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.

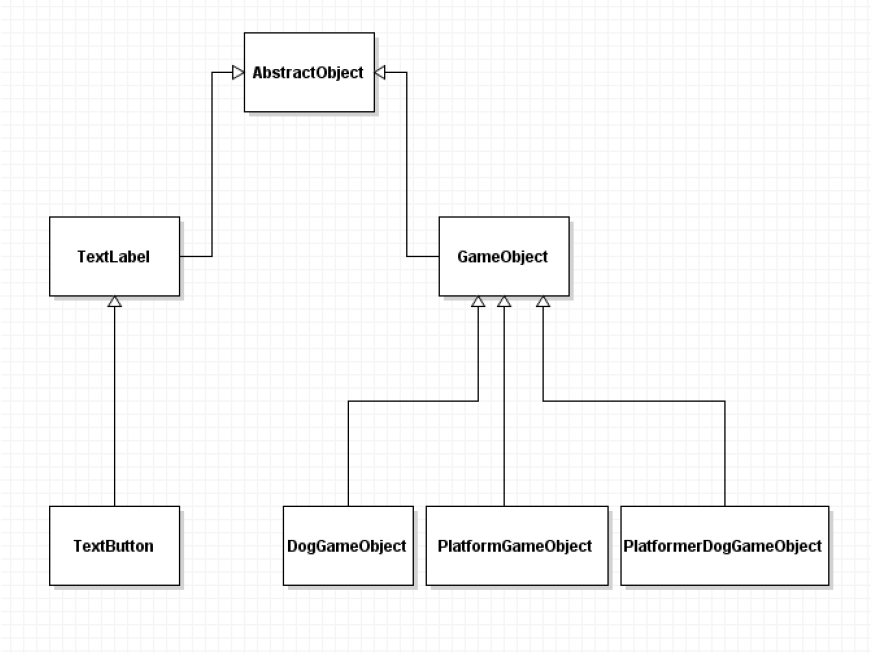
In phase 0, we only had a screen where you could pet a dog. In phase 1, we added a shop button to take you to a shop screen with buttons to buy dogecoin miners. These miners increase in cost per purchase, and they increase your "dogecoin per second." This means that every second, you get free dogecoin 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, in addition to expanding the domain of our projects, we plan on adding hats that you can buy for the dog. We also plan on adding music and sound effects.

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 three 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, and Transform, which represents coordinates. 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.

Adapters

For adapters, 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 a DogGameFrameLoader class, whose job is to load sprites from files, which we think counts as a gateway class. 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 is not only responsible for taking in user input, but it also creates the minigame stage. The DogGame class also creates most of the stages. We could very well delegate these to their own classes, but unfortunately we don't have enough time, so it will be a phase 2 task.

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.

Liskov Subsitution

Unfortunately, this one is difficult to follow in our case, especially with respect to GameObjects. Something like a PlatformGameObject cannot simply be substituted with GameObject (especially because it's abstract) and still work. TextButtons cannot simply be replaced with TextLabels, because then they can't be clicked on!

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

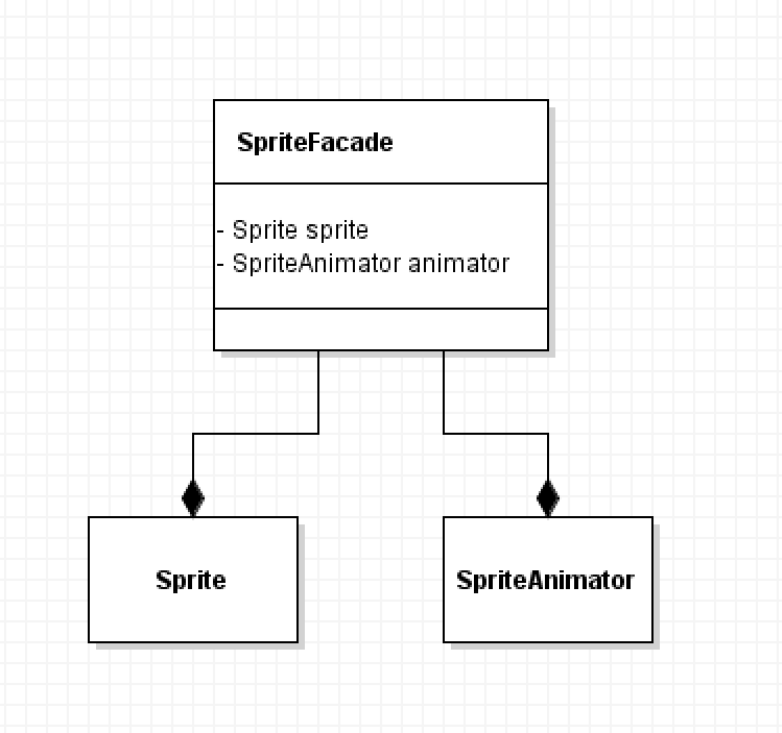
We made use of dependency inversion a lot, for example the ICamera, IFrameLoader, IGameController, and IGameGraphics interfaces were all made solely for dependency inversion. Without IGameController for example, how could a Mover know which keys are pressed? How

