Cupet Phase 2 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. Users are able to select which pets they are interested in matching with from this list.

If two users have selected each other's pets, 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 (Since Phase 1):

Backend:

New Use Cases and Endpoints:

- Set User/Pet Profile Image
    - Set a user or pet's profile image by passing in a Base64 Encoded String, which is uploaded to a remote image hosting service
- Fetch User/Pet Profile
    - Fetching user/pet profile image is now combined into the fetch user/pet profile use case
- Pet Matches Generator
    - Computes a list of pet ids of potential matches for a given pet based on geocode location, instead of simply grabbing all pets

Refactoring:

- Reorganized messy packaging
- Combined PetSwiper, PetRejector, PetUnmatcher, and PetMatcher into a single use case PetInteractor, using Enums to reduce code duplication
- PetMatchesGenerator no longer pings the Geocoding API during each use, instead, latitude and longitude are lazily generated upon user creation and user account editing and stored in the database

Frontend:

New features:

- Add all pages needed for the App
    - Login page
    - Registration page
    - View my pets page
    - Account Settings & Edit Account Settings Page
    - Create Pet Page
    - My Pet Profile
    - Other Pet's Profile
    - Successful Matches Page

API Interactions:

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

Refactoring:

- Fixing duplicate code with subclassing and delegation
- Introduce the Dagger library for more readable & cleaner dependency injection

Major Design Decisions

In our design, we prioritized the application of the SOLID design principles and the 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.

In Phase 2, the majority of design decisions made for the backend were related to refactoring the code rather than changing the core functionality. For example, we split long classes into smaller, more focused classes, such as the HttpGateway class, which originally contained every single API endpoint, into WebGateway, PetWebGateway, UserWebGateway, among others. We also decided to remove the APIGateway and CmdLineGateway classes which manipulated the controller classes since they were very long, opting to access the controller classes directly instead.

One major design decision for frontend in Phase 2 was the restructuring of the request/response models. From Phase 1 feedback, we noted that there was a lot of duplicate code across our request/response models as they often passed around the same type of data. To solve this, we used superclassing to share instance attributes and getter functions across multiple different request/response models simultaneously. One downside of this is that there is increased coupling as many request/response models now rely on superclasses. Although this is the case, we also noted that a lot of request/response models should be inherently connected. For instance, if we were to change what constitutes a user profile, it's natural that both the fetch user profile request model and edit user profile request model should be updated. So the issue of increased coupling is not super significant.

