Group 16

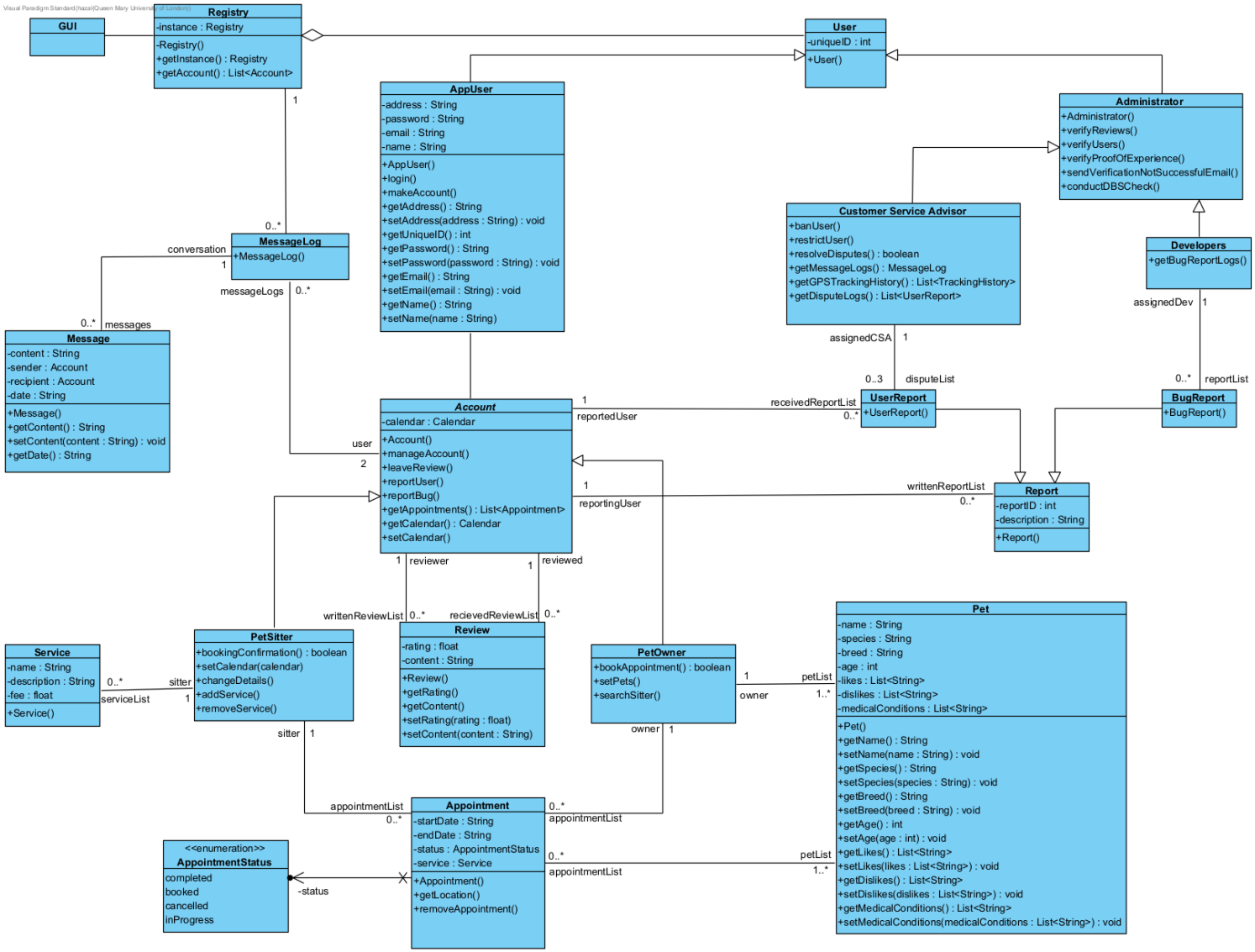
Fetch

ECS506U Software Engineering Group Project

Design Report

Table of Contents

1. Class Diagram



2. Traceability matrix

In the traceability matrix map, all the data requirements you identified to a class from your class diagram.

