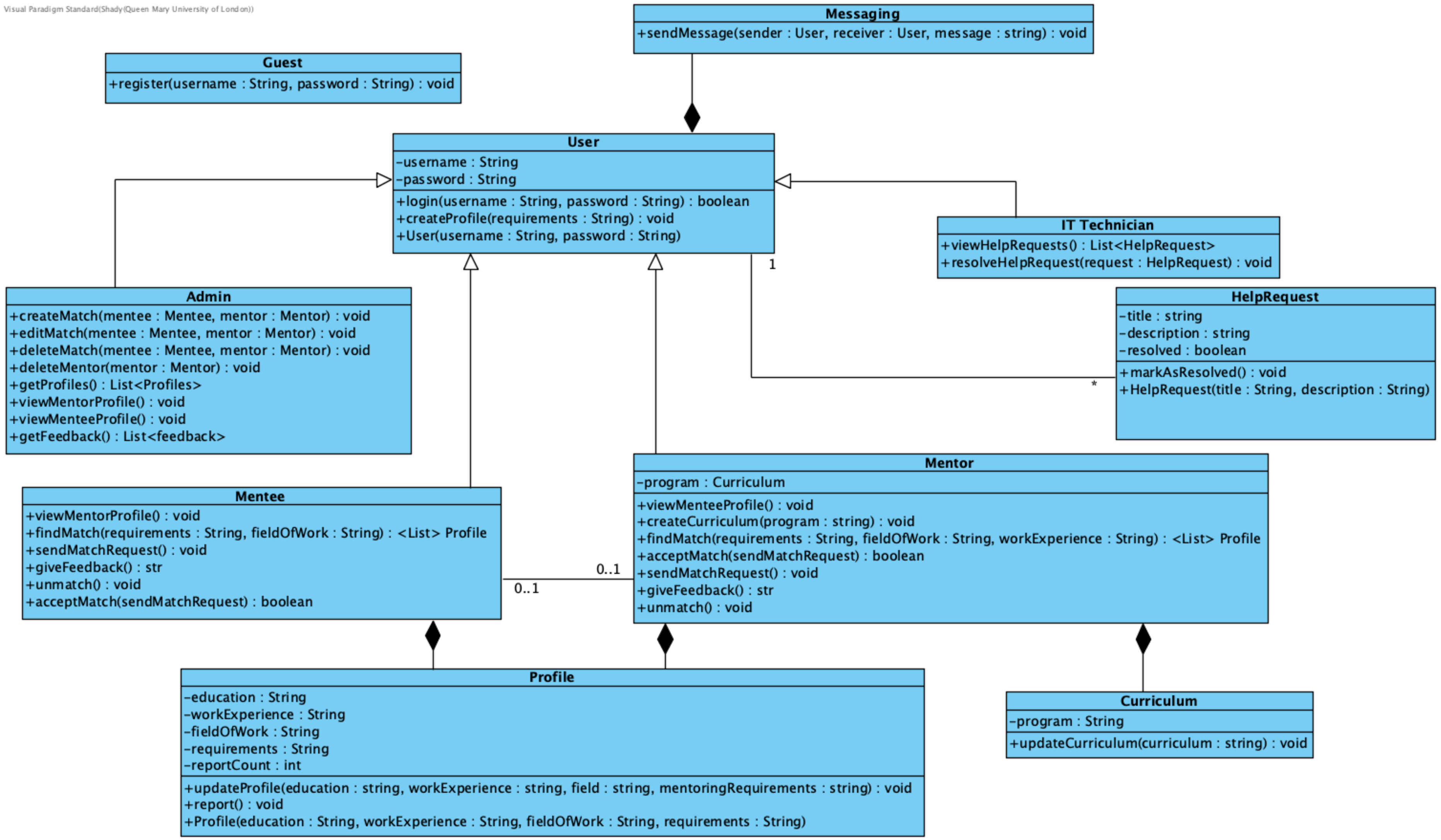
Group 30

Mentor Matching App

ECS506U Software Engineering Group Project

Design Report

1. Class Diagram (40%)


Messaging
+sendMessage(sender : User, receiver : User, message : string) : void
Guest
+register(username : String, password : String) : void
User
-username : String
-password : String
+login(username : String, password : String) : boolean
+createProfile(requirements : String) : void
+User(username : String, password : String)
IT Technician
+viewHelpRequests() : List<HelpRequest>
+resolveHelpRequest(request : HelpRequest) : void
1
HelpRequest
-title : string
-description : string
-resolved : boolean
+markAsResolved() : void
+HelpRequest(title : String, description : String)
*
Admin
+createMatch(mentee : Mentee, mentor : Mentor) : void
+editMatch(mentee : Mentee, mentor : Mentor) : void
+deleteMatch(mentee : Mentee, mentor : Mentor) : void
+deleteMentor(mentor : Mentor) : void
+getProfiles() : List<Profiles>
+viewMentorProfile() : void
+viewMenteeProfile() : void
+getFeedback() : List<feedback>
Mentee
+viewMentorProfile() : void
+findMatch(requirements : String, fieldOfWork : String) : <List> Profile
+sendMatchRequest() : void
+giveFeedback() : str
+unmatch() : void
+acceptMatch(sendMatchRequest) : boolean
0..1
0..1
Mentor
-program : Curriculum
+viewMenteeProfile() : void
+createCurriculum(program : string) : void
+findMatch(requirements : String, fieldOfWork : String, workExperience : String) : <List> Profile
+acceptMatch(sendMatchRequest) : boolean
+sendMatchRequest() : void
+giveFeedback() : str
+unmatch() : void
Profile
-education : String
-workExperience : String
-fieldOfWork : String
-requirements : String
-reportCount : int
+updateProfile(education : string, workExperience : string, field : string, mentoringRequirements : string) : void
+report() : void
+Profile(education : String, workExperience : String, fieldOfWork : String, requirements : String)
Curriculum
-program : String
+updateCurriculum(curriculum : string) : void

2. Traceability matrix (10%)

Class	Requirement	Brief Explanation
User	RQ1	Each User has a unique attribute **‘username’**
User	RQ2	The class contains a **‘login’** function that takes both arguments, username and password
Guests	RQ6	The “**register**” function allows guests to create a profile
Profile	RQ7	Each profile has an **‘education’** attribute that allows user to give detail on their education
Profile	RQ8	Each profile has an **‘workExperience’** attribute that includes the profiles work experience
