Cupet Phase 1 Progress Report

Group Members (Group 18):

Andrew Qiu
Daniel Baek
Fangyi Li
Harry Xu
Katherine Jelich
Kenneth Tran

Specification:

Summary:

Cupet is an Android app with two parts: a frontend and a backend. The frontend is where a user interacts with the app. The backend is responsible for storing, processing, and relaying data from and to the frontend. Users of the app can register and log in to the app using their email and password. To register, a user needs to provide details, including their full name, email address, home address, and city. Each user has their own user profile.

Each user can have multiple pets, which each have their own pet profile. For each of their pets, the app displays to the user a list of other pets—or, potential matches for that pet. The potential matches are displayed based on location, as well as the preferences that the user has selected (ex., types of animals they want their pet to be matched with). Users are able to select which pets they are interested in matching with from this list.

If two users have selected each other's pet, they have matched with each other, and their pets are added to each other's match list. The two users are then able to view each other's contact information on each other's profiles.

New Functionality:

Backend:

Implemented Use Cases and Endpoints:

- Session Token (JWTs) Generation and Validation
    - Upon user login, generate a JWT token for the user. This token is passed in the header of every HTTP request requiring authorization
- Create Pet
    - Create a pet with given basic pet information (e.g.name, age, breed, bio). Return a pet id for further fetching and editing.
- Edit Pet Profile
    - Edit pet's name, age, breed, bio with given new information. Return if editing is successfully done or not
- Fetch Pet Profile
    - Fetch a pet's profile information (e.g. name, age, breed, bio) given the pet's id. Fetching may fail if an invalid id is given.
- Edit User Profile
    - Edit a user's profile information (e.g. bio, contact info, etc) given new information. Return if editing is successfully done or not.
- Fetch User Profile
    - Fetch a user's profile information (e.g. bio, contact info, etc) given the user's id. Fetching may fail if an invalid id is given.
- Edit User Account

    - Edit a user’s account given the user's id and new information (e.g. name, password, email, etc). Return if editing is successfully done or not.
- Pet Swiping
    - Given two pets, add Pet2 to Pet1’s swiped list. If both pets have swiped on each other, match them by adding them to each other’s matched list
- Pet Unswiping
    - Given two pets, remove Pet2 from Pet1’s swiped list
- Fetch Pet Swipes
    - Return a list of pets that a pet has swiped on
- Fetch Pet Matches
    - Return a list of pets that a pet has matched with

Frontend:

Working pages:

- Login page
- Registration page

Work in progress:

- User Pet Slots page
- User Account settings page
- Create Pet page
- Pet Profile Pages for Potential Matches
- Edit Pet Profile page
- Edit Pet Preferences page
- Successful matches page

API Interactions:

- Implement interactions needed (requests and response handling) for all backend endpoints except pet unswiping

Major Design Decisions

In our design, we prioritized the application of the SOLID design principles and a clean implementation of clean architecture. When expanding upon our work from Phase 0, our priority was to allow for our program to continue to be aligned with SOLID and clean architecture. In this way, as different aspects and layers of our program are implemented, we can expand upon our previous code with fewer conflicts. As we experienced while building upon our Phase 0 code, an emphasis on adhering to these principles allowed for us to have a clear strategy when implementing new features. Our program has an evident structure that is mirrored throughout the implementation of different features. In this way, when we improve and expand our design for Phase 2, when considering design principles that we wish to implement, we will have a clear and replicable structure to apply them to.

For Phase 1, a lot of the structure of our code was based on what was written for Phase 0. For the backend in particular, we mainly dealt with expanding the program's functionality by introducing new endpoints and their accompanying use cases.

