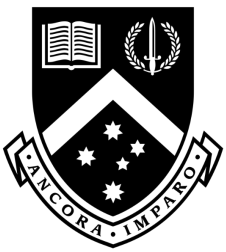


MONASH University

FIT3077 Sprint 3 (20%)

Team: CL_Monday06pm_Team377

Table of content

1. Design Metrics

In this implementation review, we uses several metrics to aid in the evaluation of the designs:

- Coupling Between Objects (CBO): for a particular class, we count how many other classes are involved in the implementation of that class.
- Reused Ratio (RR): the number of classes that are used by other classes over the number of total classes.
- Number of children (NOC): for a particular class, we count how many children that class has.
- Depth of Inheritance Tree (DIP): for a particular class, we count how many parent classes there are, up to the base class.

All of the above metrics are applied only to the classes in the design and the classes provided by the LibGDX library are disregarded. For CBO, we only consider direct coupling between classes and interface is involved in the calculation. As Prof. Jean-Guy mentioned, the marks of classes in each metric does not conform to a normal distribution. Hence, we decided to keep track of the highest and lowest mark of each design only.

Design	**Pham Nguyen**	**Nhat Nguyen**	**Duy**	**Khang**
Highest CBO	**7**	**4**	**6**	**4**
Lowest CBO	**2**	**1**	**1**	**0**
RR	**10/11**	**13/14**	**11/12**	**14/15**
Highest NOC	**0**	**2**	**4**	**3**
Lowest NOC	**0**	**0**	**0**	**0**
Highest DIT	**0**	**1**	**1**	**1**
Lowest DIT	**0**	**0**	**0**	**0**

Table 1: Design evaluation metrics

2. Design review

Criterion	Measurement Scale	Criterion	Measurement Scale
Completeness of the solution (Functional completeness, and functional correctness)	1 - Solution is missing all key functionalities. Fatal errors thrown frequently	**Rationale behind the chosen solution (Functional appropriateness)**	1 - Missing or inadequate rationale, doesn't answer why design decision was picked
	2 - One or two major omissions. Bugs are frequently encountered during gameplay		2 - Rationale has been stated but not appropriate
	3 - Many minor omissions. Many bugs occur in edge cases.		3 - Adequate rationale but some functions are not appropriate or missing
	4 - One or two minor omissions only. Minimal number of bugs in key functionalities		4 - Strong rationale with most parts in alignment with the requirements
	5 - No omissions of functionality detected. Not bugs detected		5 - Clear and compelling rationale, all parts align with the requirements
Understandability of the solution direction (Appropriate-ness recognizability)	1 - Highly confusing and unclear direction. Solution cannot be understood by both author and others	**Extensibility of the solution direction (Modifiability)**	1 - Solution is hardcoded and/or cannot be extended
	2 - Confusing and unclear direction. Solution can barely be underwood by author but not others		2 - Limited extensibility, the game is quite fixed and cannot be extended much
	3 - Adequately understandable direction. Solution takes considerable time to understand for both author and others		3 - Can be extended but takes huge effort
	4 - Clear and understandable direction. Takes relatively short time to understand.		4 - Highly extensible, most parts could be made use for developing for Sprint 3
	5 - Highly clear and understandable direction. Can be understood quickly, even without assistance from author		5 - Extremely extensible, every parts of the game are design for dynamically purpose which could be easily extended or reused
Quality of the written source code	1 - Poor coding standards, multiple instances of hard-coding. Not understandable	**Aesthetics of the user interface**	1 - Unpleasant and disengaging, hard to understand what the interface is trying to convey

(Maintainability)	2 - Below average coding standards, some down-casts. Code is hard to understand and strong coupling between multiple classes	**(User engagement)**	2 - Below average aesthetics, somewhat disengaging. Bad user experience with many interactions not reflecting the game
	3 - Average coding standards, clean code but not efficient. Strong couplings between classes		3 - Average aesthetics, understandable and usable but disengaging and boring
	4 - Good coding standards, minimal down-casts. Understandable with minimal coupling is involved		4 - Good aesthetics, good looking. Reflects the physical game well
	5 - Excellent coding standards, understandable design/code and no violations of clean coding practices.		5 - Excellent aesthetics, highly engaging. Make full use of the medium to make the game even more enjoyable than the physical game

Table 2: Measuring the quality of designs from 1 to 5

2.1. Tran Ngoc Duy Ngo's design - flip chit cards

Functional completeness: 5 - His program completes as the chosen functionality was to flip the chit card and at the end, the game was set up correctly with the position of the caves, volcano cards, chit cards inside and the chit cards were able to be flipped up and down without any errors.

