CSC 207 Final Project - To The Moon
Aria, Andy, Juntae, Fatimeh, Praket, Jimin

Specification

A pixel-style dog petter game (Cookie Clicker clone) where the user can interact with a dog and play minigames to gain currency and experience. Every time you click on the dog, you get dogecoin. The dog you click on also gets experience, and rewards more coins the more experience it has.

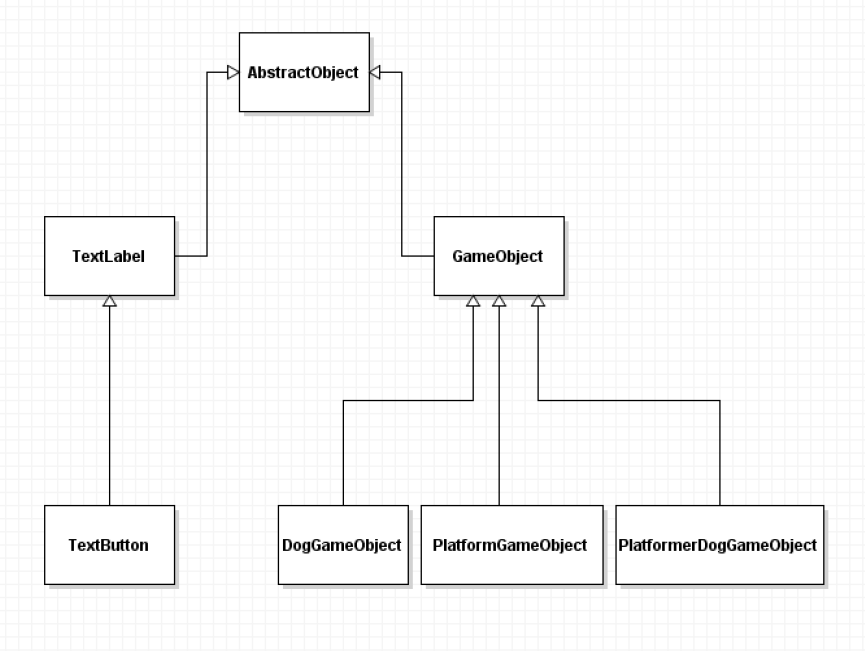
In phase 0, we only had a screen where you could pet a dog. In phase 1, we added a shop button to take you to a shop screen with buttons to buy dogecoin miners. These miners increase in cost per purchase, and they increase your "dogecoin per second." This means that every second, you get free dogecoin without doing anything! We also finished implementing the minigame, with a button on the home screen to play it. It's basically Doodle Jump (the dog automatically jumps, you press the left and right keys to move), and when you get to the top platform, you're awarded with a whole lot of dogecoin. If you fall off the screen, you get sent back to the main screen and receive nothing.

For phase 2, we generally expanded upon and refined features implemented in phase 1. For example, we added a background to the doodle jump minigame, and added the dinosaur minigame from Google Chrome as well. We created a minigame selection stage that allows the user to select the game they want to play. Other features include how the shop was updated to constantly change the price of each item based on real-life Dogecoin stock price data, the shop buttons say "Purchased!" after an item is bought, and the addition of in-game music and sound effects. Unfortunately, we didn't have time to implement hats.

Design Decisions

Before, we had a class called "DogManager." We found online that naming classes "manager" isn't very useful, so we decided to change this when we did a major refactoring at the beginning of phase 1.

We would have an AbstractObject class, which just represents some object drawn to the screen. There would then be a GameObject class which extends AbstractObject, and it represents all the objects that are physically there in the game's world. It has a sister class called TextLabel, which represents a text label. These don't actually exist in the game's world; an NPC shouldn't be able to physically interact or see a label. We also have a Clickable interface to represent a clickable object (excluded from the diagram for less clutter). TextButton extends TextLabel and implements Clickable, because text buttons are simply clickable text labels.



The idea behind making game objects was originally inspired by the Unity engine, where each object had a sprite associated with it, as well as a transform object which keeps track of its position and rotation. We decided to implement the same type of composition, which is why we have Movers, Transforms, and SpriteFacade classes. GameObjects that could be collided with would also implement the Collidable interface (because not every GameObject is necessarily collidable).