Profile	RQ9	Each profile has an attribute **‘fieldOfWork’** where they can specify what field they are in.
Profile	RQ11	Each profile has an attribute **‘requirements’** where they can specify what they require in a mentor or mentee
Admin	RQ17	This class contains a **‘createMatch’** function that allows for admins to create mentorship opportunities
Admin	RQ18	This class contains a **‘editMatch’** function that allows for admins to edit mentorship opportunities
Admin	RQ19	This class contains a **‘deleteMatch’** function that allows for admins to delete mentorship opportunities
Admin	RQ27	The class contains functions **‘viewMentorProfile’** & ‘viewMenteeProfile’ where admins are able track the progress of both mentors and mentees
Mentor	RQ45	This class has a function **‘giveFeedback’** that can be used to forward any complaints they may have as a mentor
Mentee	RQ45	This class has a function **‘giveFeedback’** that can be used to forward any complaints they may have as a mentee
Admin	RQ46	This class has function **‘getFeedback’** that is used to handle any issues or complaints

3. Design Discussion (20%)

Domain Model vs Class Diagram

Our class diagram is similar to our original domain model in many ways. The classes we used in the class diagram are listed in the table below showing the differences in the use or purpose of these classes between the domain model and the class diagram itself.

Classes/Entities	Design Decisions	
	Domain Model	Class Diagram
Guest	Referred to users that did not meet the criteria to join the mentorship programme and could only browse through the limited pages on our web-app.	Refers to anyone without an account. They can register for an account by creating a username and password.
Mentor	Referred to and was a parent class for Alumni, Consultants and Internal Staff – individual directly involved in the matching process	Generalisation of users and can create curriculums, view mentee profiles, accept or find matches and give feedback.
Mentee	Referred to and was a parent class for Graduates, Returners to Work and Ex Forces – individuals directly involved in the matching process	Generalisation of users and can view mentor profiles, accept or find matches and give feedback.
Users	Parent class to mentors and mentees that was also associated with entities called: support panel, profiles and matching algorithm.	Parent class to admins, IT technicians, mentors and mentees. Can login and create profiles. Represents anyone with a registered account.

