FIT3077 SOFTWARE ENGINEERING: ARCHITECTURE AND DESIGN

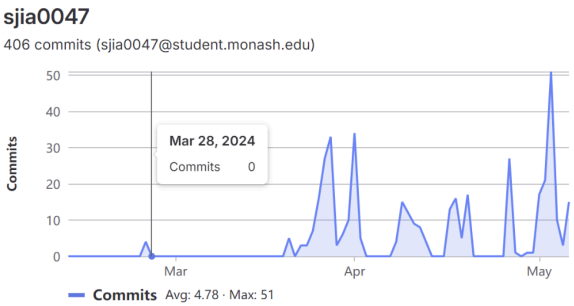
CL_Monday06pm_Team625 - Sprint 4

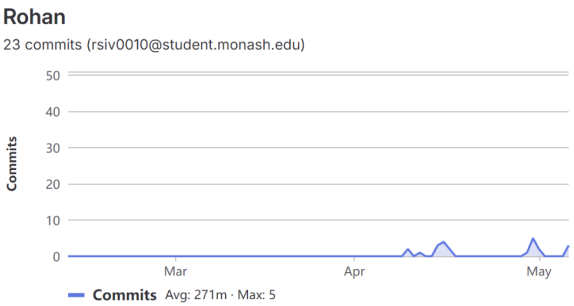
Team Members
Shen Jiang
Rohan Sivam
Zhan Hung Fu
Desmond Chong Qi Xiang

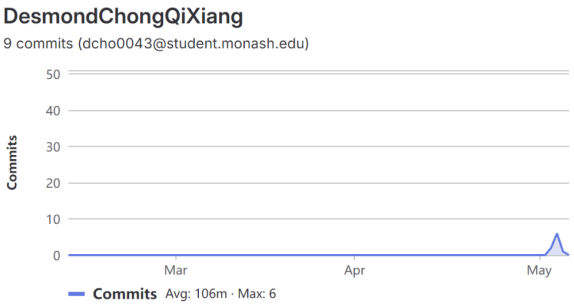
1.0 Contributor Analytics

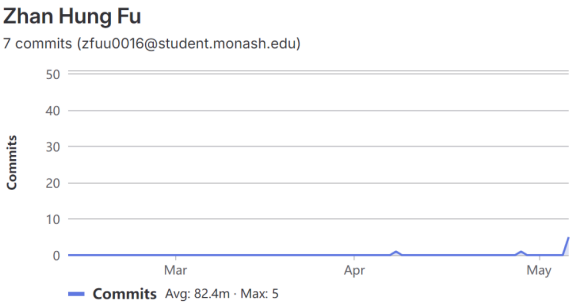
The contribution log is located in our git repositories history.

Though it may look like some of us hardly did much, all contributed roughly equally among all the other deliverables and code. Some commits are also large commits.









2.0 Self-Defined Extensions

Extension 1: Skip Chit Card
Description: Chit card that can skip other players turns by the indicated number when flipped by a player, once that player's turn ends. The number of turns skipped is not summed, rather it is the maximum that the player flipped before their turn ended.

Extension 2: Universal Tile
Description: A tile that when landed on allows a player to match any flipped animal chit card , allowing them to move no matter the animal indicated by a flipped chit card.

3.0 Class Diagram

The class diagram is attached below this page.

Python types used in class diagram:
Sequence: Read only list. Equivalent to <? extends class> in OOP languages
Optional: Can be null or have a value
*args: Accept any number of arguments

LEGEND
Green: New classes and methods that are associated with the save/load functionality
Red: New extensions & other refactoring

SwapPower
- game_board: GameBoard
+ __init__(Optional[GameBoard])
+ on_execute() : Void
+ use_game_board(GameBoard) : Void
+ on_save(dict[String, any]) : Optional[Any]
+ create_from_json_save(dict[String, any], Optional[GameBoard]) : SwapPower

SkipTurnPower
- turn_manager: TurnManager
- skip_no_of_players: Int
+ __init__(TurnManager, Int)
+ on_execute() : Void
+ on_save(dict[String, any]) : Optional[Any]
+ create_from_json_save(dict[String, any], TurnManager) : SkipTurnPower