The Transform object is useful for not repeating code pertaining to x and y coordinates. We could just have that data (and methods related to it) stored in a class, which we could use in any object we wanted.

The idea behind making a Stage class also came from Unity. In Unity, you can switch between scenes, each with its own hierarchy of objects. Stage was obviously inspired by this, with each stage keeping track of its GameObjects and TextLabels. The Camera class was inspired by multiple game engines. Most game engines have a camera object, which basically represents what the player can see. This makes sense, because if the player moves left, you wouldn't move every other object to the right; you would simply move the camera left.

Clean Architecture

Entities

There are four entities in our program: Dog, which tracks the coins and experience of a single dog, Sprite, which holds the frames and attributes of a sprite, Transform, which represents coordinates, and GameState, which is a serializable class that stores the relevant values to be saved. These are all core "entities", and don't need to manipulate other objects. These could all be easily reused in another project, with minimal changes needed.

Use Cases

As for use case classes, they are all classes that directly manipulate entities, or are composed of other use cases. DogGameObject for example, is a composition of many entities and some other use case classes. SpriteAnimator is a class specifically for animating a Sprite object. The Stage class holds a bunch of GameObjects and TextLabels.

Bank was originally an entity, but we had doubts about what it really was. It had a PropertyChangeSupport object in it, and we weren't entirely sure if entities could have that. A lot of adaptors also use it, and we aren't sure if adaptors are allowed to directly use entities. The Bank also encapsulates a control flow, namely the processing of money. So we thought it made more sense to put it as a use case class.

Because we had so many use case classes, we decided to segment them further into sub-packages based on their role: dinominigame, interfaces, mainhub, object, and platformer minigame. More general use case classes, such as MinigameStageFactory and Sprite, were left in the parent usecases package.

Adapters

For major adapter classes, we have our DogGameController and Camera object. The controller processes user input, and manipulates use case classes accordingly. The Camera is our presenter; given the camera's position and its bounds, it "captures" what game objects to show on the JPanel. We also have two gateway classes: DogGameFrameLoader class, whose job is to load sprites from files, and MarketAPI, which is responsible for making the API call to get the Dogecoin stock price change. MarketAPI interacts with the Economy class, which is responsible for generating the random walk for the shop item prices. In addition, we have the GameReadWriter class, which is responsible for saving and reading serialized files for the game.

Frameworks and Drivers

For the outermost layer, we have our main DogGame class and the DogGameJPanel. The DogGameJPanel is our adaptation of Java's JPanel, so it counts as a framework. DogGame is the class with the main method, and "drives" the program, so we think it's appropriate to put it in this layer. It also uses the swing framework to create the game's window.

SOLID

Single Responsibility

We feel that for the most part, we were able to follow this principle. For example, Sprite is mostly responsible for storing and flipping a sprite. SpriteAnimator is responsible for animating a sprite. SpriteFacade uses the Facade design pattern, and delegates the task of storing and animating a sprite to the Sprite and SpriteAnimator classes. The way that GameObjects are composed out of different use cases also adheres to this principle, as each use case class is responsible for a single task.

The Dog class is responsible for storing and calculating the coins earned and experience earned of a dog. Sure, we could have delegated the calculations to their own classes - in fact we did initially! But those classes were way too short, as they only contained one short method! We decided to sacrifice the single responsibility in this case in favour of a cleaner and less cluttered code structure.

We notice that DogGameController was not only responsible for taking in user input, but it also creates the minigame stage. The DogGame class also created most of the stages.
In phase 2, we added a new Controller builder class using the builder design pattern that manages to resolve the problem that DogGame had with SRP because the task of building all of those components is now extracted from DogGame to the builder.
The minigame stages and other components such as the platforms are now created using a factory design pattern in order to take out the responsibility from the controller.

Open/Closed

We feel like the AbstractObject class hierarchy that we have is a decent way of applying this principle. If we wanted to make new game objects or text elements, we could simply extend TextLabel or GameObject, and maybe implement an interface or two. This is used especially well for gameobjects, where we have 4 different kinds (Dog, Platformer, PlatformerDog and Dino) depending on their use in the program and the specific way that they are used. The use of the stages also made it very easy to add new stages rather than re-write previous code.