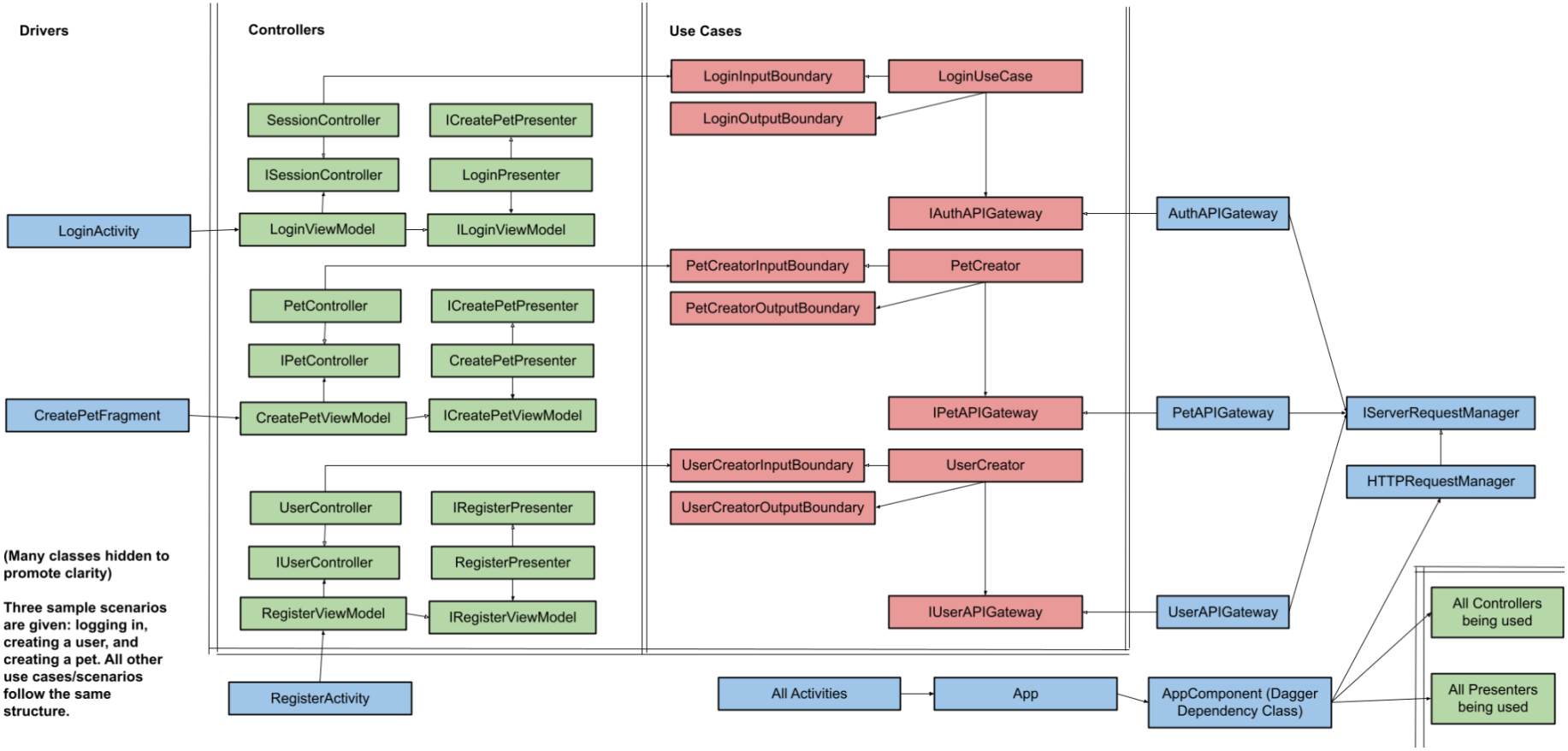
Another major design decision for the frontend in Phase 2 was to offload text validation from view models to dedicated form validator classes. In our app, our view model classes store the states of our various pages (data stored in the page, the state of any editable forms, etc.). In Phase 1, we had this data validation functionality as private methods (like validateFirstName) inside our view models. We noticed that this was a mild violation of SRP: the view model is responsible both for representing the page, and for determining whether or not entered data is valid. Furthermore, as we built more and more pages, we wanted to keep our validation requirements consistent: the data validation on field for editing a pet should be the same as creating a pet. Thus, we refactored our code and introduced validator classes, whose sole purpose was to validate the data we indeed. The individual view models would then delegate the validation functionality to these validator classes, thus solving our problem with SRP and keeping our data validation requirements consistent.