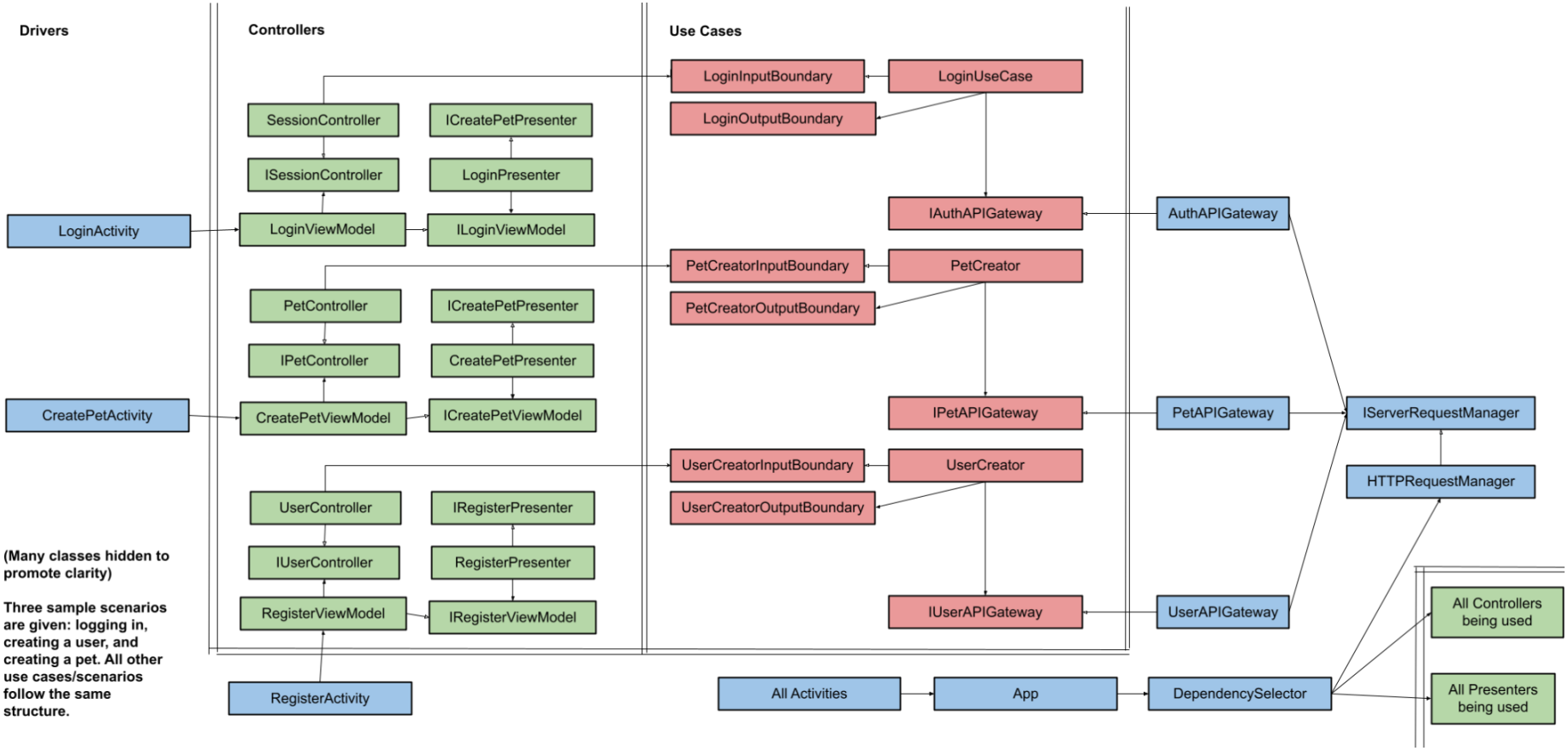
The frontend codebase is structured similarly to the backend. As all the frontend code is new in Phase 1, this gave us an opportunity to write better-structured code based on what we learned from writing the backend. For example, one design decision we made for the frontend was to split (wherever possible) classes, methods, and interfaces into those that pertained to users, pets, and authentication separately. This can be seen in our packaging strategy, which separates pet and user classes (like in the use_cases.request_models package). More significantly, we also made the design decision to separate the APIGateway class to PetAPIGateway, UserAPIGateway, and AuthAPIGateway to better conform to the Single Responsibility Principle for the frontend. Although the APIGateway class in the backend still exists, we plan to split that class as well but were unable to do so for Phase 1 due to time constraints.