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.

Packaging Strategy

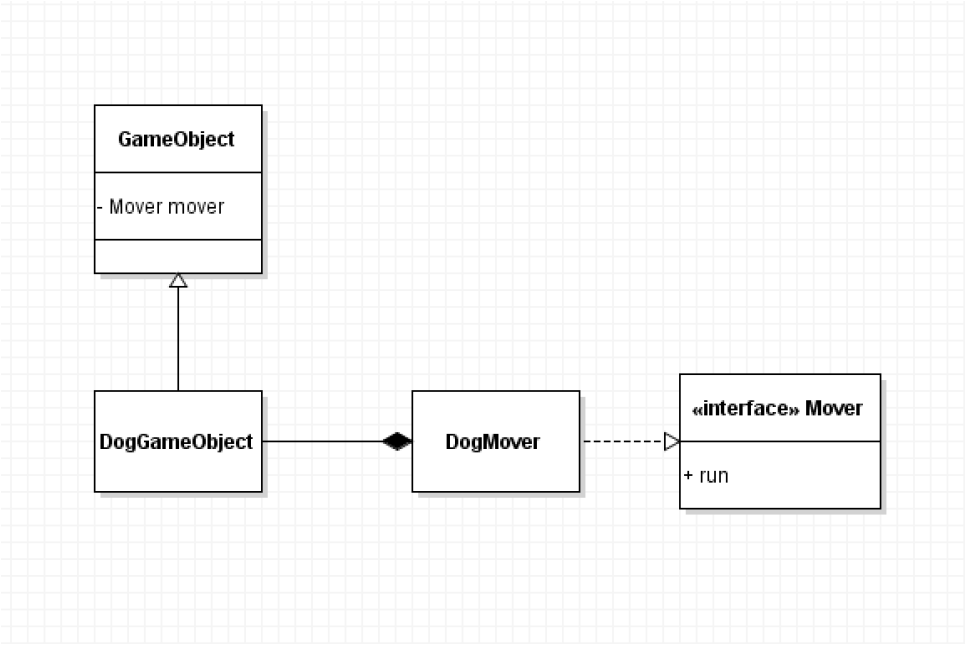
The packaging strategy that we decided to use is the layers strategy. We have entities, use cases, adaptors, and frameworks and drivers as our packages. The reason we decided to do this is because the formatting is simple and the layout of the files makes it very easy to correctly implement Clean Architecture, 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.

Design Patterns
Facade



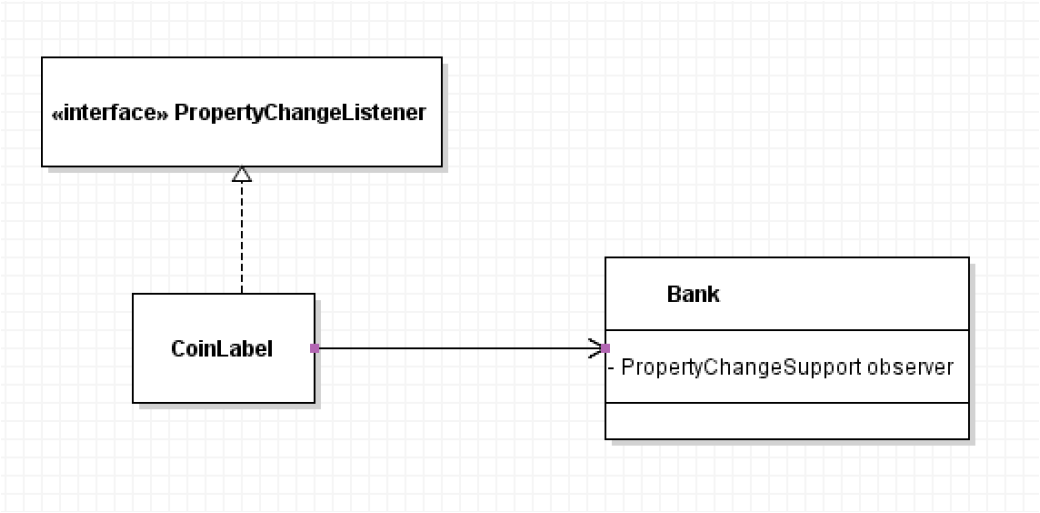
Before refactoring in Phase 1, the Sprite Class had more than one role, which was violating the Single Responsibility Principle. So we created SpriteFacade in order to delegate these roles to the Sprite class and SpriteAnimator classes.