<<abstract>>
Power
user: PlayableCharacter
+ __init__(PlayableCharacter = None)
+ execute() : Void
 # on_execute(PlayableCharacter): Void
+ set_user(PlayableCharacter): Void
+ on_save(dict[String, any]) : Optional[Any]

PowerChitCard
- power: Power
+ __init__(Int, Power, String, DrawProperties = None)
# on_draw_request(DrawProperties): [(DrawAssetInstruction, ModularClickableSprite)]
+ on_click(PlayableCharacter) : Void
+ create_from_json_save(dict[String, any], Power) : PowerChitCard

<<enum>>
Animal
.salamander
.baby_dragon
.spider
.bat
.universal

AnimalChitCard
- animal: Animal
+ __init__(Int, DrawProperties = None)
# on_draw_request(DrawProperties): [(DrawAssetInstruction, ModularClickableSprite)]
+ on_click(Optional[PlayableCharacter]) : Void
+ on_save(dict[String, any]) : Optional[Any]
+ create_from_json_save(dict[String, any]) : AnimalChitCard

PirateChitCard
+ __init__(Int, DrawProperties = None)
# on_draw_request(DrawProperties): [(DrawAssetInstruction, ModularClickableSprite)]
+ on_click(Optional[PlayableCharacter]) : Void
+ on_save(dict[String, any]) : Optional[Any]
+ create_from_json_save(dict[String, any]) : PirateChitCard

DefaultGameBoard
- main_tile_sequence: [Tile] = []
- starting_tiles: [Tile] = []
- starting_tiles_set : set(Tile) = set()
- starting_tiles_destinations_set : set(Tile) = set()
- chit_cards: [ChitCard] = []
- character_tiles_visited: dict[PlayableCharacter, set(Tile)] = dict()
- character_location: dict[PlayableCharacter, Int] = dict()
- character_starting_tiles: dict[PlayableCharacter, Tile] = dict()
- save_codec: SaveCodec[dict[str, Any]]
+ __init__(Sequence[Tile], [(Tile, Tile)], [ChitCard], [PlayableCharacter], SaveCodec)
- check_enough_main_tiles() : Void
- set_chit_card_draw_properties(): Void
+ move_character_by_steps(PlayableCharacter, Int) : Void
- move_character_to_tile(PlayableCharacter, Tile) : Void
- move_character_to_tile(PlayableCharacter, Int) : Void
+ get_character_floor_tile(PlayableCharacter) : Tile
+ get_closest_character(PlayableCharacter) : Optional[PlayableCharacter]
+ swap_characters(PlayableCharacter, PlayableCharacter) : None
+ on_player_turn_end() : Void
+ get_draw_assets_instructions() : [DrawAssetInstruction]
+ get_all_clickable_assets() : [ModularClickableSprite]
+ get_tiles_required() : Int
+ add_chit_card(ChitCard, bool, Int) :Void
- get_chit_card_safe_area() : ((Int,Int), (Int,Int))
- get_main_tile_sequence_properties() : MainTileSequenceProperties
+ on_save(dict[String, any]) : Optional[Any]

MainTileSequenceProperties
- main_width : Int
- main_height : Int
- main_x0 : Float
- main_x1: Float
- main_y0 : Float
- main_y1 : Float
- square_size : Float
__init__(Int, Int, Float, Float, Float, Float, Float)

<<abstract>>
ChitCard
- flipped: Bool
- symbol_count: Int
draw_properties: DrawProperties
board_delegate: GameBoard
+ __init__(Int = None, DrawProperties = None)
+ get_draw_clickable_assets_instructions() : [(DrawAssetInstruction, ModularClickableSprite)]
 # on_draw_request(DrawProperties): [(DrawAssetInstruction, ModularClickableSprite)]
+ on_click(PlayableCharacter) : Void
+ set_game_board_delegate(GameBoard) : Void
+ set_draw_properties(DrawProperties)
+ set_flipped(Bool) : Void
+ get_flipped() : Bool
+ get_symbol_count () : Int
+ on_save(dict[String, any]) : Optional[Any]