Another major design decision we made was to use the Chain of Responsibility design pattern for our frontend. By having to probe our API for data, our frontend has to be able to handle both successful and failed responses and be able to work asynchronously. Instead of condensing success and fail responses into a single response object, which can make typing complicated if the success response and error response bodies look significantly different, we decided to separate success and error responses into separate response models and into separate callback methods (ex. onSuccess, onFailure). These callback methods would then be chained from API responses -> use cases -> presenters -> view models (as shown in the Chain of Responsibility design pattern), allowing us to transform our responses in steps. This is discussed more in the design patterns section.

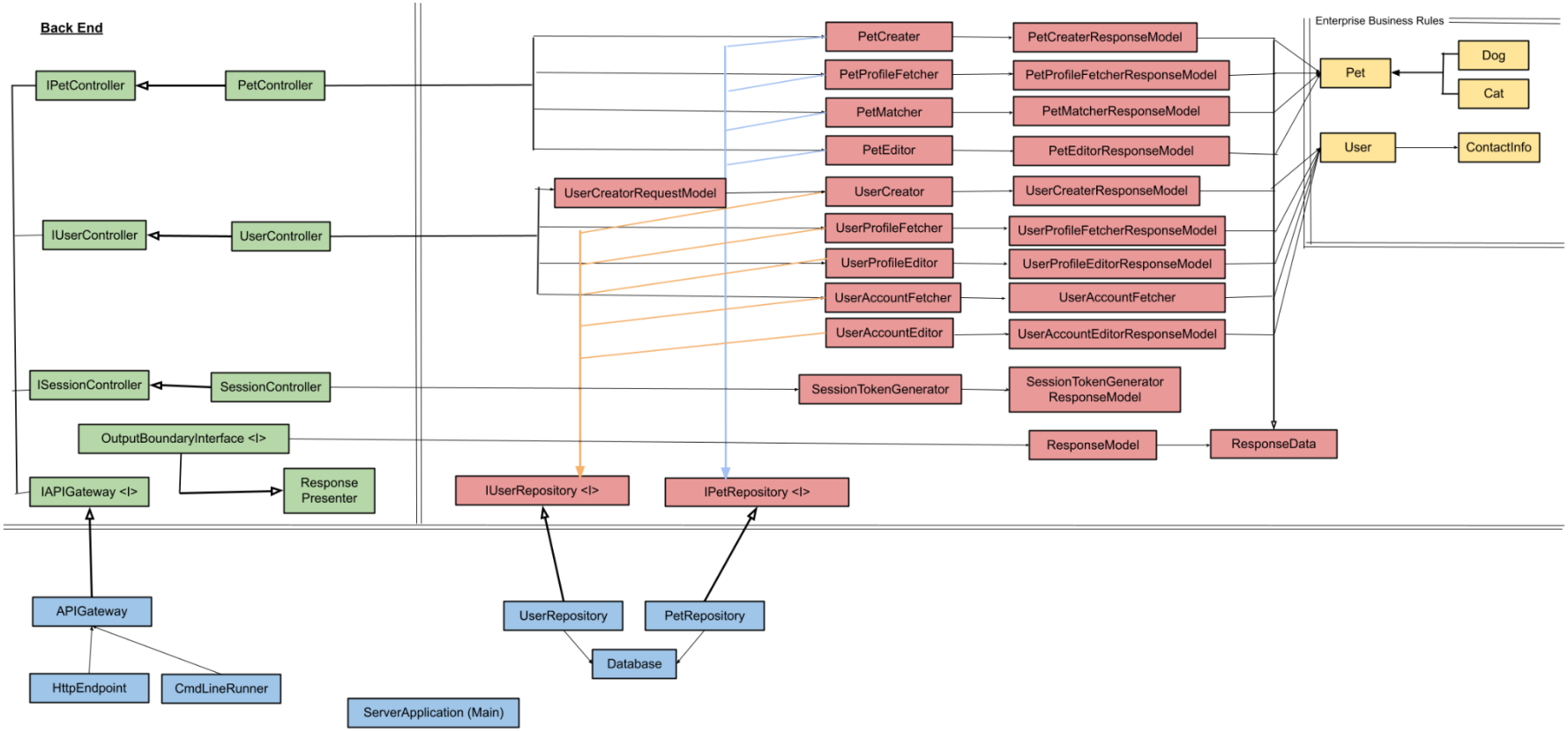
Our intentions now are to continue implementing desired features and to analyze and implement different design patterns that are appropriate for Phase 2. In this way, we hope to build a final product that is designed effectively and allows for expansion.

