Design Rationale for Sprint 2

By Nhat Quang Nguyen - 3278 1954

Online version of diagrams:
https://app.diagrams.net/#G1k00PvTL5Z9fET2_ZSjfrBYiJ_aMCXtdd#%7B%22pageId%22%3A%22FzCXPY7aEGIFWH4fZPOv%22%7D

Introduction

This document details the rationale for Nhat Quang Nguyen's (32781954) design of the Fiery Dragon game for Sprint 2. Please refer to the linked UML class and sequence diagram as you are reading this document. Each sequence diagram represents how I plan to implement the game functionality listed in the specification.

For the tech-based implementation part of the assessment, the game functionalities I have implemented are :

[REQUIRED] (i) set up the initial game board (including randomised positioning for dragon cards);
[CHOSEN] (iii) movement of dragon tokens based on their current position as well as the last flipped dragon

Rationale for 2 key classes

My design started by modelling after the interactions/actions that occur during a game, the Fiery Dragon game can be broken down into these domains:

+ Moving players on the board
+ Selecting Chit Card
+ Changing turns
+ Rendering the game
+ Initialising the game (Not really part of game)

This design was inspired by the Model-View-Controller architecture from web/app development. However since the game doesn't involve any data persistence (yet), the design doesn't include the Model responsibility. The first three domains mentioned relate to the game logic, each will get a class with the responsibility of the Controller for the domain i.e Managing states and communication. Each domain will also get a class with the responsibility of a View (classes under the package UIComponents) i.e rendering to the screen based on the state of its respective Controller and also passing user input to said Controller, this takes care of the final domain being 'Rendering the game'..

Please see the 'Bridge pattern' section under Design Pattern rationale to see how Controllers and Views are structured in code.

The 2 key classes below were chosen as they give a representation of how the architecture stated above is implemented. As well as different approaches that were considered but scrapped, why they were considered and how they compared to the final design.

Key class 1: ChitCard

+ **What is it and what is it responsible for?**
  ChitCardManager is a class that has the responsibility of being a Controller for the domain ''Selecting Chit Card". It is responsible for handling state changes and communication to other classes when a user selects a Chit Card. Each Chit Card that is displayed has its own corresponding ChitCard object that contains its states and a ChitChadUI object to render to screen and detect user input. The ChitCard is abstract and employs the Template Method design pattern, subclasses are expected to implement the handleSelection() method to allow for custom behaviour when the user selects a Chit Card.
  ChitCard has an associated ITurnManger and Board to get information about turns and players on the board in order to inform it's behaviour when flipped. ChitCard is instantiated and managed by

ChitCardManager, which is a class that is a collection of ChitCard and also manages turn (by implementing ITurnManger)

+ **Why can't it be implemented as a method or as a concrete class?**

Approaches	Method of ChitCardManager	Concrete Class	Abstract Class
Advantage	Instead of having individual objects for each ChitCard, chit card selection could've been implemented as a grid of buttons that have chit type associated with them, and the input is handled by switch cases in a method. This design is the least complex and easy to understand	Why can't we just implement the chit card selection handling logic? The logic is fairly simple and would remove 2 subclasses from the design. Making it more simple	Best encapsulation. Provides the easiest way to add new functionality to the game in Sprint 4. Existing subclasses show examples of how to extend new behaviour to the game. Meaning it is easier to understand just by reading the code => easier for teammates to pick up
Disadvantage	This provides worse encapsulation than other approaches and will make the code both harder to reuse and extend. This also violates the SRP* of ChitCardManager, which will need to take care of both handling chit card selection and turn management. This would make the ChitCardManager class big and harder to read and understand.	Abstract class still provides better encapsulation. For example, if we want to change just the behaviour of the Pirate Dragon chit card, we can't reuse any common behaviour to other Chit Card types, due to the Open-Close Principle.	More complexity and classes in the design, but with the use of the Template Method design pattern, it won't be too hard to understand.