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

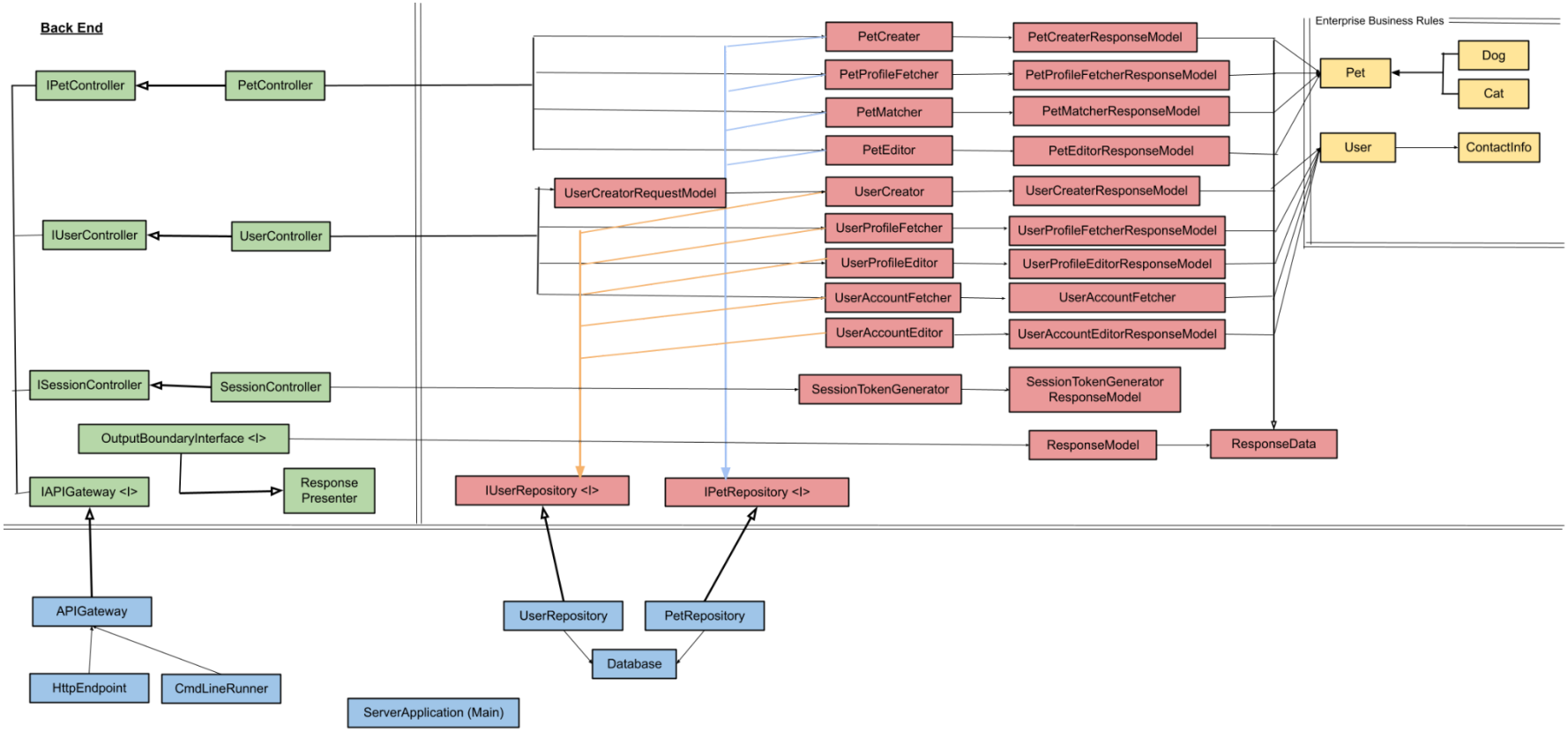
separate callback methods (ex. onSuccess, onFailure). These callback methods would then be chained from API responses -> use cases -> presenters -> view models (as shown in the Chain of Responsibility design pattern), allowing us to transform our responses in steps. This is discussed more in the design patterns section.

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.

- **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, for each use case, we define a small input boundary interface, and

each class providing a service (e.g. ImageService, GeocodingService, JwtService) is implementing an interface that tells it exactly what methods it must implement.

- **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. As a newer example for Phase 2, we fixed SRP issues in our view models by delegating tasks (like data validation) to smaller, dedicated validator classes.

- **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 mainly uses superclassing for sharing functionality across multiple classes. Therefore, we don't do a lot of method overriding. The behaviour of all subclasses is thus exactly the same as any superclass.

- **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 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.

While one of the planned design patterns in the backend was to use the Factory method for creating different types of Pets and Users, due to time constraints we were unable to proceed with our original plan of having different types of Users such as VerifiedUser and PremiumUser, or different subclasses of Pet for different species. Instead, we are simply using the User and Pet classes as concrete classes rather than abstract classes and instantiating instances of these classes. Since this design decision renders the Factory pattern useless in this case, we decided to switch to a Builder design pattern for User and Pet creation. This is more suitable since both Pet and User have many optional fields that are not required upon object creation, such as phone number and social media information for User, and biography for Pet. This is, in our opinion, a good alternative since it still eliminates the use of the "new" keyword by directing all object creation to the classes UserBuilder and PetBuilder. This design pattern was introduced in the backend's PR #33.

Another new design pattern seen in Phase 2 is the Strategy pattern in the backend. Previously, PetSwiper, PetUnswiper, PetRejector, and PetUnmatcher were each a single use case. These use cases are very similar in that they require two pet ids as inputs, but only differ in a handful of lines of codes. To reduce the need to create input and output boundaries and request and response models for all four of these use cases, we decided to merge them into a single use case called PetInteractor, which contains a PetInteractionContext that stores a reference to the current strategy and executes the strategies. The PetInteractorRequestModel is the object that passes the desired strategy in an enum to the PetInteractor, which is responsible for passing a reference to the strategy into the PetInteractionContext. This design pattern was introduced in the backend's PR #41.