In the explanation, briefly explain how the requirement is fulfilled by the class design.

Class	Requirement	Brief Explanation
Users	RQ2, RQ11	Superclass of AppUser, contains unique ID which would be common to all users
AppUser	RQ20,RQ21,RQ44,RQ17,RQ3,RQ15,RQ16,RQ27, RQ28, RQ29	Contains Methods and features including username and password common to pet owner and pet sitter, subclass of Users and superclass of Owner and PetSitter
Owner	RQ13, RQ22, RQ32, RQ33, RQ43, RQ45, RQ30	Can give reviews of sitters search for sitters and book them, subclass of AppUser
Pet Sitter	RQ13, RQ24, RQ25, RQ31, RQ37,	Contains methods specified to giving and removing services and confirming them, subclass of AppUser
Administrator	RQ12, RQ58, RQ59, RQ60, RQ46	Contains Methods common to CSAs and developers, subclass of Users
Customer Service Advisor	RQ14, RQ49, RQ50, RQ51	Subclass of Administrator, has methods specified to solve disputes
Developer	RQ14, RQ52, RQ53, RQ54, RQ55, RQ56	Subclass of Administrator, has methods specified to solve bugs and access commits from
Message	RQ35	Contains details of sender, recipient and date
Registry	RQ18	Singleton pattern, stores all accounts
Report	RQ44	Superclass of two dedicated classes for reporting user and errors in the code
Review	RQ36	One to many association with account, used to give reviews
BugReport	RQ48	Used to report bug to developer, subclass of Report

UserReport	RQ44	Used to report details of user to CSA, subclass of Report. 0 to 3 association with CSA
GUI	RQ1, RQ2, RQ3, RQ4, RQ5, RQ6, RQ7, RQ8, RQ9, RQ10	Associated with message log, used to act as GUI for entire program to display messages and handle communications between all users
Appointment	RQ38, RQ39, RQ40, RQ41, RQ42, RQ43	1 to many association between pet owner, pet and appointment, used for storing bookings
Pet	RQ30	Contains details of the pet, has a 1 to many association with pet owner
Service	RQ28	One to many association with pet sitter, used for recording services provided by pet sitter
MessageLog	RQ51	One to many association with message, used as a record of messages

3. Design Discussion

Explanation of design decisions

Firstly, we defined a User super class to represent a general user in the system. Each User object has an integer type uniqueID attribute to uniquely identify each user. It also has a User() method, the constructor method for the class. We also used an AppUser class which is a subclass of the User class to represent a user who's using the app as a customer and is not involved in the development or maintenance of the app like an administrator, developer or customer service advisor for example. They have an email and password attribute to log them in, and the email also provides a way to contact them if necessary. They also have a name and address attribute to help identify the user, and the address helps to calculate the distance between certain pet sitters and owners and helps them to find each other in person for an appointment booked between them. An AppUser class has the AppUser() constructor method, login() method for logging a user into their account, and a makeAccount() method for creating an account for that user. It also has a getter and setter method for each of its attributes along with a getUniqueID() method to fetch the unique id for that app user.

We also have an Account class which is a subclass of the AppUser class and a superclass for the PetOwner and PetSitter class because an account represents an account for an app user and the methods and attributes specific to an account, and either of the subclasses could implement certain methods within an account differently. For this reason, as well, Account is an abstract class meaning its subclasses must override and define the methods within the account class to ensure each of the subclasses implements each method in its way. Account is a superclass for PetOwner and PetSitter because both subclasses are in-app users and both will have an account, so having the methods defined in the account abstract class ensures each subclass knows what methods it should have, and what methods it needs to implement. It has the Account() constructor method, manageAccount() method for managing the account, leaveReview() method to allow a user to leave a review on a pet sitter or owner. It also has the reportUser() and reportBugs() methods to allow users with an account to report any users who are breaking the app's community guidelines or the law and report any bugs noticed while using the app. It also has the getAppointments() method to fetch any appointments that the account has, as well as the getCalendar() and setCalendar() method to fetch or modify the schedule of an account in terms of what appointments they have and when. We have a PetOwner class to represent a pet owner within the app which has a bookAppointment() method to allow them to book an appointment with a pet sitter, a setPets() method to allow them to define what pets they have and a searchSitter() method to allow them to search for a pet sitter out of any pet sitter within the app. We also used a PetSitter class that doesn't have any attributes as it inherits all the attributes it needs from its superclasses. It has a bookingConfirmation() method to confirm a booking of their services, a setCalendar() method to allow them to make changes to their scheduled appointments and bookings, a changeDetails() method to allow them to make changes to their details when necessary, an addService() method to allow them to add a service to their profile that they can provide to a pet owner, and a removeService() method to allow them to remove a service if they can no longer provide it. We also have a service class to represent a service that a pet sitter can offer. It has a name attribute to represent the service's name, a description attribute to describe what the service is, a fee attribute to determine how much the service costs and a Service() constructor method.