Functional correctness: 5 - The program aligned correctly with the specifications where 16 chit cards were displayed inside the volcano cards, able to be flipped, and after each turn, the position of each chit card remained the same which aligns with the game rule. In addition, every time the game runs, all components in the game like Chit Cards, Caves, Volcano Cards, Player Pieces were displayed at their same correct positions, and 16 chit cards were scrambled randomly which maintains the game fairness.

Functional appropriateness: 3 - We cannot assess much about the functional appropriateness of this design as this system currently only covers the first two key functionalities of the game, which means he did not specify much about the direction that he wants to develop his system. So far, the game has the first two key functionalities working correctly, and the way he built some other functions for later Sprints are good but not specific enough; hence we gave it a 3.

Appropriateness recognizability: 3 - The code seems to follow a concrete approach as its implementation aligns with the template-method pattern and the abstract factory pattern. However the later parts of the game have not been much clarify in the design such as how the game can be ended and which class to maintain the movement of players.

Modifiability: 4 - This system's design had some parts that can be extended in later Sprints such as the fact that he created the 'GameElements' abstract class which could be extended in the future if there are more components in the game, and he made an abstract class 'State' which could be reused when there are more states in the game such as winning states or states where more than 4 players can play. In addition, regarding the metric RR, his system has the ratio of 11 out of 12 as he had efficiently reused all his components except for the mainGame class rather than duplicated which indicates that his code is easy to maintain and extend. Other than that, the game does not implement any part related to the number of game components and game board that can be taken advantage of for later modification when there are more than four players which may not be a good idea if we want to extend the game to be played by multiple players.

Maintainability: 3 - The system takes advantage of the Abstract Factory pattern then it makes the code when initialising the position of each component less "hard code" and easier to maintain through an Interface called "GameElementsFactory" which seems to contain all the functions to create those components. Besides, the code also has an abstract class called GameElements that takes all the common features of components in the game that make it easier to be tracked, maintained, and changed. However, there are some parts that he still hard-coded, and a few functions were not used with full capacity (like it has not been extended for later sprints). And in regard to the highest CBO metric which is 6, this not only increases the complexity of the system, but also makes the maintenance more difficult as changes in one class can have a ripple effect on many other classes due to high coupling. This makes his codebase more fragile and impairs reusability as classes rely on specific implementations and other classes.

User engagement: 4 - The images of the Caves, Volcano Cards, Player Pieces look great, correspond to each other, and can be distinguished easily by players. However, the UI of Chit Cards when turned upside down were a bit ugly. Besides, the game board tends to be oval shaped but it is rectangular now.

2.2. Khang Huynh Nguyen's design - flip chit cards

Functional completeness: 4 - The implementation succeeds in delivering game components such as Pieces, Volcano Cards, and Chit Cards into their correct position and forming an initial stage of the game. In addition, the system also randomised the chit's position and made sure all the cards were flipped after clicking and closed whenever they were not used.

Functional correctness: 4 - All components are fully displayed in their correct positions whenever the game runs. His game enabled the chit cards to be randomised every time we restart the game, which aligns with the game requirements. Not only that, these cards can be flipped without glitches as the second functionality, and only one card can be flipped at a time.

Functional appropriateness: 2 - We cannot assess much about the appropriateness of this design as this system only covers the first two critical functionalities of the game and a bit of expansion of the fifth functionality. This means that the game will require various modification processes and approaches to complete the basic game structure.