Profile	Profiles were used to create a list of requirements for the matching algorithm to create matches and return a list of profiles	Composed of mentors and mentees, can be updated, or profiles can be reported when there are issues.
HelpRequest	-	Users can create help requests that include a summary of the issue and can be marked as resolved when dealt with by an administrative team.
Messaging	-	A communication panel composed of all users, allowing them to communicate with each other.
Admin	Could view profiles and resolve disputes	Can change matching status, view user profiles and give feedback to other users.
IT Technician	Had a support panel and communicated closely with administrators/moderators	More focus on providing support to all users. They can view and resolve help requests.

Our new design did not include a class for **Supervisors**. Initially we had them listed as an extra user in our domain analysis. However, after re-evaluating we realised that the functionalities for guest users and supervisors were not too different and can be combined as one guest class for better design. So whilst our domain model had a supervisor entity, our class diagram represents this user type in the "**Guest**" class.

Our Domain Model also has 3 sub-categories for mentors and mentees.From initial research we learnt that FDM mentors are usually **Alumni, Consultants or FDM Employees**. We also learnt that FDM accepts mentees that are **Graduates, Returners to Work or Ex Forces**. We illustrated this by showing them as generalisations in the domain model. Our class diagram does not include this detail as we aimed to create a more focused and detailed design where we focused on the matching process itself rather than the process of becoming a mentor or a mentee. In our design, verifying whether they fall into the relevant categories is not directly a part of the matching process.

Design Decisions: User Relations

The class diagram implements multiple **associations** and 4 **generalisations**. These generalisations are used for user relationships. The Users class is a parent class to the admin, IT technicians, mentors and mentees classes. The parent class includes only the common functionalities or attributes between its child classes so the design is clearer and easily shows the differences in the user types; any unique attributes or functionalities are in their own distinct user group classes. In this case, the common feature between all child classes is that they represent groups that have a registered account on the web-app.

The only other user group that exists is the Guest class, which represents the group of people that do not have an account. In our domain analysis report we represented this group as “Other users”. In the domain analysis report, this group represented third parties that are not involved in the matching process. This could be random browsers that came across the website or FDM competitors, FDM clients and any other affiliates they may have. However, in this class diagram, the Guest class is more generic and represents any user that does not have an account and can register; this includes future mentors and mentees.

We assigned **multiplicities** based on our requirements elicitation report. For instance, one mentee can only have one mentor and multiple help requests can be sent by the same user. Although mentors might, realistically, have time to take up more than one mentee, our focus is on creating an algorithm that can create matches by identifying whether users are already matched or not so we limited the mentor-mentee relationship to only 1 or no match each. This is to improve the quality of the mentorship which we believe would be reduced by increasing the quantity. In future, this can be changed to allow mentors to have more than one mentee by changing multiplicities such that mentors can have 1-5 mentees; mentees will still only be assigned to one mentor.

Design Decisions: Algorithm & Requirements

One issue we explored in our requirements elicitation’s risk assessment was that the objective algorithm might keep matching users to inactive accounts. Administrators are able to change the status of accounts with this new design and can prevent such issues from occurring. This is possible through the create, edit and delete match functions. Feedback is also stored as a list and is accessible by the IT technicians for better understanding of potential improvements to the application and any errors that occur for users.

Key **attributes** required for the functionality of our application include the list of compatible profiles and the list of feedback. The matching algorithm would first check whether the status is true or false and match users only to other users who do not already have a match - preventing any scope for disappointment or confusion and maximising the objectiveness of the matches made. The list of compatible profiles is created by the matching algorithm and displayed to the users involved. The use of lists is most convenient for a profiles database that the algorithm can refer to and a feedback database that the learning and development team (both IT Technicians and Admins) can refer to.

Our class diagram also used multiple compositions. This is evident where the profiles class is composed of mentors and mentees. The curriculum is composed by the mentors and the messaging panel is composed of all users. These compositions are used to show the involvement of each user in these specific functionalities of the web-app. The messaging class represents a communication panel - we used a composition here because the panel instance does not exist if there are no users. All user types can use the communication panel including admins and IT technicians. Similarly, profiles are composed of mentors and mentees since they cannot exist without these user groups. The attributes listed in profile are displayed when a user checks another user's profile.