*Single Responsibility Principle

Justification for final decision: I decided to use Abstract Class out of all the alternatives because I believe that the flexibility for extension in the future (Sprint 4) outweighs the extra complexity that it brings. The extra complexity is also rather easy to understand as it is just a use of the Template Method design pattern, which other teammates should have a knowledge of.

Key class 2: Board

+ **What is it and what is it responsible for?**

Board is an abstract class that has the responsibility of a Controller for the domain 'Moving the Player around the volcanoes and caves'. This includes managing states relating to the board (types and position of Volcano Card and Cave Card), player positions, moving player and ending the game. To support other game functionality, the Board allows access to its state through getters, but only allows other classes to change its state by moving the player. The consumer specifies how many moves it wants to make and for what player, and the Board would ensure that the movement follows the rule of the game.

+ **Why can't it be implemented as a method?**

Approaches	**No class**	**Create a class**
Advantages	If this handling logic relating to Board isn't a separate class, a different approach is to implement these responsibilities as methods and attributes inside the rendering class for Board (BoardUI), meaning that BoardUI can readily render the board from its own state without needing to communicate with another object. This is very straightforward and easy to understand, it	Decouples the game logic from the rendering logic. => We can change the rendering implementation in the future without even touching the Board logic code. For example, if we want to use different assets, or render the board as a circle instead of a rectangle, etc.

	also removes a class from the design, making the diagram less complex. And it wouldn't violate the SRP as much as Key Class 1, since all of the responsibility still falls under Board.	More realistically, any major change needed from updating the game engine will only affect the rendering class
Disadvantages	However this would couple the logic of the Board to the implementation of how it is rendered. This is very bad for encapsulation and both the rendering code and the board logic code are not reusable separate from each other. This would pose a difficulty when we need to refactor to add functionalities implemented by other teammates in Sprint 3. Since the rendering logic might also be very complex, adding this responsibility on top of it would make the Board class very large and hard to read.	A separate class is needed for the Board logic => More classes in the diagram. Developers will need to learn and understand the communication between the Board and BoardUI.

Justification for final decision: Like for the first key class, the encapsulation and maintainability offered by creating a separate class significantly outweighs the very minor disadvantages here. Another question might be why didn't I create an abstract class for Board as well, my reason for this is that the methods defined in Board depend on implementation of the Board class significantly, meaning that if a developer wants to create an alternative implementation of Board, it is unlikely any of the code is reusable.

Rationale for 2 key relationships

Key relationship 1: Aggregation relationship of Board to ChitCard

+ What is this relationship for?

  ChitCard needs to hold an instance of Board so that it can perform updates on it when handling users selecting a chit card.

+ Why is it not a composition?

  ChitCard and Board are responsible for 2 different domains that are both required for the game to function as required, so having their life cycle independent of each other is most ideal here. Furthermore, a composition wouldn't make any sense here, since that means every single ChitCard has a different Board instance (due to dependent lifecycle), and we're supposed to only have one board, for the entire game.

Key relationship 2: Composition relationship of ChitCard to ChitCardManager

+ What is this relationship for?

  The ChitCardManager holds a collection of all ChitCard that have been instantiated in the game, and it needs this to aid rendering and perform actions that affect all cards (ending a turn causes all cards to be hidden). The lifecycle of ChitCard depends on the ChitCardManger, as ChitCardManager instantiates them.

+ Why is it not an aggregation?

  A valid approach that uses an aggregation relationship between these two classes would be to instantiate them independently from ChitCardManager and pass them to ChitCardManager via Dependency Inversion. But this doesn't make a lot of sense since ChitCard objects aren't used for any other game logic outside of the ChitCardManager. And since information on turns are provided by ChitCardManager, ChitCard can't really operate and exist independently from them, meaning that they should be removed if the ChitCardManager holding them is removed. This is perfect for composition. And the same logic applies for the composition relationship between Cave/Player class and Board.