Appropriateness recognizability: 3 - The code gives a straightforward, visible approach to the initial board setting up since it follows the factory method pattern and the flipping of the chit card. However, we cannot find any parts of the implementation that can promote the way for other functionalities such as the ‘piece’s movements’ or ‘turn taking’ or the ‘end of game’ functionalities. Hence, we gave his design a 3 for the excellent and clear approach so far but not a good foundation for some subsequent development of the game.

Modifiability: 4 - This design approaches a good expansion since there is an abstract for creating components and expanding the game state. As the implementation, the components of any game stage can be created using the abstract class “ComponentsStateCreator” class. The code also aims to expand various game stages, making it easy to add additional stages such as menu stage, restart stage, analysis section, etc. However, the code was hard-coded inside the “PlayStateCreator” class and was not ideal to maintain. From the metric table above, the implementation has a high reuse ratio; other classes use 14 classes, which total 15 classes. It can be seen that this code can be easily modified and extended. Moreover, the highest number of children that each parent has is 3; for example, the game states such as “WinningState”, “PlayState”, “StartState”, and their parent class - “State‘. It also proves that this code has a reasonable foundation for the sprint four extension.

Maintainability: 2 - The code of this game is relatively clean as he implemented the Interface called “ComponentsDisplay” to use typical attributes of game Components such as Chit Cards, Caves, and Volcano Cards, which increase the code maintainability and extensibility. However, some parts seem to be “harem, " such as when it is in, initialising the components' position in the game. The system did not take advantage of any design patterns at this point to make any Interface or Abstract class to maintain the creation of game components so that if there were any problems with the position of any components, it would be hard to navigate and fix the bug. So, one way to improve this structure would be to implement a separate abstract class to keep the component's creation functions and increase code maintainability and readability. Furthermore, this design's Coupling Between Objects (CBO) is at the bottom of the list. This means that this code is not complicated and can be maintained independently compared to others. Not only that, but the depth of the inheritance (DIT) is extraordinarily reasonable, which is the depth of one. This makes the code more straightforward to keep track of and understand.

User engagement: 4 - The board design doesn’t follow the original game since the board is not in a circle shape. In addition, the board game card doesn’t have many in-game instructions. Overall, all components are of a reasonable size and neat and visible. Also, the game components and the backgrounds are colourful and contrast, making it easy to differentiate. Moreover, the game is divided into various stages, such as the start state and the in-game state, so that the player knows what is happening in the game.

2.3. Pham Nhat Quang’s design - Change of turn

Functional completeness: 5 - The system initially allocates all game components such as pieces, volcano cards, chit cards into their correct positions and forms an initial stage of the game. In addition, each player can take turns each time to flip the chit card and the system clarifies each player’s turn. The system is complete without any errors when running which deserves full marks for this aspect.

Functional correctness: 5 - After the game starts, all game components are fully displayed and in the correct positions which creates a shape of the board game. Following that, there is an arrow to demonstrate the player whose turn to flip the card. This arrow will move on to the next player if the current player chooses the card's type that does not match the board's type which aligns with the game rule and is completed for this aspect. He also correctly implemented the position of the game board, number of game components, which will be given a 5.

Functional appropriateness: *4* - The game builds up a good base of the game for Sprint 3 collaboration. We can assess the functional appropriateness of this design although the system covers the first and the fourth key functionalities of the game. Furthermore, the code base has been built as a good foundation for other functionality such as flipping chit cards and moving pieces. This means that this design will become an option for us in sprint 3. Hence we gave it a 4.

Appropriateness recognizability: 4 - The code seems to have the right and clear direction for later sprints. About his specific functionalities, when he wants to change the player's turn, he just needs to call the get function to the playerList then that list will send back the position for the current player. We think that is an efficient way of doing this. In addition, his design took advantage of the Builder pattern which emphasised that every component that he wants to design will be maintained by that concrete Builder class.