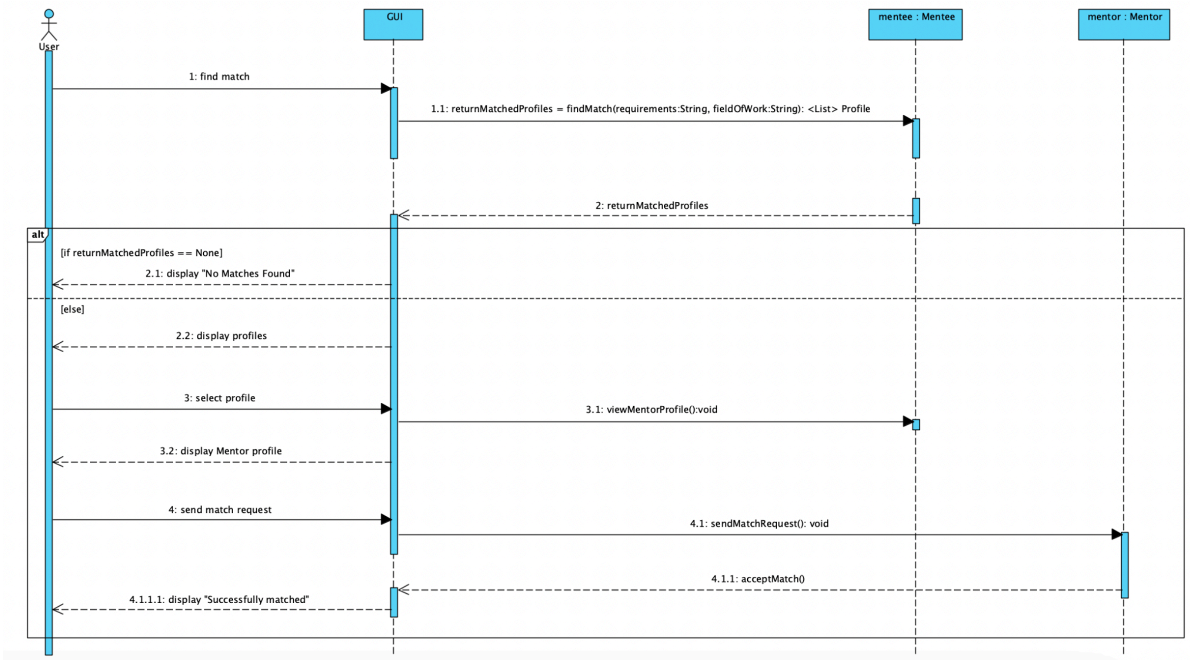
4. Sequence Diagrams (30%)

NOTE: The GUI class is not introduced in the class diagram, rather, it is included in all 3 sequence diagrams to make them easier to understand. And the function 4.1.1 acceptMatch() in Sequence Diagram 1 is meant to be acceptMatch(sendMatchRequest): boolean.

Sequence Diagram 1: Find match

This sequence diagram describes how a mentee finds a match with a mentor. It corresponds to the “find match” use case, which is a key use case. Prerequisites include that the user already has an account and is logged in.