<<enum>>
PlayableCharacterVariant
.blue
.green
.orange
.purple

DrawAssetInstruction
- asset_path: String
- x: Int
- y: Int
- size: (Int, Int)
- rotate: Float
+ __init__(String, Int, Int, (Int, Int) = None, Float = 0)
+ get_asset_path(): String
+ get_x_coord(): Int
+ get_y_coord(): Int
+ get_size(): (Int,Int)
+ get_rotation(): Float

<<interface>>
ModularClickableSprite
+ on_click(Optional[PlayableCharacter]) : Void
+ get_draw_clickable_assets_instructions() : [(DrawAssetInstruction, ModularClickableSprite)]

<<interface>>
DrawableByAsset
+ get_draw_assets_instructions() : [DrawAssetInstruction]

<<interface>>
JSONSavable
+ on_save(dict[String, any]) : Optional[Any]

main.py

<<interface>>
GameConfiguration
+ generate_game_world(): GameWorld

PygameScreenContoller
- screen: Pygame.Surface
- image_cache: dict[String, Pygame.Surface]
- image_cache_resized : dict[String, dict[(int,int), Pygame.Surface]] = dict()
- __init__(self)
+ fill_screen_with_colour((Int,Int,Int))
+ draw_asset(str, int, int, int = 0, int = 0, float) : pygame.Surface
+ draw_drawable_by_asset(Sequence[DrawableByAsset]) : [pygame.Surface]
+ draw_modular_clickable_sprites(Sequence[ModularClickableSprite]) : [(pygame.Rect, ModularClickableSprite)]
+ draw_assets_from_instructions([DrawAssetInstruction]) : [Pygame.Surface]
+ get_screen_size() : (int, int)
+ instance() : PygameScreenController

GameWorld
- game_board: GameBoard
- turn_manager: TurnManager
- should_game_run: Bool
- mouse_click_enabled: Bool
- __init__(GameBoard, TurnManager)
+ run(params): Void
+ get_player_characters(): [PlayableCharacter]
- fire_onclick_for_clicked_hitboxes([pygame.Rect, ModularClickableSprite], PlayableCharacter)
+ on_player_win(PlayableCharacter) : Void
- process_current_player_turn(): Void
+ disable_mouse_clicks() : Void
+ enable_mouse_clicks() : Void
+ stop_game() : Void
+ instance(): GameWorld
+ load_from_json_dict(dict[String, any]) : Void

<<interface>>
GameBoard
+ get_characters_on_board: [PlayableCharacter]
+ move_character_by_steps(PlayableCharacter, Int) : Void
+ get_character_floor_tile(PlayableCharacter) : Tile
+ get_closest_character(PlayableCharacter) : Optional[PlayableCharacter]
+ swap_characters(PlayableCharacter, PlayableCharacter) : None
+ on_player_turn_end(): Void
+ get_all_clickable_sprites() : Sequence[ModularClickableSprite]
+ on_save(dict[String, any]) : Optional[Any]

<<abstract>>
PlayableCharacter
- draw_properties: DrawProperties
- name: String
is_currently_playing: Bool
- should_continue_turn: Bool
variant: PlayableCharacterVariant
+ __init__(DrawProperties = None, PlayableCharacterVariant, Int = 1)
+ set_draw_properties(DrawProperties) : Void
+ set_currently_playing(Bool): Void
+ should_continue_turn(): Bool
+ set_should_continue_turn(Bool): Void
+ name() : String
+ on_save(dict[String, any]) : Optional[Any]

Dragon
- draw_rotation : float
+ get_draw_assets_instructions() : [DrawAssetInstruction]
+ on_save(dict[String, any]) : Optional[Any]
+ create_from_json_save(dict[String, any]) : Dragon

DrawProperties
- coordinates: (Int, Int)
- size: (Int, Int)
- rotation: Float
+ __init__((Int, Int), (Int, Int), Float = 0)
+ get_coordinates() : (Int, Int)
+ get_size() : (Int, Int)
+ get_rotation(): Float