Modifiability: 4 - His code overall is highly-extensible for several reasons. His design follows the Builder design pattern that allows him to create a separate Builder interface to help build game components more dynamically, if we want to increase the board size or the number of players, caves we can simply modify from that pattern without having to rebuild everything from scratch. Besides, his boardSlot is also really useful and highly reusable as it can help navigating every position in the game board, and apparently it could be used when implementing other functionalities or extensions. However, his boardSlot seems to do a bit many things, which is close to the God Class, so we decided to give his design 4 marks for this aspect.

Maintainability: 5 - The code was written in a good way without any parts that he hard-coded and he followed the builder and the Chain of Responsibility pattern where he allowed all components in the Board to be connected together. He could also update the movement and image every time the player made a move which is really efficient and he did make the code really clean without any redundant functions. Hence we gave him a 5 for this part.

User engagement: 4 - His game has a nice UI, of the Caves, Volcano Cards, Chit Cards, which is attractive for users. However, the outlook of the chit cards look real but those are a bit hard for users to distinguish. This is not a big problem so we decided to give him a 4 as the rest of the game looks good enough.

2.4. Nhat Quang Nguyen's design - movement of player around the board

Functional completeness: *5* - All the required functionalities are covered in the prototype. No bugs or defects are identified that critically affect the operation of the prototype. No game components are missing on the screen with enough numbers, including the chit cards, the game board and the players. The player can click on the chit cards to move their player piece. The prototype supports the needed interaction for the functionalities.

Functional correctness: *5* - The covered functionalities are covered to a correct and sufficient extent. The game board is rendered in a correct manner. The chit cards are rendered inside the game board and the initial location of the players is correct. In addition, the movement of the players matches the number of steps displayed on the chit cards, and the player is able to continue their turn when the flipped card matches the slot, and cannot vice versa.

Functional appropriateness: *5* - The functions of each class corresponds to their physical counterparts operation, and there is a cooperation between the classes to fulfil the requirements. In addition, there are no redundant methods or functionalities of any classes, every class or method involved has unique and irreplaceable logic.

Appropriateness recognizability: *5* - Regarding this quality, the prototype has the game elements printed on the screen clearly and intuitively. This helps players to easily recognize these components. Moreover, the interactions between the game and player are simple and straightforward. This prevents any confusion for the players that discourages them from playing the game.

Modifiability: *5* - The classes are implemented so that the system can be extended, especially for the next sprint with techniques akin to the Strategy pattern. For example, the Board class can be inherited so that more configurations of the board can be added. In addition, the ChitCard base class can be inherited in the future to have more functionalities and special effects. We could add builder classes in the future to construct the classes more effectively. The UI classes are separated from the game logic, which is beneficial to accept change in the future. These classes can be modified to support the change of visual aesthetics, i.e when we want to change the way the Board or Chit cards are shown. Besides, in regards to his really high RR points, he successfully managed to reuse many of his classes. This not only improves the consistency, reliability of the system but also suggests that his interfaces are well-defined so that those are more likely to be reused and integrated into different contexts. In general, the system is highly extensible and we gave him a 5 for this.

Maintainability: *5* - In general, the system includes loose-coupled classes, which means it is easy to be modified without modifying other classes. In addition, the hard coded logic is located centrally in some specific classes, mostly to construct the objects. Hence, these hard coded logic can be easily located and modified in the future. As described above, the UI classes are separated from the main logic of the game. Hence, if a bug is raised, we can locate where the bug is by testing the main logic classes and the UI classes. As the UI

classes are supported by LibGDX, the UI classes will not likely be the source of the bug. However, they still can be a source of defects and having this separation makes it easier to debug. The high maintainability is somewhat reflected in the low overall CBO metric scores - 4. This indicates that classes in his design have fewer dependencies on other classes which can promote simplicity and are easier for testing and fixing bugs.In general, we gave him full marks for this aspect because of the easy to understand implementation that also intuitively shows us that there is loose coupling between key classes.

In addition, as we referred to his CBO - 4 which is relatively low. This indicates that classes in his design has fewer dependencies on other classes which can promote simplicity and easier for testing and fixing bugs

User engagement: *3* - In terms of visual aesthetics, the game does not look attractive. The animals are replaced by squares or circles, which reduces the attractiveness of the game immensely. Realistic images for the animals should be generated and added to the game. It would make the game more visually appealing which encourages the player to play the game more.