In addition to this, we also have an appointment class to represent an appointment made by a pet owner for a pet sitter. It has two string attributes startDate and endDate to determine when the appointment will start and how long it will last. It also has a status attribute which is of type AppointedStatus which is an enumeration which can have a value of either: completed, booked, cancelled or inProgress. An enumeration is ideal for this attribute as it gives it a defined valid status that cannot be misinterpreted cause of a simple typo if it was a String for example. It also has a service attribute to define what services will be offered and provided by the pet sitter during this appointment. The class also has an Appointment() constructor method, a removeAppointment() method for cancelling the appointment and removing it from the calendar attribute or in other words schedule of any accounts involved, and a getLocation() method for getting the location of where the appointment will take place. We also have a pet class to represent each pet and any specific needs or preferences they might have. It has these attributes: name, species, breed, age, likes, dislikes and medicalConditions. These attributes all define what kind of pet it is, what they like, what they do not like and any medical conditions they have which may need to be taken into consideration. It has a pet constructor method and a getter and setter method for each of its attributes so that their values can be fetched or changed whenever needed. We also have a Review class to represent a review of a pet owner or pet sitter and are important for reputation scoring and ensuring the pet sitter and owner can trust each other. It has a rating attribute for the rating a pet sitter or owner is given, and a content attribute to allow the user to justify their rating and describe their overall experience. It has a Review constructor method as well as a getter and setter method for each of its attributes to allow the values of these attributes to be fetched or modified when necessary.

We also have a registry class which stores all the accounts and users. It has one attribute, the instance attribute of the type registry to provide a global point of access to the registry. It has a registry constructor method for creating a registry object, a getInstance() method for accessing the instance of that registry and a getAccount() method for fetching a certain account or list of accounts. We also have a message class to represent a direct message between two users. It has a content attribute to store the content of the message, a date attribute to identify when it was sent, a sender attribute to identify who sent the message and a recipient attribute to identify who received the message. It has a Message() constructor method for creating a Message object and a getContent() and setContent() method for fetching and modifying the content of the message so they're essentially getter and setter methods for the content attribute. It also has a getDate() method to fetch the date and time the message was sent at. We also used a MessageLog class to store multiple messages or a conversation between two users to allow us to access that conversation in case that information is needed for a criminal investigation for example. It has a simple constructor method MessageLog() to create an object of that class and initialise a conversation between two users.

Moreover, we also have an Administrator class to represent users within the system who are not customers of the app but are directly involved in its building, maintenance, and management. It has an Administrator() constructor method for creating an Administrator object, a verifyReviews() method to verify reviews after ensuring that no inappropriate content is contained within those reviews, a verifyUsers() method to verify that users are who they are after they provide some form of identification. It also has a verifyProofOfExperience() method to allow administrators to determine whether a proof of experience given to them is valid or not, and it also has a sendVerificationNotSuccessfulEmail() method to allow them to send an email to the user if any