JSONSaveCodec
- json_dict: dict[String, any]
- saveables: [JSONSavable] = []
+ __init__(String)
+ registerSaveable(JSONSavable) : Void
+ receive_save_command() : Void
+ load() : dict[String, any]
+ save() : Void

<<bind>>
<T → dict[String, any]>

ArcadeGameConfiguration
- json_save_factory: JSONSaveClassFactory()
- json_save_codec: JSONSaveCodec
+ __init__(JSONSaveCodec)
+ generate_game_world(): GameWorld
+ on_save(dict[String, any]) : Optional[Any]
+ create_game_world_from_json_save(dict[String, Any]) -> GameWorld

JSONSaveClassFactory
- mapping: dict[ClassTypeIdentifier, Callable[..., Any]]
+ __init__(String)
+ create(ClassTypeIdentifier, dict[String, any], *args) : Any

<<abstract>>
SaveCodec<T>
save_path: String
+ __init__(String)
+ load() : T
+ save() : Void

SaveCommand<T>
- saver: SaveCodec<T>
+ __init__(SaveCodec<T>)
+ execute() : Void

<<interface>>
Command
+ execute() : Void

<<abstract>>
TurnManager
playable_characters: [PlayableCharacter]
current_player: PlayableCharacter
+ __init__([PlayableCharacter], Int)
+ get_number_players() : Int
+ get_currently_playing_character() : PlayableCharacter
+ get_tracked_playable_characters(): [PlayableCharacter]
+ skip_to_players_turn(PlayableCharacter): Void
+ skip_to_player_on_turn_end(PlayableCharacter): Void
+ get_player_character_n_turns_downstream(Int, PlayableCharacter) : PlayableCharacter
+ tick() : Bool
+ on_save(dict[String, any]) : Optional[Any]

DefaultTurnManager
+ __init__([PlayableCharacter], Int)
- process_current_player_turn(): Void
+ get_number_players() : Int
+ get_currently_playing_player() : PlayableCharacter
+ skip_to_player_on_turn_end(PlayableCharacter): Void
+ get_player__character_n_turns_downstream(Int, PlayableCharacter) : PlayableCharacter
+ tick() : Bool

<<abstract>>
Tile
draw_data: DrawProperties
- character: PlayableCharacter
- animal: Animal
+ __init__(DrawProperties = None, PlayableCharacter = None, Animal = None)
+ get_character_on_tile(): PlayableCharacter
+ set_character_on_tile(PlayableCharacter): Void
+ get_animal(): Optional[Animal]
+ set_draw_data(DrawProperties) : Void
+ set_character_on_tile(PlayableCharacter) : Void
+ place_character_on_tile(PlayableCharacter, Bool) : Void
 # on_draw_request(DrawProperties) : [DrawAssetInstruction]
default_character_draw_instructions() : [DrawAssetInstruction]
default_animal__draw_instructions(): [DrawAssetInstruction]
+ on_save(dict[String, any]) : Optional[Any]

NormalTile
+ __init__(Animal, DrawProperties = None, PlayableCharacter = None)
+ place_character_on_tile(PlayableCharacter) : Void
# on_draw_request(DrawProperties) : [DrawAssetInstruction]
+ on_save(dict[String, any]) : Optional[Any]
+ create_from_json_save(dict[String, any]) : NormalTile

CaveTile
- variant = CaveTileVariant
+ __init__(Animal, CaveTileVariant, DrawProperties = None)
+ place_character_on_tile(PlayableCharacter) : Void
# on_draw_request(DrawProperties) : [DrawAssetInstruction]
+ on_save(dict[String, any]) : Optional[Any]
+ create_from_json_save(dict[String, any]) : CaveTile

<<enum>>
CaveTileVariant
.blue
.green
.orange
.purple

<<interface>>
WinEventListener
+ on_player_win(PlayableCharacter)

WinEventPublisher
- subscribers: [WinEventListener]
- __init__()
+ subscribe(WinEventListener): Void
+ unsubscribe(WinEventListener) : Void
+ notify_subscribers(PlayableCharacter) : Void
+ instance() : WinEventListener

4.0 Extension Reflections

In this sprint, we implemented the required “New Dragon Card 2” extension, alongside the ability to save and load game states. For our self-defined extensions, we introduced a new universal tile that allows a player to match any chit card animal when stepped on, and added a new chit card type that skips a set number of turns after the current player’s turn ends.