2.5. Final design decision

From the rating above, we can see that the fourth design - Nhat Quang Nguyen's design is rated the highest scores among designs. Our final conclusion was the fourth design due to its structure and approach to fulfil the requirement. The design demonstrates setting board functionality and the movement of the pieces on the board, but it also was implemented with collaboration and consolidation of functionality in mind for Sprint 3 as his functionality was movement; hence he has already included the turn-changing functionality was half-implemented, ChitCard already implemented inside the ChitCardManager, Board already has checks for win checking, etc). As a result, the implementation built up a good foundation for the flipping chit card, turn taking, and winning the game. Additionally, all team members agreed that this design is the easiest to learn and understand, and as Nhat also presented in detail his plan for implementing the remaining functionalities, extending from his base design will also have been the fastest and easiest.
While choosing the final design, we abandoned Nguyen's design and Ngo's since those designs are just for Sprint 2 and there are not many classes that can be taken advantage of for the next Sprint. Moreover, the implementation of these 2 designs did not show a possible maintainability due to hardcoding while setting up the board. Not only that, these implementations also reflected a low possibility of expansion due to some missing function. This means that we had two designs left, of Nhat and Pham. A strength of Pham's design was its modularity, and the algorithms employed were understandable and transferable to different designs, but coupling presents a challenge for refactoring. In addition, his design followed the Builder pattern which makes the code easier to be extended for the next sprint. However, we all think that Pham's design was too complicated which is shown in the high CBO points, while Nhat's design is loosely coupled and highly understandable, making transferring the algorithms from Pham's implementation to Nhat's relatively easy, giving us the best of both.

3. CRC Cards

3.1 Final CRC design

i. Board

Board is responsible for concepts that relate to the movement and position of the player on the board. Other classes' interaction with the board will be fetching information about a player's position and requesting the board to move a player. Chit Card requests information about the type that a player is standing on, and it also requests the board to move the player by the number of moves on the card.

Classname: Board	
Responsibility	Collaborators
+ Keep track of player position + Keep track of whether game has ended + Moves player legally + Decide where caves are located + Keep track of what tiles/volcano has which type + Instantiate Cave for each player + Initialise player and cave position on the board + Keep track of what type of tile/volcano each player is standing on	Player Cave Chit Card

ii. Chit Card

Chit Cards are responsible for handling user selecting itself, and information about the Chit. It requires collaboration with the Chit Card Manager to get which player's turn it is and it requires collaboration with the Board to get what type volcano/tile the player is standing on. This two information is required to determine what action to take when a Chit Card is selected.

Classname: Chit Card	
Responsibility	Collaborators
+ Handle what happens when a chit card is selected (end turn or move player?) + Keep track of whether if the chit card was flipped + Keep track of what type it is + Keep track of how many move it makes the player do + Notify board of when and by how many move to move a player + Notify Turn Manager when a turn has ended + Can reset it flip state after a turn	Board Chit Card Manager Player

iii. Chit Card Manager

Chit Card Manager is responsible for managing all Chit Cards in the game and managing turns. Other classes can interact by requesting information about the player whose turn it is or requesting to change turn. The reason for grouping these responsibilities together is because turns only change as a result of selecting a chit card, and operations that apply to all chit cards are applies at the end of a turn, i.e resetting all chit cards.

Classname: Chit Card Manager	
Responsibility	Collaborators
+ Keep track of which player's turn it is + Change turn when get notified + Handles when turn change + Generate and track all chit card to be used in the game + Reset all Chit Card when turn ends	Chit Card Player

iv. Game/Starter

Game is responsible for initialising the game and connecting all related classes together for the game to render and run. It instantiates the other classes and connects them together via Dependency Injection.