Liskov Substitution

We made sure that children of classes didn't do anything completely different from their parent classes, and we made sure that children don't throw any errors not thrown by the parents. As a general rule, any parent class can be replaced with their children and still work, even if doing so may become more inefficient. With the help of the final TA meeting we managed to get rid of some possible Liskov violations that were present in our code.

Interface Segregation

We made sure that our game objects and text labels implemented all the methods from our custom interfaces (like Drawable and Clickable). Unfortunately, when adding the KeyListener and MouseListener to the DogGameJPanel, there were a lot of methods that we didn't need, so they had to be left blank.

Dependency Inversion

We made use of dependency inversion a lot, for example the ICamera, IFrameLoader, IGameController, and IGameGraphics interfaces were all made solely for dependency inversion. Without IGameController for example, how could a Mover know which keys are pressed? How would we switch stages when a TextButton is clicked on? These problems were easily fixed by passing in an implementation of IGameController to the classes who need it. We later expanded on these with more interfaces such as ISoundPlayer being used for dependency inversion.

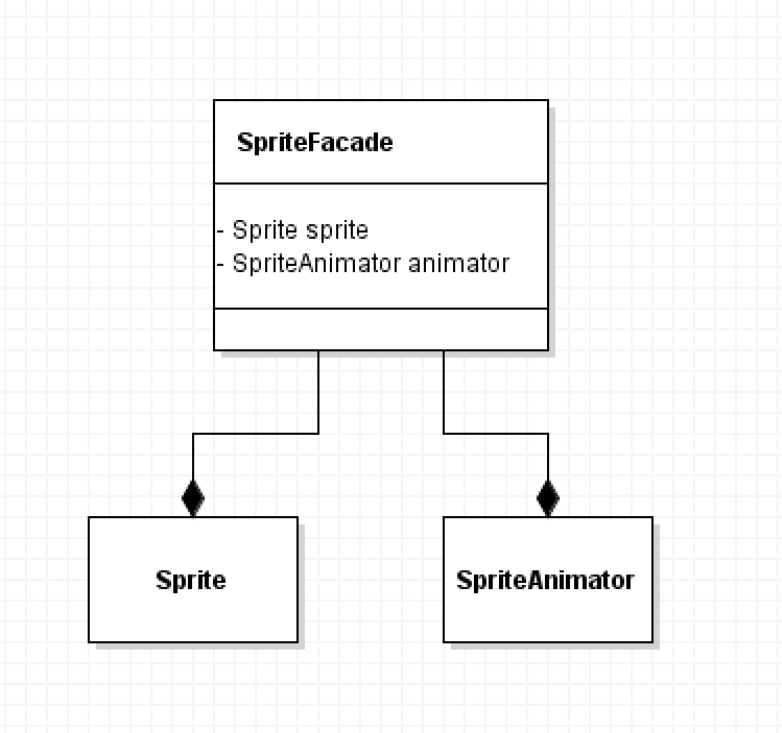
Packaging Strategy

The packaging strategy that we decided to use is the layers strategy. We have entities, use cases, adaptors, and frameworks and drivers as our packages. The reason we decided to do this is because the formatting is simple and the layout of the files makes it very easy to correctly implement Clean Architecture, which is vital. We had also considered packaging by component which is also recommended, but we ultimately rejected this because in that case there would

likely be too many packages and it would be more difficult to find violations in Clean Architecture.