Incorporating these extensions into our design and implementation from Sprint 3 was relatively straightforward with the exception of the save/load functionality. This feature needed to capture the entire state of the game board, including the new universal tiles, so it was crucial to determine the suitable data structure for storing the save files. We evaluated various options and decided that using JSON to store the current player’s turn, game state, and other relevant information was the most suitable choice. However, we also had to consider other methods, such as how to efficiently encode and decode the data, and how to manage the save files to ensure consistency and integrity. Therefore, to ensure its successful implementation, we prioritised completing the self-defined extensions first to avoid errors during the save/load process.

Dragon Card 2 (i.e. Swap Position Chit Card) and Skip Chit Card extensions:

For the extensions of adding new chit card types, we took a highly extensible approach. We began by creating an abstract ‘TurnManager’ class, allowing any inheriting classes to modify turn handling. Recognising the “skip player turns” and “swap positions” actions as distinct powers, we designed an abstract ‘Power’ class, from which the ‘SkipTurnPower’ and ‘SwapPower’ classes inherit. We then introduced a ‘PowerChitCard’ class, a specialised chit card encapsulating a ‘Power’ object, to activate unique effects upon clicking.

To implement the “skip player turns” power, we introduce an additional attribute called ‘__players_to_skip’, which specifies the number of players to skip ahead of the current player upon execution. We then leverage the ‘TurnManager’ class to seamlessly skip the configured number of turns once the current user’s turn ends.

While the “skip player turns” power was relatively straightforward to implement, the “swap positions” power effect presented a more complex challenge. To implement this feature, we calculate distances between characters within the ‘DefaultGameBoard’ class. A dictionary named ‘__character_location’ stores character positions, where the key is the character and the value is their location. As we iterate through the dictionary, we exclude the current player and calculate the forward and backward distance between each character and the other characters. To ensure valid swaps, we reference the ‘__starting_tiles_set’ (defined in Sprint 3) to avoid swapping characters standing on cave tiles. This distance calculation process is straightforward. After determining all distance values, we select the minimum distance and execute the position swap between the two corresponding characters.

Implementing both chit card extensions presented a **moderate** level of difficulty. Throughout sprints 2 and 3, we consistently adopted an abstract approach to design, so we were familiar with the concepts of abstract classes and inheritance. However, designing such a robust and extensible ‘Power’ class required careful consideration to ensure it could accommodate a wide

range of future power type additions. On the other hand, focusing on abstraction in our codebase has yielded significant improvements. Reusing code through the abstract 'Power' class has cut down on unnecessary redundancy. Without this abstract approach, adding a new power type would mean rewriting the same code for user assignment, execution checks and other boilerplate logic. Thus, making the codebase larger and more difficult to maintain.

Our Sprint 3 design decisions facilitated the implementation of the power chit card extension. The existing abstract 'ChitCard' class provided a solid foundation and clear integration points for our new 'PowerChitCard'. However, we believed that we could further streamline this process by applying the Decorator design pattern to our chit cards. This pattern would allow us to dynamically add powers to existing chit card types without the need to create new subclasses like 'PowerChitCard'. This can be done by creating a 'ChitCardDecorator' class which wraps a base 'ChitCard' object and delegates its core functionality to it. It would also contain a reference to a 'Power' object. When the decorated chit card is clicked on, the decorator would trigger the associated power's 'execute()' method in addition to the base chit card's behaviour (i.e. flipping itself and moving players around the game board).

Overall, this design decision allows us to easily introduce various new power types into chit cards in the future, simply by creating new classes that inherit from the 'Power' class. This focus on abstraction throughout our code significantly enhances its extensibility, ensuring that our implementation can readily accommodate future additional types of power without requiring major refactoring.