Classname: Game	
Responsibility	Collaborators
+ Gather or generate game parameters (number of players, how volcano/tiles are ordered on the board) + Instantiate + Instantiate and inject dependencies for main game classes (Board, Chit Card, Chit Card Manager) + Instantiate User Interface classes + Stop application when there is a winner	Chit Card Board Chit Card Manager Player User Interface

v. User Interface

User Interface is responsible for retrieving information from a state-keeping class in the game and defining a method to convert information into a graphical format (rendering the game). These classes include Board, Chit Card Manager and Chit Card, there should be a separate User Interface class for each.

Classname: User Interface	
Responsibility	Collaborators
+ Retrieve the the state of a class in the game + Render to the screen the state retrieved from class + Does these operation periodically to keep up with state changes(every frame refresh)	Board Chit Card Chit Card Manager

vi.Player

Player is responsible for keeping information about its name and whether or not it has left its cave. Its only purpose is to hold information in order to support the operation of the Board, and be a unique representation of a player.

Classname: Player	
Responsibility	Collaborators
+ Keep track of whether or not it is in it's cave, this can be publicly accessed and updated + Keep track of its name + Act as a unique identifier for a player + Act as a more like a struct in C than a real class, store information needed by Board	Board Chit Card Chit Card Manager Game

vii. Cave

Cave holds information about what type it is and which tile/volcano card on the board it is connected to. Similar to Player, it holds information to support the operation of the Board and act as a unique representation of a cave.

Classname: Player	
Responsibility	Collaborators
+ Keep track of what animal type does it have + Keep track of which tile/volcano it is attached to + Act as a more like a struct in C than a real class, store information needed by Board	Board

3.2 Alternative Distributions of Responsibility

These alternatives have advantages to them that were discussed, but the disadvantages that come with them can outweigh the benefits.

- **Alternative distribution 1: Forgo User Interface, merge it's responsibility with game**

Advantage - The Game class already contains the state-keeping class and information that we need for rendering, meaning we don't have to worry about how to structure extra relationships and collaboration between extra classes, which reduces design complexity and improves ease of understanding. Having rendering logic centrally in one location can also make it easier to navigate the code base.
Disadvantage - Rendering logic might be extremely long, complex and might vary for different state-keeping classes (algorithm for rendering Board can be different from Chit Card). Keeping them all in one place is poor encapsulation and can be hard to maintain down the line.

- **Alternative Distribution 2: Forgo individual Chit Card, have Chit Card Manager handle player selecting chit card**

Advantages - This would be simpler and reduce the number or parts in the design. Logic for handling base game behaviour can be simpler as well, as we have one less relationship/collaboration to think about (between Chit Card and Chit Card Manager).

Disadvantages - Similar to Alternative distribution 1, the logic for handling player interaction can vary depending on the type of chit card. Handling selection centrally like this can be hard to maintain in the future, and provide challenges if we decide to introduce extra Chit Card type and behaviour in Sprint 4.

- **Alternative Distribution 3: Having the board manages turn**

Advantage - By design, the Board class already holds information on all players, so if we perform turn manager in Board we don't need to keep a copy of the player information, reducing redundancy. Board can request ChitCardManager to perform operations on all Chit Card at turn ending.

Disadvantage - The Board responsibility is already well defined, and adding turn management can violate the SRP (Single Responsibility Principle). Spreading it to another class distributes intelligence more evenly, and turn management is more closely related to chit cards, as interaction with chit cards is the only thing that can initiate turn change. Additionally, Chit Card Manager is relatively lightweight, so introducing extra relevant responsibilities can prevent it from becoming a vapour class.