UML Diagrams

Frontend Diagram



Backend diagram:



SOLID Design Principles

Examples from Backend:

- **Single responsibility principle(SRP):**
  - For the most part, our classes followed this principle since we tried to create a single class for each functionality. For example, we separated each use case into its own classes as well as their response and request models so that any modifications to a use case can be easily tracked. In addition, we made sure that the only purpose of the controller class was to call use cases using a request and to return the response. Since the controller classes don't do anything else, the only time they would need to be modified is when adding a new use case or when changing the request arguments for an existing use case.
  - A few select classes that are currently violating this principle are the IAPIGateway interface and its implementation APIGateway, as well as the classes that depend on IAPIGateway, which are CmdLineRunner and HttpEndpoint. Currently, APIGateway defines every single API endpoint of the program in a single class. This means that each time we need to modify, delete, or add an endpoint, this class and all of its implementations and dependents must be modified. For Phase 2, we can fix this by splitting APIGateway into multiple classes that encapsulate a specific domain. One choice is to split them by the type of action, e.g. create, read, update, delete.

- **Open/closed principle(OCP):**
  - As the backend has examples of other SOLID principles such as the Liskov Substitution Principle and the Dependency Inversion Principle explained below, many of these classes also naturally follow the Open/Closed Principle. For example, the superclass ResponseData is a class that is not meant to be instantiated or modified but rather inherited by another class to create new models of response data for a new use case. Another example of this is the interfaces IPetRepository and IUserRepository. These interfaces are meant to define what methods any given implementation of a database must implement, so we can always add a new implementation of a database simply by having it implement the interfaces. These classes are therefore closed for modification but open for extension.

- **Liskov Substitution principle(LSP):**
  - One example of the Liskov Substitution Principle that appears in several classes is the request model inheritance structure. We have defined a superclass ResponseData, which the response data model of each use case extends. An arbitrary ResponseModel will thus contain an instance of a ResponseData that can always be safely downcasted into a response data object. This was shown in a few of the test cases. For example, in the TestUserCreator test unit, the User Creator use case returns a ResponseModel object, whose ResponseData is then

downcasted into a UserCreatorResponseModel in order to extract the fields that are specific to that subclass.

- **Interface segregation principle(ISP):**
  - In our backend, every concrete class implementing an interface always makes use of the exact methods that the interface defines. This way, no class is forced to implement some irrelevant method and then raise an Exception. We achieve this by defining many small interfaces that are meant for specific tasks. For example, we have many small input boundary interfaces for the use cases, and we have a UseCaseOutputBoundary interface in the Interface Adapters layer that a ResponsePresenter, whose job is to format response data out of the controllers, implements. On the other hand, while it is the case that the APIGateway class only uses the methods defined by its interface IAPIGateway, there is still some violation of the principle since the interface is too large.

- **Dependency inversion principle(DIP):**
  - Nearly every class, regardless of the layer in clean architecture that it belongs to with the exception of entities, is implementing an interface that declares the methods it should implement. Any dependencies of these classes are actually dependencies of the interfaces that they implement, allowing us to have both high and low-level modules depending on abstractions. This is best demonstrated by the IUserRepository and the IPetRepository interfaces. These two interfaces define the relevant methods for fetching entity information from a repository but do not instruct how the repository is defined. In the test cases, we have a DummyPetRepository class and a DummyUserRepository class that implement the respective interfaces. These classes simulate a database by storing values in memory, which is appropriate for testing purposes. In the actual program, we have a PetRepository and a UserRepository class which also implements the interfaces; however, they hook into the Spring JPA framework, which accesses a real, remote database online. Thus by having the interfaces, we are not enforcing any specific implementation of a database which allows for loose coupling.