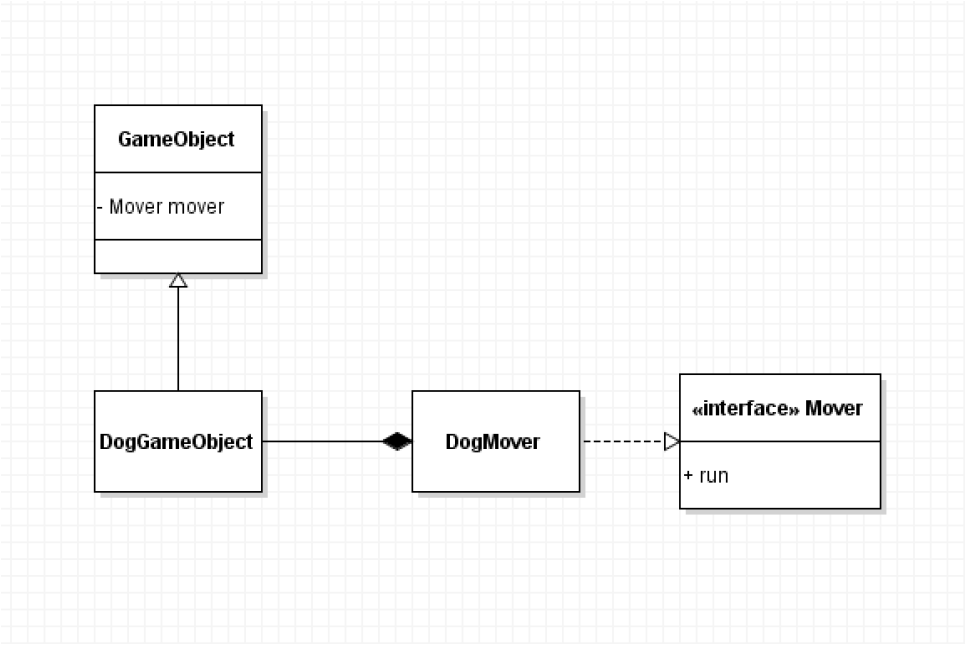
However, within the use case layer, we packaged classes by component, and tried to sort them based on how they work together. There are some minor issues, like a single class being used in two different places (for example, the PlatformGameObject is used mainly for the platformer minigame, but it is also used in the dinosaur minigame. We still packaged it in the platformer section, because that is its “main” use.) But overall, it still makes the use case layer more organized.

Design Patterns
Facade



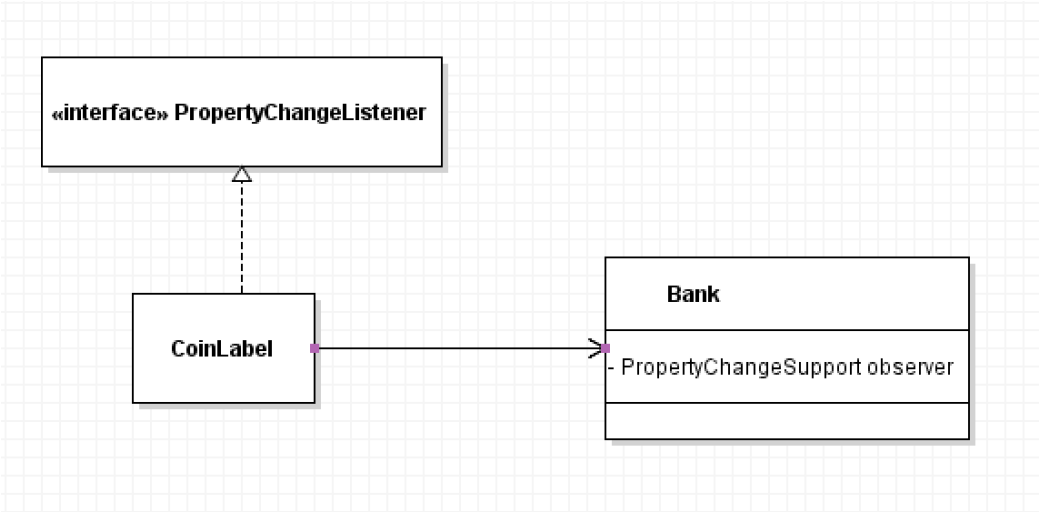
Before refactoring in Phase 1, the Sprite Class had more than one role, which was violating the Single Responsibility Principle. So we created SpriteFacade in order to delegate these roles to the Sprite class and SpriteAnimator classes.