- **Alternative Distribution 4: Having the player manages their position instead of board**

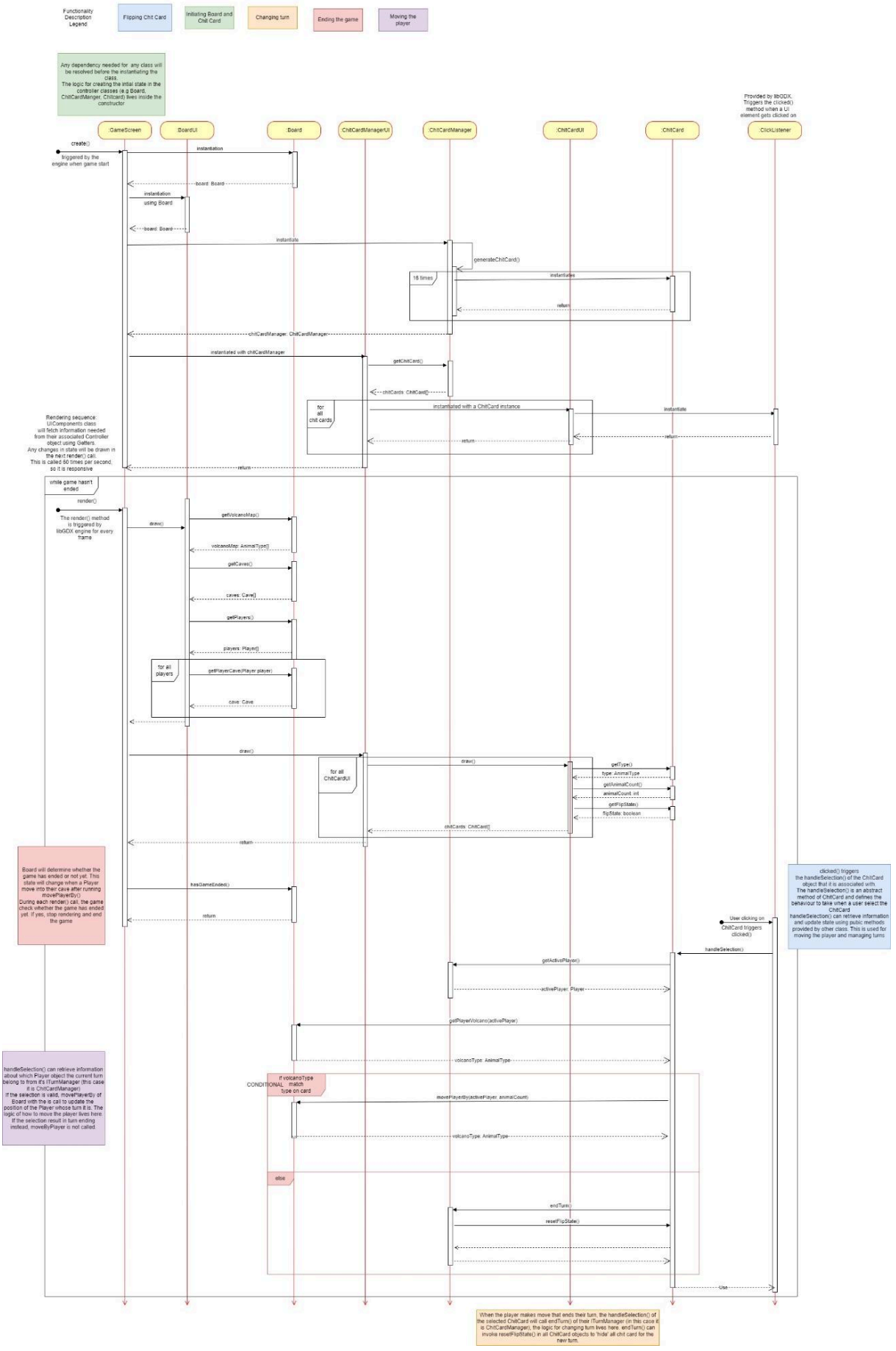
Advantage - Most operations with the player in Board will need information about where it is on the board, so it makes sense logistically to couple this information with the Player class.

Disadvantage - Since the position information is only relevant in the Board class and not anywhere else, introducing it could cause coupling intuitively. The Player class also collaborates with other classes beside the Board.