Save/Load extension:

It was easy to add the saving and loading functionality to each of the classes in our Sprint 3 design. As our classes adhered to the single responsibility principle and dependency inversion principle for all concrete classes, adding the additional methods on how each concrete class is saved and loaded was easy because each class only held state corresponding to its own. Hence, there was no need to refactor when adding the save/load functionality nor did it cause bugs with the functionality of other classes, mainly stemming from the open/closedness from following the two aforementioned principles for our sprint 3 submission.

Implementing the saving and loading functionality system as a whole presented a **high** level of difficulty. The difficulty mainly stemmed from figuring out how to convert data to the json format and back from the json format. Converting the data back into usable objects in the correct order so that the required dependencies were present at the correct times proved to be difficult, but that was not due to our Sprint 3's design but mainly our inexperience in creating and using save files. The main roadblock was the correct structure the save file needed to take in order to be able to load those dependencies at runtime. Figuring this out required a lot of experimenting and re-structuring of the JSON save file as details missed came to light. Furthermore, it was difficult to decide on what type of helper classes and data were needed to construct the concrete objects from the save file, requiring us to evaluate the benefits of certain approaches such as encoding with the help of satellite binary files, or completely sticking to a JSON format.

We eventually landed on using string identifiers and factories after seeing that the save files and code would be the most readable and learnable this way (due to not using a mix). Finally, due to the numerous classes that our sprint 3 design was composed of, we needed to delegate

a larger portion of time to implementing the behaviour of saving and loading for each of the concrete classes, leaving us with less time for polishing our other extensions.

Universal Tile extension:

For our final self-defined extension, the universal tile, our Sprint 3 design and implementation provided a strong foundation for accommodating different tile types. We treated the universal tile as an animal tile where each tile, except the dragon pirate chit card, has an associated “animal” enum. To visually distinguish the universal tile, we modified its asset to display a star image instead of an animal.

Implementing the universal tile presented a **low** level of difficulty. To enable the universal tile to match any chit card, we simply made adjustments to the animal chit card class’ ‘on_flip’ method. Initially, if the chit card’s animal didn’t match the tile’s animal, it would not move the player and instead end their turn. However, as we aimed to make all chit cards match a universal chit card, we needed to alter this behaviour. We added a check to see if the animal didn’t match, and if the tile’s animal was universal, the player would move forward regardless, and their turn would not end.

However, we realised that there was a more flexible and scalable approach. Rather than directly modifying the ‘on_flip’ method of the animal chit card class, we can introduce a ‘TileManager’ class that is responsible for determining the outcome of a player landing on a tile based on the chit card they picked. This ‘TileManager’ class could have a method like ‘resolveTileInteraction(tile, chitCard)’ that determines the effect of landing on a specific tile. For the universal tile, we can then modify the ‘TileManager’ to handle universal tiles instead of directly checking for a match between the chit card’s animal and the tile’s animal in the ‘on_flip’ method. So that when a player lands on a universal tile, the ‘TileManager’ can check if the current chit card matches the tile’s animal. If it does, the player moves forward as normal. If it doesn’t, the ‘TileManager’ can then check if the tile’s animal is universal. If it is, the player can still move forward without ending their turn. This approach decouples the logic of tile interactions from individual chit card classes, allowing for easier addition of new tile types in the future, as they can be handled by the ‘TileManager’ without needing to modify existing classes.

Overall, our design approach in Sprint 3 has facilitated the implementation of this extension. By treating the universal tile as a specialised case of an “animal” object, we were able to integrate it seamlessly into our existing code. This design not only made our codebase more modular and extensible but has also simplified the process of adding new tile-related features and extensions to our game in the future.

Human Values (Hedonism):

For our Fiery Dragons game, the extensions we’ve developed contribute to the human value of pleasure. The universal tile extension adds to the players’ pleasure by providing them with a versatile tile that allows them to forward regardless of the animal they match. This flexibility can be enjoyable as it reduces frustration and adds an element of unpredictability to the game, keeping players engaged and excited about their progress.