Strategy



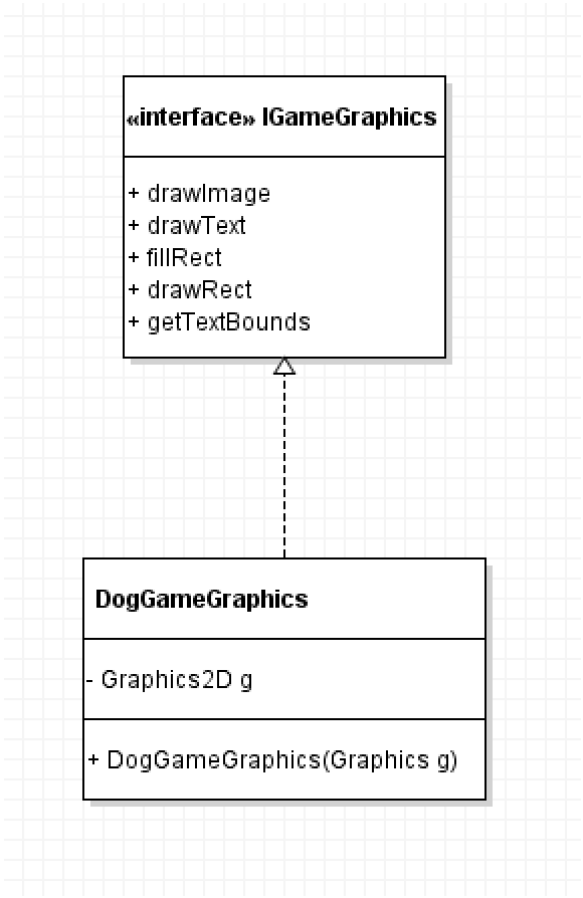
Before refactoring we only had a DogMover class which moves a dog randomly. However, in the future, we are going to have other game objects which need to move in a separate thread. Therefore, we are going to have multiple Mover classes that only differ in their behaviour, using the same algorithms, and we want to decouple the implementation of the class from the implementation of the algorithms. Mover is the interface that DogMover and PlatformerDogMover implement, and GameObject maintains a reference to a Mover object, but is independent of how the algorithm is implemented. By using Strategy design pattern, we could avoid nested classes and don't need to add other classes or modify the methods.

Observer



We needed a TextLabel that displays coins, and updates whenever the Bank is updated. So we created a subclass of TextLabel (CoinLabel) that implements PropertyChangeListener, and made the Bank have a PropertyChangeSupport object. The Bank has an addPropertyChangeListener method which uses dependency inversion to add an observer.

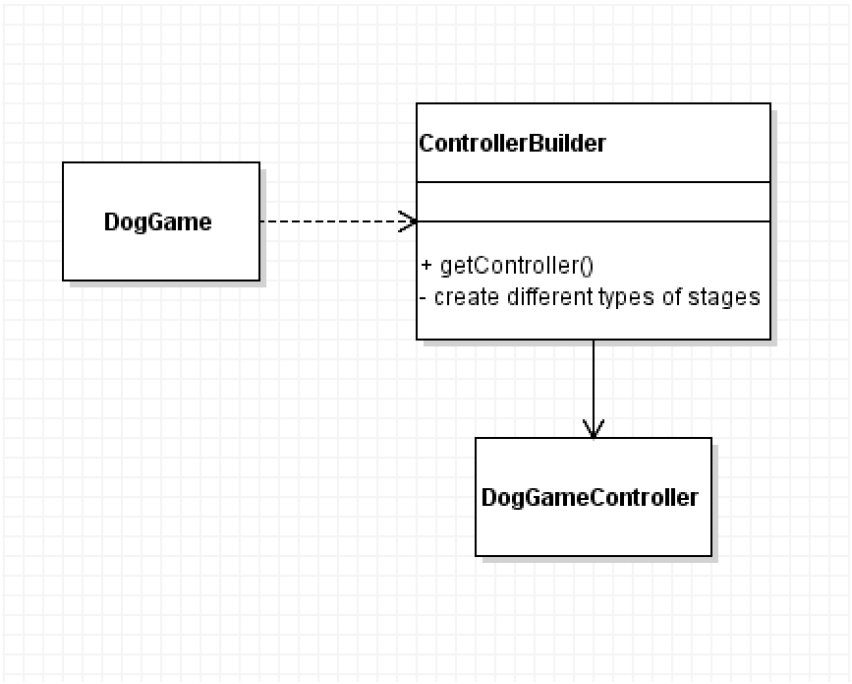
Adapter



We wanted to use dependency inversion with the Graphics object so that Drawable GameObjects could have a draw method (this makes drawing them all easier in the DogGameJPanel, as we would simply call the draw method. Hurray for polymorphism)!