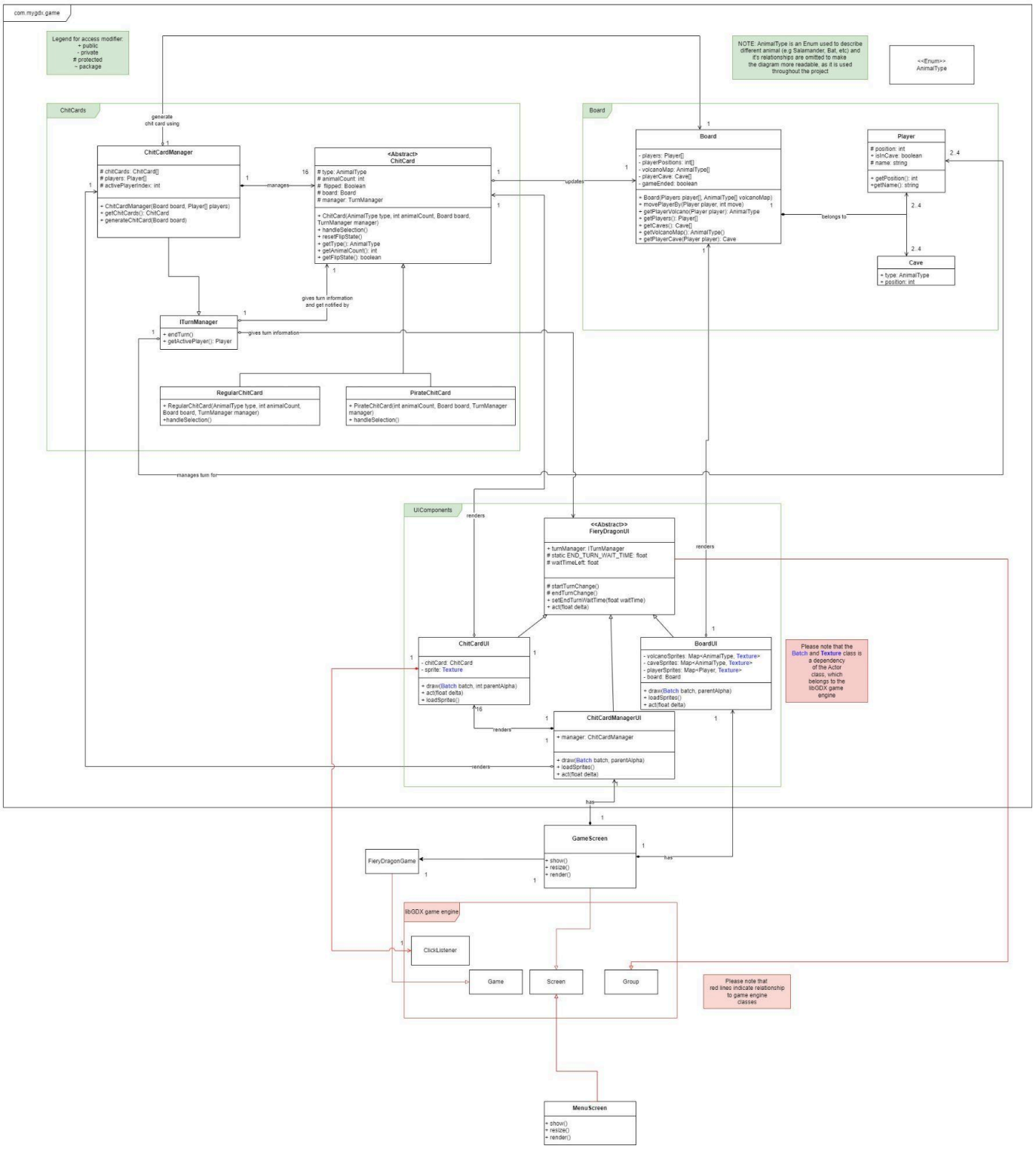
4. Sequence Diagram and Class Diagram

(If needed, there is a high resolution of this diagram under Sprint3_Team377/docs/classdiagram.jpg, and Sprint3_Team377/docs/sequencediagram.jpg)

SEQUENCE DIAGRAM



CLASS DIAGRAM



5. Video link

Drive: Final 3077.mp4
Youtube: https://youtu.be/z_xZe5oawuY