verification involving that user was unsuccessful. It also has a conductDBSCheck() method to allow them to conduct a DBS check on a user and allow them to check if the user has a criminal record, and if they do, they can find out more details about their criminal record. We also have a Customer Service Advisor class which is a subclass of the Administrator class, and this represents an administrator within the app who is focused on ensuring the best customer experience for everyone and ensures people are staying within the law and the app's community guidelines. It has a banUser() method to allow them to ban users who have violated the app's community guidelines or the law, and it also has a restrictUser() method to restrict a user's access to the full app and its features. It has a resolveDisputes() method to allow them to resolve any customer disputes, and it also has a getMessageLogs() method to allow them to gain access to message logs between two users when that information is needed which could be during a criminal investigation for example. It also has a getGPSTrackingHistory() method to allow them to get the location history of a certain user when needed. It also has a getDisputeLogs() method to allow them to gain access to a list of different user reports or disputes. Another sub-class of the Administrator class was the Developer class which represented anyone who was helping with developing the app. This class had one method which was the getBugReportLogs() method to allow them to get bugReports() which contained information about bugs that had been noticed by other users, and this helped developers to know what to work on and what needs fixing. The class diagram also contains a Report class with a reportID attribute to uniquely identify that report, and a description attribute containing the report's contents which could describe a bug or customer dispute. It also has a Report() constructor method for creating a Report object. The report class had two sub-classes: the UserReport class which represented reports of customer disputes, and the BugReport class which represented any reports of bugs noticed while running the app. These two sub-classes both have constructor methods for creating objects of that defined class. We also included a GUI class which represents the interface the user uses to interact with the app, and it's also the class used to take input, interact with other objects in the system and deliver an output to the user who gave it the initial input. It essentially allows the user to interact with the app or system and is used in sequence diagrams to take an input and produce the right output.

Explanation of associations

One association we used was a 1:N association for the Account class to the Report class as each account can make multiple reports on disputes or bugs, but a report can only be made by one account. Another association we used was 1:N for the pet sitter class to the service class. The reason for this is that pet sitters can offer multiple services, but these services will only come from one pet sitter and won't be shared between pet sitters. Another association we used was 1:1 to N for the owner class to the pet class as each owner must have one or more pets and each pet must belong to one owner. We also used a 1:N association for the owner class to the appointment class as an owner can make multiple appointments, but appointments cannot be made by multiple owners. We also used a 1:N association for the pet sitter class to the appointment class as a pet sitter can take multiple appointments, but appointments cannot be taken by multiple pet sitters. We also made use of aggregation where multiple user objects would make up an aggregate registry object. We did this because each user and their details will need to be stored somewhere, and in this case, it was a single registry, and as a registry contains these user objects and their details and the user objects can exist independently of the registry object, it is therefore an aggregation because it's more of a Registry "has a" user relationship which is what defines an aggregation. Also, we used a 1 to N:N

association for the pet class to the appointment class as an appointment must be made for at least one pet if not more, and an appointment can be made to look after each pet multiple times. Additionally, we used a 1:N association for the PetSitter class to the Service class as each pet sitter can offer multiple services, but each service can only be provided by one pet sitter as pet sitters cannot share services among themselves.

Furthermore, we also used a 2:N association for the Account interface to the MessageLog class as each message log will be between two accounts, and each account can have multiple message logs. We also used a 1:N association for the Registry class to the MessageLog class as the registry can store multiple message logs whereas each message log is stored in one registry. Also, we used a 1:N association for the MessageLog class to the Message class as each message log can have multiple messages but each message can only belong to one message log. We also used two 1:N associations for the Account class to the Review class as each account can leave multiple reviews, but each review belongs to only one account. One of these associations is for dealing with a written review list which is a list of all the reviews a user has written, and one of these associations deals with a received review list which is list of all the reviews a user has received from other users.

Moreover, we used a 1:N association for the Account class to the UserReport class as an account can make multiple user reports, but a user report can only be made by 1 account. We also used a 1:0..3 association for the CustomerServiceAdvisor class to the userReport class as each customer advisor can deal with up to 3 user reports or customer disputes at a time, but each userReport or customer dispute can only be handled by one customer service advisor at a time. As well as this, we also used a 1:N association for the Developer class to the BugReport class as each developer can handle multiple bugs, but only one developer can be assigned to handle each bug.