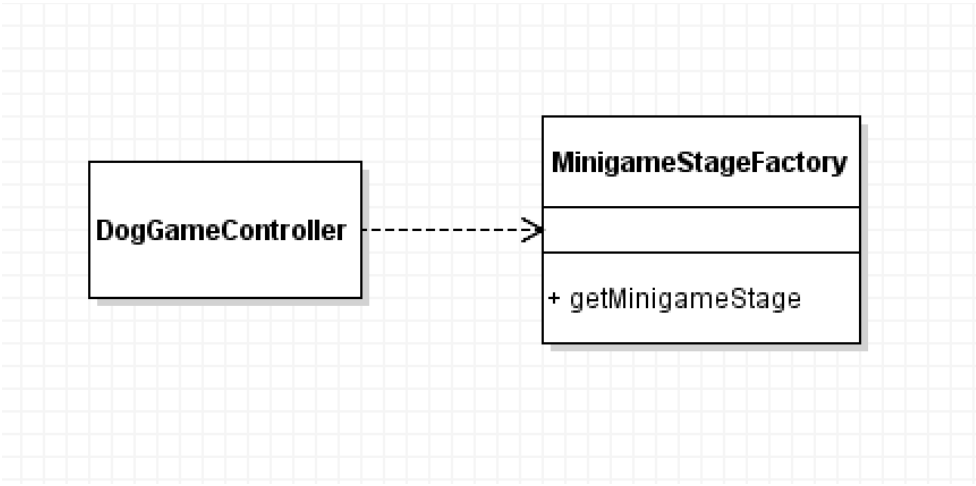
We went further and adapted the Graphics object using an IGameGraphics interface. This interface has methods that make drawing things a bit more streamlined, like including colour as a parameter, instead of having to call “setColor” every time we wanted to change it. It also includes a customized drawRect method that takes in stroke width. It even has a method to get a string’s pixel bounds, which makes it easy to place text in precise locations.

Builder



Originally, DogGame directly created the main stage, the shop stage, and other basic stages and added them to the controller. We thought that this was a slight violation of Clean Architecture, as something in the Framework layer shouldn't directly access Use Cases. So we used a Builder class (in the adaptor layer) to create the stages and the controller instead.

Simple Factory



Originally, DogGameController created all the minigame stages. To better adhere to the SRP, we delegated the creation of those to a Factory class instead.

Dependency Injection

There are many places where dependency injection is used. Namely in the controller class with all the "add" methods. Furthermore, dependency injection is used a lot for

dependency inversion, like how PlatformerDogMover needs the controller in order to see which keys are pressed, and the ShopButton needs the controller as well, in order to switch stages.

Progress Report

Group Member Info

Praket: For phase 1, I implemented the ShopButton class which represents a button on the main stage that can be pressed in order to go to the Shop stage, where you can purchase miner upgrades. The implementation of the shop button involved creating a new ICamera interface (to make stage switching work), and minor changes to the DogGame program driver. For Phase 2, I worked on implementing the changes to the design pattern suggested by the TA such as implementing the Builder design pattern, ControllerBuilder, which is called by DogGame in order to add the controller and create the different stages. The Factory design pattern, MinigameStageFactory, was made to create the minigame stages, and would be called by the DogGameController. I also worked with Jimin and Fatimeh to help write and edit the accessibility report.
https://github.com/CSC207-UofT/course-project-to-the-moon/pull/26
https://github.com/CSC207-UofT/course-project-to-the-moon/pull/29

Fatimeh: For phase 1, I implemented the MinerButton class which represents a Dogecoin miner button that can be purchased from the shop. I also created the buttons on the Shop stage to represent the three types of Dogecoin miners (Computer, Factory, and Lunar Dog Cafe). I implemented the HomeButton class as well which can be used in the shop to return to the main stage. I also contributed to the design document/progress report. In phase 2, I worked on improving the Shop stage by adding a timer to the MinerButton that says “Purchased!” when an item is bought. I also created the Minigame Selection stage which allows the user to select a minigame and added the minigames button to the main stage. Once the user clicks the button, it leads to the Minigame Selection stage where I added the Dino and Platformer game buttons. Since we added another minigame, I refactored all of the classes that used the term “Minigame” and changed it to “Platformer” instead to reflect the fact that we have two minigames. I also worked with Jimin to write the accessibility report, added to the design document and created the final presentation.
https://github.com/CSC207-UofT/course-project-to-the-moon/pull/22/files
https://github.com/CSC207-UofT/course-project-to-the-moon/pull/27/files

Andy: Code reviews, pointed out a few errors, and told the group about them on the Discord. Implemented sound effects and music. Fixed a few sprites.