Rationale for 2 set of cardinalities

Cardinality 1: ChitCardManager to ChitCard

This cardinality depends on the implementation of the generateChitCard() method, technically this relationship could be 1 to 0..* as there isn't any hard limit on how many Chit Cards you can have. But

the game rules stated that there are 16 Chit Cards, and all Chit Cards is instantiated and managed by one ChitCardManager (there is only one ChitCardManager for the whole game, since it manages EVERY chit cards), the best relationship to match requirements would be 1 to 16, but this is flexible to change in Sprint 4. The same reasoning can be applied to why the relationship of Board to Cave or Players is 1 to 2..4.

Cardinality 2: Between Board and BoardUI

Board is a Controller class and BoardUI is a View class, only the 1 to 1 relationship makes sense here since a View can't have none nor multiple Controllers and vice versa. You can't render one thing from multiple sets of states, and there is only one screen to render to.

Rationale for inheritance

Inheritance from the abstract class ChitCard

- The decision to make ChitCard an abstract class instead of a concrete class was discussed in Key Class 1's rationale. In short, this was to make use of the Template Method design pattern, which offers good encapsulation and extendability for future game extensions.

Inheritance to Game Engine (libGDX) classes

- Inheritance to the Actor class: Required by the libGDX library for rendering and input handling functionalities
- Inheritance to the GameScreen class: Required by the libGDX library for implementing different Screen (e.g Main Menu screen vs The actual game)
- Inheritance to the Game class: Required by the libGDX for the driver class, needed for launching the game using libGDX.

Rationale for Design Patterns

Template method - As mentioned multiple times before, the Template method is used here to allow maximum flexibility when defining behaviour for a chit card, this helps with extending the game in Sprint 4.

Bridge - This is the core principle behind the View and Controller architecture, the game logic (Controller's responsibility) is decoupled from the rendering logic (View's responsibility). A View class contains an abstraction of the Controller class that it renders, and is provided with the implementation (the instance) at runtime via Dependency Inversion. This can be seen the clearest for the ChitCardUI and ChitCard class. The other View class is associated with the concrete class of their Controller, initially I also had them associated with an abstract class instead, but I found adding an abstract class didn't have other uses except for this, and using concrete class doesn't impair this design pattern. For example, BoardUI has a Board attribute as its Controller, any subclass of Board would work here as well, but if Board inherits from an abstract class instead, that abstract class doesn't have much use, see Key Class 2 rationale.

Other design patterns that was considered but didn't end up in final design:
Factory - I had an idea of configuring and initialising the Chit Cards using information from an external file, where the Factory method will instantiate a subclass of ChitCard that corresponds to a name that was read from the external file. This approach is sound but it requires adding extra complexity to the design compared to what we have now, for something that is not a requirement.

Builder/Strategy - Due to our use of the bridge pattern, the game logic is decoupled from the rendering logic, if we were to create an abstraction for the classes under the UIComponents package, we could've used the Strategy pattern to allow for different algorithms to be use to render the game. We could have the player choose to configure the game in the main menu, and use the Builder pattern to piece together different rendering strategies to create a game instance. For example, if the player wants a circular board instead of a rectangular one, the Builder could instantiate a FieryDragonGameScreen that uses a subclass of BoardUI that has an algorithm that renders Board into a circle instead. However, this adds a lot of complexity for something that is not a requirement, but it could still offer some good modularity. If a teammate uses a different rendering algorithm for their implementation, this pattern can help us adapt that to our implementation.

Singleton - A lot of classes makes sense to be a singleton like Board, ChitCardManager or any of the UIComponents since we only need one of each for the entire lifecycle of the game. But, aside from making intuitive sense, using a Singleton pattern doesn't really solve any problem here. There isn't any global variable or

any external process that needs an interface here. Using singleton could bring the risk of creating god classes and bad practices