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.

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 occur 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 used GitHub actions to autorun our unit tests. This helped us recognize from which commits errors or new bugs stem come from, and allowed us to more easily recognize when something in our codebase is broken.

Testing

Tests are available for all of the endpoints on the backend, and for most of the classes as well. On the backend, we currently have 80% test case coverage for classes. Test units mostly consist of testing different inputs to use cases, and are relying on dummy repositories and services which are decoupled from the real frameworks and drivers.

Tests are available for all use cases in the frontend, and for all functional classes (like data validators, HTTP request managers, JWT parsers, etc.). We ran into time constraints, so we did not test classes whose methods were trivial (ex. Classes that just forward data). There is also one major integration test suite on the frontend located in androidTest/integration_tests. These tests include integration tests that actually test that the frontend and backend can communicate.

Refactoring

Refactoring was done throughout the project as issues, code smells, or cleaner ideas for implementation came up.

As many of the core features were already implemented in the backend during Phase 0 or Phase 1, Phase 2 mostly consisted of refactoring. Some backend examples are PR #28, which fixed some violations of clean architecture, added more interfaces for dependency inversion, among other assorted fixes such as optional request bodies in the REST API, password encryption, and HTTP response statuses. A later example is PR #41, which tied up loose ends by adding missing javadocs, removing unused code, and fixing previously undetected bugs.

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. In frontend's PR #33, we refactor the whole dependencySelector class with the Dagger dependency framework. Dagger helps us to inject dependencies into our classes in a much cleaner way (by using Java annotations). Another example of refactoring is in PR #38, where we refactored our view models to delegate data validation responsibility to a dedicated validator class.

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. As a result, we have found that during the transition from Phase 1 to Phase 2, we were able to expand our project by adding new functionalities and features smoothly thanks to our investment in these principles and concepts. There have been limited modifications to the core code that we submitted for Phase 0 and Phase 1. 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 invested 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 reviewing, 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.

For Phase 2, we also used more tools for planning and structuring our app. For example, to track our progress and to outline what tasks we need to do, we used Trello. As a result, we were better able to better plan our time and set deadlines. We also used Figma for drafting a sample user interface for our Android App. This made it significantly easier to create the individual pages in Android as we had something concrete to refer to (ex. it outlined exactly what fields and buttons we needed).

Group Member Workload Distribution Summary (since Phase 1):

Andrew Qiu:

- ➢ Updating the frontend structure and Android app (splash page, potential matches page, linking pages together).

Daniel Baek:

- ➢ Building the frontend Android app.

Fangyi Li:

- ➢ Building the frontend Android app (pet profile page, create pet page, edit pet page, etc).

Harry Xu:

- ➢ Building the frontend Android app (user profile, user account, etc)

Katherine Jelich:

- ➢ Building the frontend Android app (successful matches page, etc)

Kenneth Tran:

- Updating and refactoring the backend, assuring backend endpoints are consistent with frontend features

Group Members' Significant Pull Request

Andrew Qiu:

- Frontend PR #28: Introduced the potential matches page and made sure that it correctly worked with the controllers and handled API request success/failure conditions
- Frontend PR #33: Refactored our existing implementation of injecting dependencies (DependencySelector) with a more intuitive dependency injection framework (Dagger)

Fangyi Li:

- Frontend PR #25: PR #25 on the frontend includes files for presenting the pet profile page in the app.
- Backend PR #17: PR #17 on the backend implements user-related use cases and tests.

Harry Xu:

- Frontend PR #34: PR #34 on the frontend includes the implementation of user account settings, edit user profile and user profile fragments.
- Frontend PR #37: PR #37 includes the implementation of the edit user account fragment.

Katherine Jelich:

- Frontend PR #26 and Frontend PR #27: PR #26 on the frontend includes the fragment for presenting a pet's matches. PR #27 on the frontend includes the fragment for presenting a user's pets, and these are two significant pages in the app.