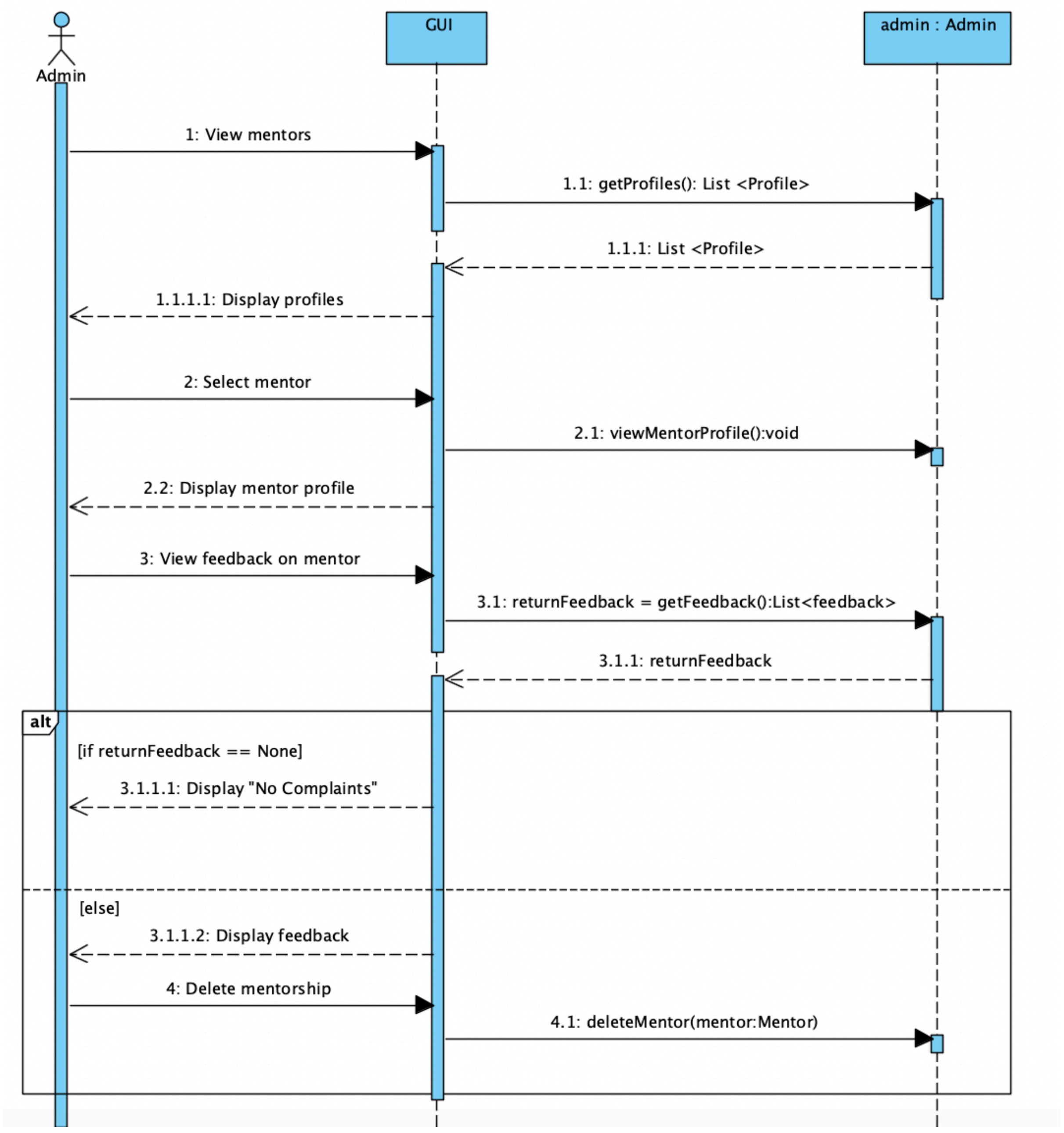
The sequence diagram involves the user clicking a find match button, this will request the system to return a list of matched mentor profiles to the user, this list will be returned to the user. If the list is empty then it will display “No matches found”. If not empty, it will display the matched mentor profiles to the user. The user can then view the mentor's profile in detail, the system will then call a function that views the mentor's profile to the user. The user can then send a match request to the mentor, the system will send the match request to the mentor, and the mentor will then send an “accept match” response to the user. When this happens, the system will display “successfully matched” to the user, and match the mentor and mentee.


User
GUI
mentee : Mentee
mentor : Mentor
1: find match
1.1: returnMatchedProfiles = findMatch(requirements:String, fieldOfWork:String): <List> Profile
2: returnMatchedProfiles
alt
[if returnMatchedProfiles == None]
2.1: display "No Matches Found"
[else]
2.2: display profiles
3: select profile
3.1: viewMentorProfile():void
3.2: display Mentor profile
4: send match request
4.1: sendMatchRequest(): void
4.1.1: acceptMatch()
4.1.1.1: display "Successfully matched"

Sequence Diagram 2: Admin End Mentorship

This sequence diagram describes how an admin ends a mentor's mentorship. It corresponds to the “end mentorship” use case, which is a key use case. Prerequisites include that the admin already be logged in.