My major contribution in Phase 2 was adding the sound. This is a major contribution, since it was a feature that we wanted to add on our specification.
https://github.com/CSC207-UofT/course-project-to-the-moon/commit/d8d96ac3f500e971cb2f59443fbc654a5c384707

One of my other major contributions from Phase 1 was adding the platforming logic for the platforming minigame. Minigames are a core part of our game, and that's what makes it major.
https://github.com/CSC207-UofT/course-project-to-the-moon/commit/a059915528878ee49133d1df0924e151698a0e46

Examples of refactoring that I did, which is a major contribution because it helps keep our code clean and organized.
https://github.com/CSC207-UofT/course-project-to-the-moon/commit/940ee1d11aa6ed823bbe4a09325fff55dcefa1ce
https://github.com/CSC207-UofT/course-project-to-the-moon/commit/62084e5a1233453b5864f2713c8af149bd6beb79
https://github.com/CSC207-UofT/course-project-to-the-moon/commit/44561a70470a8ecc7cda9c7a690de68d729c5a7e

Jimin: In phase1, I couldn't do many things like implementing codes because when I was available the previous parts weren't finished. I implemented test codes and refactored test codes from phase0. I wrote some parts of documents about design patterns. In phase 2, I wrote codes to add background to the platformer minigame and to add the moon image to the game when the user wins the game. Moreover, same as in phase1, I wrote test codes after our group finished writing codes. I also wrote the accessibility report with Fatimeh.
https://github.com/CSC207-UofT/course-project-to-the-moon/pull/20

Aria: In phase 1, I initially refactored the code from phase 0, improving the clean architecture and adding new parts to the program. Later, I wrote the rough draft of our design document, and then I created the minigame stage, and wrote the code to populate it with PlatformGameObjects and a PlatformDogGameObject, although it was later patched up by other teammates alongside the collision detection and the physics.

https://github.com/CSC207-UofT/course-project-to-the-moon/pull/15/files

I also made the button to switch the screen to the minigame stage, and I added the controller's ability to listen to what keys are being pressed. I also worked on writing the tests with Jimin.

For phase 2, my task was to create the dino minigame to add to our project alongside the doodle jump minigame that was previously implemented. Over the span of a couple of weeks I created three new classes in the "dinominigame" component (DinoButton, DinoDogMover and DinoDogGameObject) and wrote the dino game, which is based on the google dinosaur game where a dinosaur constantly moves right and the player can jump or duck to avoid platforms. I then proceeded to fix up the flaws that the game had, specifically the physics and graphics.
https://github.com/CSC207-UofT/course-project-to-the-moon/pull/21/files

Aside from this, I also helped in refactoring and pitched several refactoring ideas such as the controller builder with the help of the TA. I also helped with writing the design document.

Juntae:
A major contribution I made in Phase 2 was extending the save functionality. Previously, you would have to manually save the game otherwise the loadgame() method would throw an error. Now instead, at the start there is a stage with Load Game and New Game buttons, and if Load Game is clicked and there is no save file, it tells the user “No previous save file detected” and prompts them to make a new game instead of throwing an error. This is a major contribution as it was a feature we wanted to add in our specifications.
https://github.com/CSC207-UofT/course-project-to-the-moon/pull/25

Another major contribution was the addition of the Economy and MarketAPI classes, which is responsible for changing the price of the shop items based on historical dogecoin stock price data. In addition, when loading the game, if it has been a full day since the game was last opened, it automatically sends an api call to fetch the most recent day’s dogecoin data and updates the transition matrix of Economy. This is a major contribution as implementing an API call was another feature we had listed in our phase 2 specifications.
https://github.com/CSC207-UofT/course-project-to-the-moon/pull/8
https://github.com/CSC207-UofT/course-project-to-the-moon/actions/runs/1537898517

My contributions in phase 1 were:
Helped refactor code from phase 0. Worked on the collision & movement system for the platform minigame. Implemented the serialization process for the game. Implemented the observer design pattern for the CoinLabel class. In phase 2, will expand on the serialization classes to add options for the user to start a new game or load old one, work on implementing the API call for the in-game economy, and help with the platforming minigame.