Kenneth Tran:

- Backend PR #30: Pull request 30 on the backend is where image fetching and uploading is implemented.
- Backend PR #37: Pull request 37 on the backend concerns general refactoring and performance optimization

Project Accessibility Report

Principle 1: Equitable Use

- (Frontend) Our android application supports a minimum Android API level of 17, allowing us to support devices from 2012 onwards. Therefore, it retains the same functionality for users with older/newer devices.
- Our frontend could do a lot better in this regard however. For example, to be more equitable with people with blindness, we can implement a text to speech feature to read out text and button labels. We could also use Google APIs to implement translation features to reach individuals who speak different languages.

Principle 2: Flexibility in Use

- Due to time constraints, our app is not customizable. In the future, we could allow users to be more flexible in their use by adding app-specific features like a custom theme. We could also introduce more functionality to our program overall by adding features like differentiating what types of pets you want to match with. This way, a user can use our application the way they like (maybe they just want to match with dogs, cats, or both).

Principle 3: Simple and Intuitive Use

- (Frontend) We use many simple and intuitive images like pencils, hearts, etc. The main component of the app mostly follows Google's Material Design, which should make the app intuitive for android users. For buttons which are less intuitive (like the potential matches button), we label them accordingly so that the user can easily understand what each button does.

Principle 4: Perceptible Information

- (Frontend) In many views, information is presented in multiple different ways. For example, we represent a pet with both text and images.
- (Frontend) Views are designed to allow for essential information to be visible and legible. The pages were designed to not be overcrowded with information, and essential information is presented in such a way that it is more visible than unessential information. For instance, the profile pictures take up the majority of screen real-estate, followed by a large heading for the name, and less important information like the biography in smaller text.
- As we noted before, using a kind of text to speech/dictate feature could also help, as it would provide a different way to interface with the information we provide the user.

Principle 5: Tolerance for Error

- (Frontend) in activities where there are actions that cannot be undone or amended, a confirmation popup is displayed which makes clear the consequences of confirming the action. For example, on the page which displays

a pet's matches, there is a button to delete a particular match. If this button is selected by a user, they are prompted by a popup to confirm their intention. Since this action is not reversible, this confirmation is in place to allow tolerance for errors.
- Furthermore, editing pages for pets/users are also at least two clicks away using our app's navigation. We add a confirmation button for each page such that we minimize accidentally changing your pet/user's information.

Principle 6: Low Physical Effort

- (Frontend) Our app can be used with a minimum of physical effort. It can be efficiently used with a minimum of fatigue. There are very few repetitive actions, besides ones that inherently come with the use of a mobile device. The user has no cause to not sustain a neutral body position, and there are no demands for sustained physical effort. Our app is built such that a user may choose to take a break from usage without negatively impacting their experience with the application.
- One way we could improve in this area is by introducing a darker color theme. Our app currently uses white and pink, which can be harsh on the eyes. Using a darker theme with a less intense complementary color can reduce eyestrain and contribute to lower physical effort.

Principle 7: Size and Space for Approach and Use

- (Frontend) Our android app can be assessed by anyone who has an Android phone. Pages and images all accommodate variations in hand and grip size based on their own phone.
- We could improve our app's navigation a little, however. Right now, we have two forms of navigation: the top drawer that opens from the left, and our bottom navigation menu. As a user likely holds their phone from the bottom, it can be difficult to access this top menu. We could move the "open navigation drawer" button towards the bottom of the page to make our UI elements more easily reachable.

2. Write a paragraph about who you would market your program towards, if you were to sell or license your program to customers.

This program is marketable towards individuals who are pet owners that are interested in connecting with other pet owners nearby. This is not exclusive to owners of "traditional" pets such as cats and dogs, since any type of animal is welcome on the app. In addition, our app does not enforce any one specific goal such as having pet owners meet up in person to arrange playdates, so users who are only comfortable with connecting with others virtually could still find purpose in the app.

3. Write a paragraph about whether or not your program is less likely to be used by certain demographics.

Our program is less likely to be used by individuals who do not own any pets. Individuals who are not pet owners could still create an account, but they would not be able to use the main features of the program without registering any pets. Furthermore, individuals who do not wish to socialize with other pet owners would also be less likely to use our program. The primary features of our program involve matching one's pets with other pets in their community for the purpose of connecting pet owners. If an individual is not inclined to contact or interact with other pet owners, they would likely not be interested in using the features of our program.