Examples from Frontend:

- **Single responsibility principle(SRP):**
  - All classes in the app are responsible for only one thing. Classes that originally had multiple responsibilities were split into multiple separate classes. For example, we split all the API gateways into UserAPIGateway, PetAPIGateway, and AuthAPIGateway. We did a similar process for the DependencySelector class as well.

- **Open/closed principle(OCP):**
  - We relied on interfaces a lot on the frontend, such that most classes (who had a responsibility beyond just storing data) implemented some interface. (ex. UserAPIGateway, which implements IUserAPIGateway). These interfaces would be used as the types for objects instead of the classes themselves. Therefore, if we wanted to extend the functionality of our program in context to a specific class, we could simply make a new class that implements the relevant interface and inject the new class instead. This would allow us to extend functionality without changing the code of the old class

- **Liskov Substitution principle(LSP):**
  - Our frontend does not use much inheritance. In places where it is used, like with RoutesStore and UserRoutesStore and APIGateway and UserAPIGateway, we do not change/override the functionality of the superclass at all. Therefore, if we were to replace some instance of the superclass with a subclass, it would retain the same functionality.

- **Interface segregation principle(ISP):**
  - The frontend is structured very similarly to the backend in terms of interfaces. Throughout the frontend codebase, we use a multitude of small interfaces with few methods necessary only for that interface. For example, UserCreatorOutputBoundary is an interface that exposes the only two methods necessary: a method for catching a successful response and a method for catching a failed response. As most of our functional classes are based around these interfaces, this allows us to limit the number of public methods we expose to a user, thereby making the public interfaces of our classes as small as possible.

- **Dependency inversion principle(DIP):**
  - Most of our classes (outside of a select few like DependencySelector) contain dependencies only on interfaces, whose implementations are injected during runtime. Interfaces are also placed between dependencies between layers where applicable. For instance, we have interfaces between:
    - ViewModels and Presenters (I...ViewModel)
    - Classes in drivers layer and Controllers (I...Controller)
    - Use cases and controllers (input boundaries)
    - Use cases and presenters (output boundaries)
    - Use cases and API gateways (I...APIGateway)

Design Patterns

Implemented Design Patterns

Our codebase on both the frontend and backend employs the Dependency Injection design pattern nearly everywhere. To name a few:

- We use dependency injection with use cases and input boundaries to avoid having controllers directly depend on use cases.
- We also use dependency injection with IUserRepository and UserRepository for allowing use-cases to access our databases in the driver's layer without depending on them.
- We use dependency injection with IUserAPIGateway and UserAPIGateway for allowing use-cases to make API calls in the driver's layer without depending on them.

Dependency injection is all over our codebase and can be seen in our UML diagrams. There is no one pull request for this as this is all over our project.

For the frontend, we use the Chain of Responsibility design pattern for forwarding our response data throughout the API, use cases, and presenter classes in our project. This can be seen in our handler methods (onSuccess, onFailure) in output boundaries, view model interfaces, and in IServerResponseListener. As an example, see the frontend's PR #3. As a general structure, the chain of handlers in the frontend looks like this: API response handler -> use case output boundary handlers (presenters) -> view model handlers. This allows us to handle our data in steps. The chain of responsibility design pattern allows us to handle asynchronous requests (through the handler/callback methods, which are called once a request has finished), which are needed in context with sending requests over a network (which takes time). It also allows us to process our responses step by step. In the registration/create user example, we first retrieve a JSON response, reconstruct it into a Java ResponseModel object, remove/reformat data in the presenter, and finally pass it onto the view model where it is stored for display.

We also used the Observer design pattern in the frontend for interactions between the view model and view (although we didn't implement this ourselves as it was provided by Android libraries). You can see this in action in the same pull request (RegisterView and RegisterViewModel). We employ the observer design pattern by making the RegisterView observe the state of the RegisterViewModel such that when the state of the view model changes, our register view model can be notified and react by displaying the necessary information to the user.