Differences between domain model and class diagram

The first change that has been made is the hierarchical structure of the User. The structure in the domain model made it possible for objects of the Administrator class to be associated with an Account object, which is only meant for the app users that aren't the administrators, i.e., the Pet Owners and Pet Sitters. Now, the Account class is associated with a new class, AppUser, which is a sister class to Administrator. Thus, it is now impossible for objects of the Administrator class to be associated with an Account object.

Another difference between our domain model and our new class diagram concerns the Review class. In our domain model, the Review class was an association class between the PetOwner and PetSitter class. Our application allows both pet owners and pet sitters to review each other. This would mean that the PetOwner and PetSitter classes can be associated with the Review class in two different ways, as the reviewer and the user being reviewed. This is now reflected in the class diagram, where the Review class has two associations with the Account class, an abstract class that is the parent class of the PetOwner and PetSitter classes.

We have also changed the way the system will implement messages and direct messaging. In the domain model, there used to be only one class called Message that represented a direct message conversation as a whole. Now in the class diagram, it is a new class called MessageLog that represents a conversation between two instances of the AppUser class. A class that represents a singular text message is now called Message and will be associated with the MessageLog class.

The Registry class now is the aggregate of the User class instead of the Account class. There is also now a class called GUI for the graphical user interface of the system. Finally, the associations between Appointment and MessageLog, between MessageLog and Registry, and between MessageLog and Disputes have been removed.

Explanation of design patterns

Two design patterns have been used for our application's design: the singleton pattern and the player-role pattern. The singleton pattern is essential for creating a registry; there cannot be more than one registry in our system since it is more efficient for every piece of information and data to be easily accessible through one single registry. The singleton pattern assures that there will only be one instance of the Registry class by having the only instance of the Registry class be a static attribute of the Registry class itself and by having Registry's constructor be private, preventing the creation of any other instances.

Our application will allow users to have both a pet owner and pet sitter account under the same name. However, we must make sure that those two accounts can't interact with each other. The player role pattern offers an efficient way for our system to let our users switch "roles" from pet sitter to pet owner and vice versa. The AppUser class is associated with the Account class, which is an abstract parent class to PetOwner and PetSitter. Here, the AppUser class is the player, the Account class is the abstract role and the PetOwner and PetSitter classes are the roles.