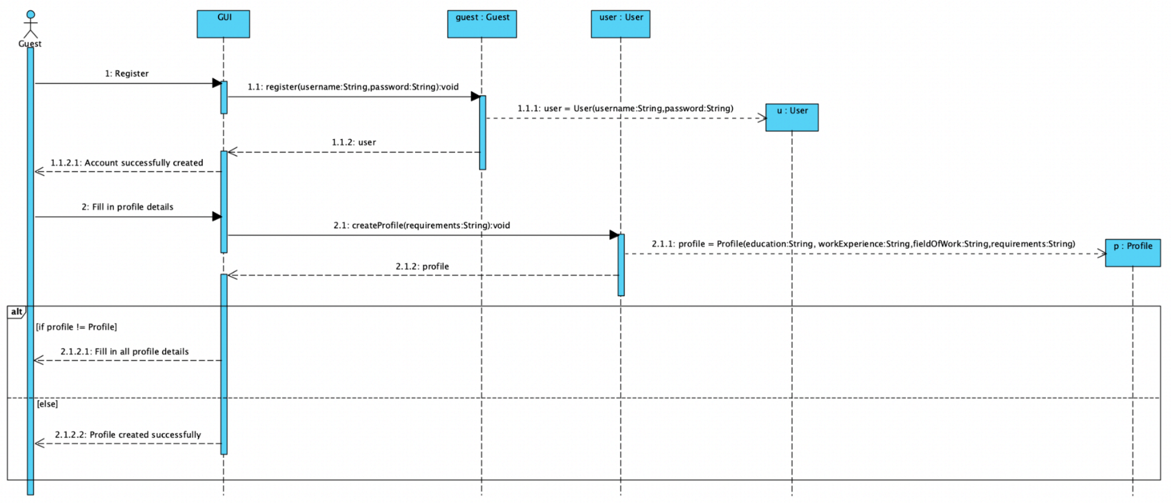
The sequence diagram involves the admin viewing the mentors profiles, this will make the system send a request to the admin class which will return a list of mentor profiles. The admin will then be able to select a specific mentor, this will make the system call a view mentors profiles function, which will display the mentors profile to the admin. The admin can then view the feedback given on the mentor, this would be done by the system calling a function that will get the feedback, and return it in the form of a list. If the list is empty, it will display “no complaints”, if it isn’t, then the feedback will be displayed for the admin to view. The admin will then delete the mentors mentorship, this will cause the system to call a function which will delete the mentors mentorship.


Admin
GUI
admin : Admin
1: View mentors
1.1: getProfiles(): List <Profile>
1.1.1: List <Profile>
1.1.1.1: Display profiles
2: Select mentor
2.1: viewMentorProfile():void
2.2: Display mentor profile
3: View feedback on mentor
3.1: returnFeedback = getFeedback():List<feedback>
3.1.1: returnFeedback
alt
[if returnFeedback == None]
3.1.1.1: Display "No Complaints"
[else]
3.1.1.2: Display feedback
4: Delete mentorship
4.1: deleteMentor(mentor:Mentor)

Sequence Diagram 3: Register

This sequence diagram describes how a guest becomes a user with a profile in the system. This sequence diagram corresponds with the “register” use case. The prerequisites are that the guest have an internet connection.

The sequence diagram involves the guest choosing to register. This will involve the creation of a username and password (which are passed as arguments in the register function). When this happens, the system will do a constructor call to create a new User object for the guest to use. The user object will be returned to the guest, and the system will display “account successfully created” to the guest. The guest will then fill in their profile details. After these details are filled in, the system will pass these fields as arguments to the User class. The User class will then do a constructor call to create a profile object filled with the guests details. It will then return those details to the guest. If the object created is not of type “Profile” (meaning not all the fields have been successfully filled in) the system will display an error message “fill in all profile details”. Else, it will display “Profile created successfully”.


Guest
GUI
guest : Guest
user : User
u : User
p : Profile
1: Register
1.1: register(username:String,password:String):void
1.1.1: user = User(username:String,password:String)
1.1.2: user
1.1.2.1: Account successfully created
2: Fill in profile details
2.1: createProfile(requirements:String):void
2.1.1: profile = Profile(education:String, workExperience:String,fieldOfWork:String,requirements:String)
2.1.2: profile
alt
[if profile != Profile]
2.1.2.1: Fill in all profile details
[else]
2.1.2.2: Profile created successfully