Planned Design Patterns

We plan to eventually support multiple types of users and multiple types of pets. For example, we may extend our app by having classes such as PremiumUser, CommonUser, or VerifiedUser that extend the User class, and classes such as Dog, Cat, or Bird that extend the Pet class. To create these different objects, we plan to use a factory, allowing us to obscure the creation project of different types of users/pets. During the pet profile creation process, we can redirect all registered data to a factory class that is responsible for instantiating different Pet objects. The factory method could also take in one subclass of a User and convert it into another subclass of a User. For example, when a user signs up, they are assigned as a CommonUser, but if they pay for a subscription, they will become a PremiumUser. The factory method would take a CommonUser as input and return an instance of PremimumUser, but keeping the same attributes.

Additionally, we are currently using the “new” keyword to construct objects in the backend, which has led to some repeated, messy code. For example, in DummyUserRepository and UserRepository, which both implement the fetchUser method defined by the IUserRepository, we construct an instance of User from the database User object which is specific to the database implementation. The new User object must then have all of its properties injected by this database User manually. This means that each time the User object is modified, we need to look into every single implementation of fetchUser and update the method. Instead, a factory method could handle these details, and any changes to the User object would only require us to update the factory method.

Now that we have our basic structure in place for the frontend, we will look into how we can implement different design patterns that are appropriate for our application. There are common structures between different views and activities where we can potentially use design patterns such as command or strategy. We will look into how we can use design patterns to solve different situations we encounter in our design.

For example, design patterns such as the template method design pattern can be useful for encapsulating shared behaviours between different screens. Suppose we have some shared behaviour between certain screens, but the specifics of this behaviour are different. In that case, we can have a base class that defines the structure and allow for derived classes to characterize and differentiate this behaviour according to the behaviour of the specific screens. Implementing design patterns such as this one would allow our program to be more open to extension since it allows for more screens with shared behaviour to be added without significant modifications to the code already in place.

Packaging Strategies

For both the frontend and backend, we split packages based on the layers of clean architecture (entities, use cases, controllers, drivers, etc.). We chose this because it made it easier to visualize in our code where our dependencies were. For example, this packaging strategy could explicitly show (through the imports) whether or not we were importing code from the correct layer to conform to Clean Architecture. The frontend also grouped related objects together inside these layer-based packages. For instance, we grouped request models together in a package and response models together in a separate package. Classes were grouped again if a package became too large (ex., Grouping pet use case request models together).

Use of Github features

For this phase, our group made an effort to use the features of GitHub more effectively. Our group has two repositories: one for the backend of our program and one for the frontend of our program. Different members of our team were focused on the frontend and the backend. Within the two repositories, team members selected and were assigned different tasks and classes to implement.

Our team made an effort to have standard naming conventions that are relevantly titled for our different branches and to appropriately describe our commits. We worked to make it so that as our project progressed, it would be clear and understandable what tasks had been completed and what changes were made with each commit. Thus, different stages and implemented features of our program can be followed through our commit history. We made an effort to clearly separate our branches based on tasks such as implementing features and bug fixes (ex. feature/<my_feature> and bugfix/<my_bugfix>). This also allowed our progress to be well organized and for it to be clear to each group member what their responsibilities were and what tasks needed to be completed.

Another aspect of GitHub which we made an effort to effectively use were pull requests. We used pull requests to merge into our main branch by ensuring that each request was sufficiently reviewed by at least one or two other group members (preferably while in voice call). Each pull request and subsequent merge (when appropriate) was documented, and a summary of changes was made, including a description of how the changes were tested. In this way, we could document our progress and determine what we still needed to do.

More recently, we've tried to use GitHub issues to document bugs we encounter. Although we've only had a few issues up on the backend, it has made tracking bugs we encounter way easier. It has helped us remember things we need to fix and allow people who depend on this code to know when and where a fix is available. This is a significant improvement over our old method of simply sending a message in our group chat, as bugs and issues can quickly get lost after a few texts.

For Phase 2, we plan on trying out GitHub actions to autorun our unit tests. This can hopefully give us a way to recognize from which commits errors or new bugs stem from, and allow us to more easily recognize when something in our codebase is broken.