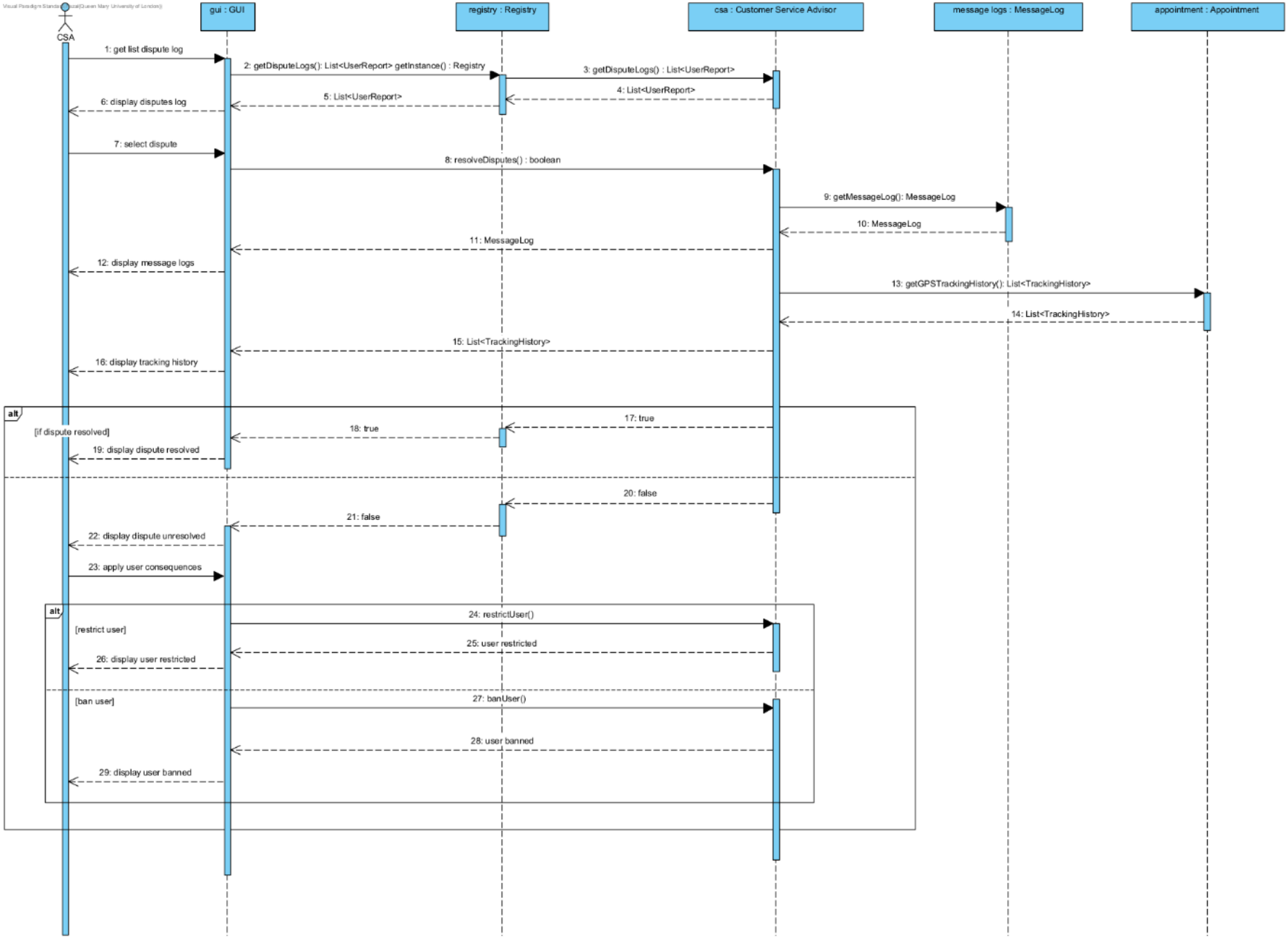
4. Sequence Diagrams

Resolving Disputes

The CSA gets at most 3 disputes from the user reports class and goes through them one at a time to resolve them by looking through the message logs between both users and the GPS tracking history of the pet. However, if the dispute is not resolved the CSA applies some consequences based on the severity of the users' actions for example the user could get restricted for a few days or banned from the platform. In the worst-case scenario, the CSA gets the police involved however this is not mentioned since it is an action out of the system.

Prerequisites:

- There must be disputes to resolve.
- The CSA must have permissions to access necessary files.
- The CSA must have permission to ban or restrict a user.