Strategy



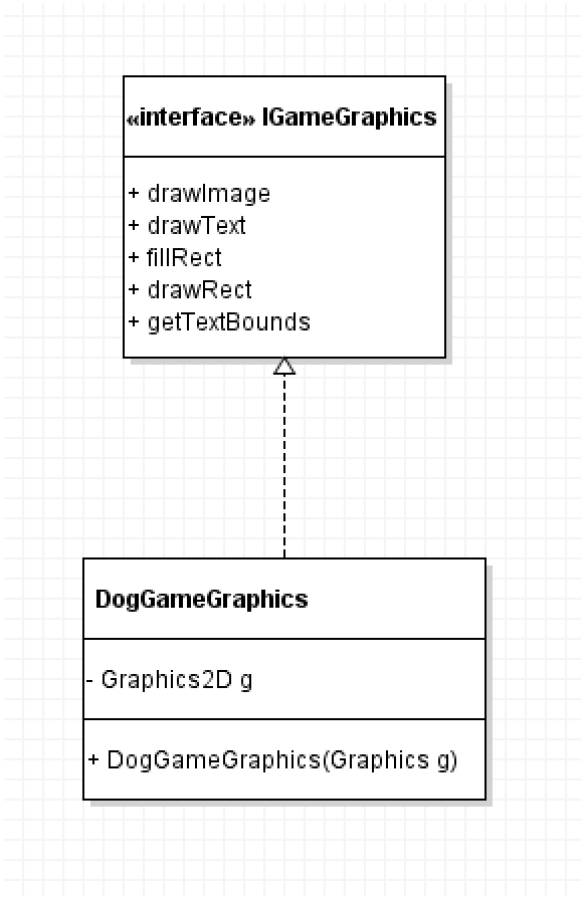
Before refactoring we only had a DogMover class which moves a dog randomly. However, in the future, we are going to have other game objects which need to move in a separate thread. Therefore, we are going to have multiple Mover classes that only differ in their behaviour, using the same algorithms, and 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.

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

Open Questions

How should we extend the domain of our project for phase 2? How exactly do we play sound effects? Should we use dependency injection again, but this time for a class that plays sounds? How many hats should we make, and what should the dimensions of each hat sprite be? How should we format save files in case we, as humans, want to edit them?

What has worked well

The AbstractObject hierarchy makes creating new game objects and text elements easier. The Stage class makes it really easy to switch between different screens on a single JPanel. The Camera class was really useful when implementing the minigame, because instead of having to loop through all of the platforms and moving each one downwards, we only had to move the camera upwards.

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 plan to contribute to adding the cosmetics to the shop as well as to any minigames that we may add in the future. Other than that I also plan on contributing to overall design improvements and cleanliness of the code.

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 (through the DogGame class) 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. For phase 2, I plan to continue adding to the shop in order to make it more interactive and visual when purchasing Dogecoin miners and aesthetic items. I also plan to contribute to any future minigames we make and improve the overall design of our code.

Andy: Lead the refactoring process at the beginning of phase 1. Made many UML diagrams to plan it out and posted them in our Discord server. Did many code reviews throughout the phase. Created PlatformerDogMover and wrote the platforming and movement logic, and finalized the Doodle Jump minigame. Formatted and revised this very design document. Will continue to plan things like hats, sound effects, and however we plan on expanding this for phase 2. Will also continue to do code reviews.

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 hope I

can contribute more on implementing codes and refactoring codes so that the project can follow SOLID principles and use proper design patterns.

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

Juntae:
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.