Testing

Tests are available for all of the endpoints on the backend. Tests are available but sparse for the frontend, as we ran into time limitations. We plan on including more unit tests for our existing classes within the next week.

There is one major test suite for the frontend at the moment, which you can find in the androidTest folder. These tests include integration tests that test the frontend code and the format of the HTTP requests and responses. This test suite currently covers the vast majority of endpoints available on the backend and their corresponding API interactions on the frontend.

Refactoring

Refactoring was done throughout the project as issues, code smells, or cleaner ideas for implementation came up. As an example for the backend, PR # 22 deals with refactoring the project as a whole. To name a few changes, this PR moved functionality (like authentication) to classes that better fit it and used superclasses to fix the Duplicate Code code smell. Refactoring in the frontend was done less explicitly, as features weren't built to working versions until very recently. The commits in frontend's PR #3 demonstrate refactoring the Android template registration page code into code that better fits our codebase structure and style.

Progress Report

What Worked Well:

We have, throughout this project, had an emphasis on making sure that our design and implementation adhere to clean architecture and solid design principles. We have found that now at the stage in which we are expanding our project by adding new functionalities and features, our investment in these principles and concepts has allowed for that process to go much more smoothly. There has been limited modification to the code that we submitted for Phase 0. We have not had to spend a significant amount of time returning to our initial work to restructure it to accommodate our new additions.

On a related note, we invest time into collectively understanding the different parts of our project as a group. This has allowed us to consider different solutions to the problems we run into. Additionally, this has resulted in us not having to frequently rework our own or each other's code in order to fit in with the larger design. Our initial implementations, with review, are often effective and sufficient for our current design plans since, as a group, members are involved in the planning process and thus have a good understanding of where the tasks they are responsible for fit into the bigger picture of the functions of our program.

Group Member Workload Distribution Summary:

Andrew Qiu:

- ➢ Building the frontend structure and Android app (API interactions, login page, register page, controllers, use cases, etc).

Daniel Baek:

- ➢ Building the frontend Android app (view profile page, pet profiles for potential matches, etc.).

Fangyi Li:

- ➢ Building the backend features (edit/fetch users, edit/fetch pets, etc).

Harry Xu:

- ➢ Building the frontend Android app (pet profile page, view potential match page, edit account setting, etc)

Katherine Jelich:

- ➢ Building the frontend Android app (login page, register page, successful matches page, user profile page, etc)

Kenneth Tran:

- ➢ Building the backend features (session control, matching, swiping/unswiping, etc)

What each group member plans to work on next:

Kenneth and Fangyi will continue working on the backend, adding the endpoints necessary for the functions we would like to implement in the frontend. In addition to this, they will expand the backend to incorporate new features we would like to include. Furthermore, they plan to analyze the current structure and implement relevant design patterns.

Andrew, Harry, Daniel, and Katherine will continue to develop the frontend of our program. They plan to work on integrating different design patterns that would be appropriate for our app. They will continue adding the activities defined by our backend to our android application. More specifically, Andrew will continue working on the API interactions and related functions. Harry will work on implementing the potential matches feature (i.e. 'swiping' to match with other pets) and creating and editing pets and editing user account settings. Katherine will continue working on the successful matches page and related pages and features, as well as viewing and editing user profiles. Daniel will work on making the pet profiles and profiles for potential matches.

Open Questions:

- The frontend use cases often only fetch partial data relating to our application's main entities. For instance, we might only fetch a user's profile, but not their home address and city. Therefore, it does not make much sense for us to construct a whole User entity for most -> all of our use cases, as key data in that entity would be missing anyways. We currently do not have a use for entities in the frontend. Is it necessary that we include them?
- We currently are not using GitHub to its fullest potential, but we plan on using Actions for our work in Phase 2. What are your suggestions for features on GitHub that might be helpful to us?
- Some of our data classes contain a large number of fields. For instance, our UserCreatorRequest model stores firstName, lastName, email, password, homeAddress, and city. Initializing all these fields at once in a constructor results in a long parameter code smell. However, initializing each field through getters will make our code repetitive and not much clearer. Does this matter? What are your thoughts on this?