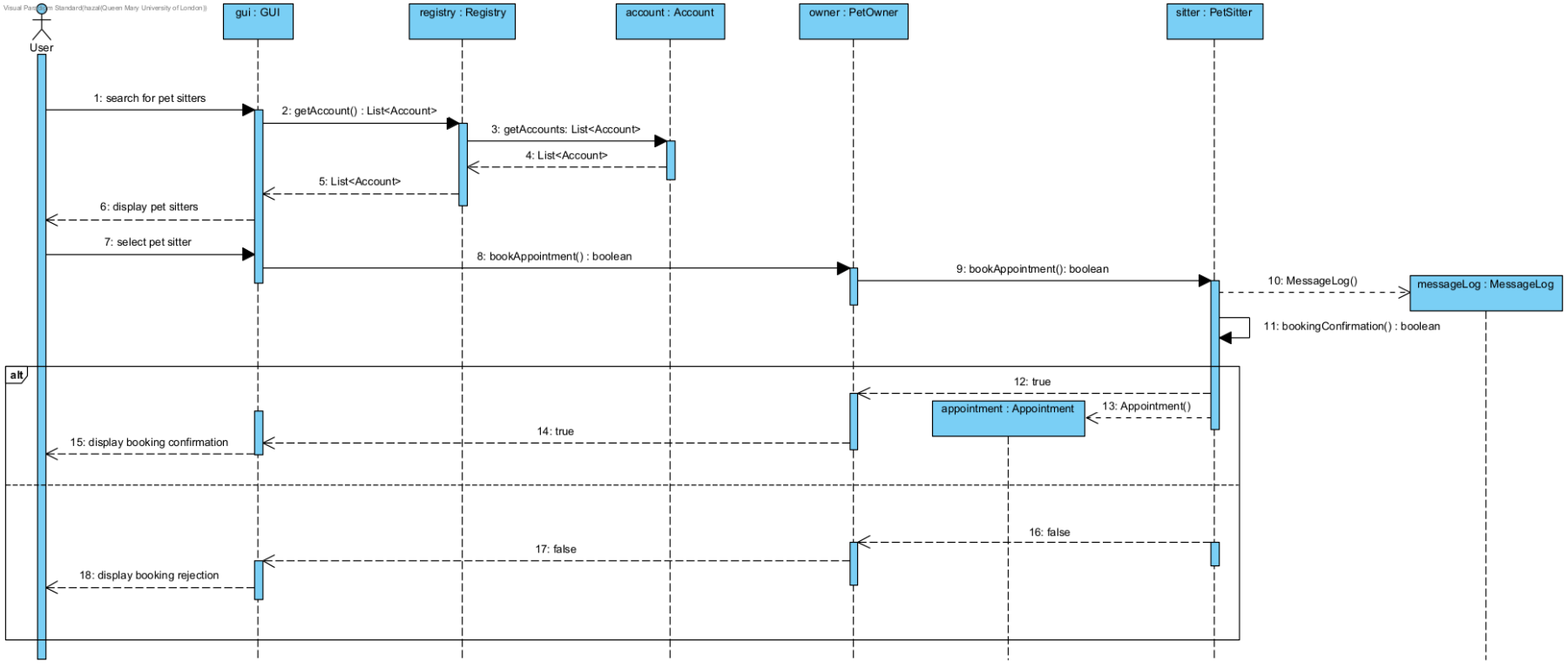

CSA
gui : GUI
registry : Registry
csa : Customer Service Advisor
message logs : MessageLog
appointment : Appointment
1: get list dispute log
2: getDisputeLogs(): List<UserReport> getInstance() : Registry
3: getDisputeLogs() : List<UserReport>
4: List<UserReport>
5: List<UserReport>
6: display disputes log
7: select dispute
8: resolveDisputes() : boolean
9: getMessageLog(): MessageLog
10: MessageLog
11: MessageLog
12: display message logs
13: getGPSTrackingHistory(): List<TrackingHistory>
14: List<TrackingHistory>
15: List<TrackingHistory>
16: display tracking history
alt
[if dispute resolved]
17: true
18: true
19: display dispute resolved
20: false
21: false
22: display dispute unresolved
23: apply user consequences
alt
[restrict user]
24: restrictUser()
25: user restricted
26: display user restricted
[ban user]
27: banUser()
28: user banned
29: display user banned

Booking an appointment

A pet owner wishes to find a pet sitter to take care of their pet dog Bifford. They log into the application and will search for a pet sitter that has a schedule that matches their requirements and can take care of their pet well, once a suitable pet sitter is found the pet owner sends a request to book and appointment and depending on the response of the pet sitter an appointment is created. The sequence diagram below shows how the system processes the request to find and book an appointment with the selected pet sitter.

Prerequisites:

- User is logged in.
- User uses the search page to find the pet sitter.



Cancelling an appointment

A pet owner has already booked an appointment for their pet, Bifford. However, their planned commitment has been cancelled, which means that they can now take care of their pet for free, meaning they do not need a pet sitter anymore. What they should do is find the appointment they want to cancel (in case they have more than 1). If the appointment is within the cancellation time frame, then it will get cancelled, if not, it is up to the pet sitter to decide whether they would still like to cancel the appointment, as their business will also be impacted by this situation. If all goes well, the appointment is then cancelled, if not, it is not cancelled.

Prerequisites:

- User must already be logged in.
- User must have an appointment already.
- Appointment must not already be cancelled.