Similarly, the skip chit card extension can also enhance player's pleasure by introducing a strategic element to the game. Players can use this chit card to strategically sabotage other players, potentially advancing their own position in the game faster than others. This adds a layer of excitement and competition, increasing the overall enjoyment and satisfaction players derive from playing the game.

Conclusion:

In Sprint 4, our team successfully implemented the required "New Dragon Card 2" extension, save/load functionality, a new universal tile that matches any chit card animal when stepped on, and a new chit card type that skips a set number of turns after the current player's turn ends. This sprint presented various challenges and opportunities for improvement in our development practices.

Reflective Analysis:

1. **Save/Load Functionality Challenges**: The implementation of save/load functionality was more challenging than anticipated. One of the main challenges we encountered was determining the correct structure for the save file. We had to ensure that it stored the game state in a way that preserved its integrity and consistency. Our lack of experience in creating and using save files became evident during this process, especially when converting data to and from JSON format. Managing dependencies and ensuring the correct order of operations during loading proved to be particularly challenging. Ultimately, we decided to use string identifiers and factories to construct objects from the save file, ensuring that both the save files and the code remained understandable. Implementing this system also required a significant amount of time and effort. Because our Sprint 3 design comprised numerous classes, we had to dedicate a substantial portion of time to implement the save/load behaviour for each class, leaving us with less time to refine other extensions.

2. **Extensibility and Abstraction**: The design decisions made in Sprint 3, such as the use of abstract classes and inheritance, significantly facilitated the implementation of new chit card types in Sprint 4. However, we identified areas where further abstraction, such as applying the Decorator design pattern to chit cards, could enhance extensibility and reduce redundancy in our codebase.

3. **Time Management and Planning**: While our team successfully completed the sprint tasks ahead of schedule, we realised that better time management and planning could have improved our efficiency. Adding new chit card types was straightforward, which could have allowed us to allocate another team member for the save/load functionality or spend more time planning and researching suitable data structures.

We have come up with these strategies for improvement based on our experiences in Sprint 4:

1. **Utilise Online Resources**: We should leverage online resources like StackOverflow more effectively when facing technical challenges or needing clarification on concepts. This can provide quick solutions and insights from the community.

2. **Attend Consultation Sessions**: Attending consultation sessions with FIT3077 tutors can provide valuable advice and guidance, particularly on complex concepts or design patterns.

3. **Reviewing Lecture Materials**: Regularly reviewing lecture materials, especially those covering design patterns and their practical application (such as in Week 7), can help improve our overall understanding and ability to apply these patterns in our Fiery Dragon game implementation, leading to better code quality.

In conclusion, the challenges faced in Sprint 4 have provided us with valuable insights into areas for improvement in our development practices. By implementing these strategies, we aim to enhance our skills and practices to achieve better outcomes in similar future projects.

5.0 Executable

Description: A package containing the entire game built from our source code, allowing the game to be run on any windows computer upon execution.

Target platform: Windows 11

How to run the executable:

1. Extract all the files in the zip containing the executable into a separate folder
2. Run the executable called ‘main’ located in the folder you just extracted to

How to build the executable from the source code:
See the video for a visual guide on how to build the executable from the source code.

Steps:

1. Open up the source code folder (called Sprint4_Game)
2. Open up a terminal in that folder, ensuring that the terminal is pointing to the source code folder
3. Activate the virtual environment by running ‘.venv/Scripts/Activate’ in the terminal without the apostrophes
4. Run ‘pyinstaller main.py’ in the terminal without the apostrophes
5. Wait until the executable is successfully built. You will know that the executable is built when the terminal once again awaits your input.
6. Close the terminal.
7. Ensuring that you are in the source code folder (i.e Sprint4_Game), copy the folder called ‘assets’.
8. Navigate to Sprint3_Game/dist/main/_internal. Paste the ‘assets’ folder you copied into this directory.
9. Navigate to Sprint3_Game/dist/main
10. You can now run the executable called ‘main’ inside the directory to launch the game.

6.0 Considerations

None.