Sprint 3 Documentation

Team 625: Desmond, Shen, Zhan Hung Fu (Ian), Rohan

Table of Contents

1 Sprint 2 prototype reviews

1.0 Metrics

The following are the metrics we used to evaluate the sprint 2 prototypes.

Additional assessment criteria that were included, allowing us to make a more informed decision were:

- Faultlessness
- Reusability
- Learnability

1.0.1 Completeness of the solution direction

Functional Completeness
Definition: The degree to which all required functionalities are included

Key terms:

- Minor functionality: Functionality that makes up a key functionality
- Key functionality: Major requirement as specified by the client

Values

- Excellent - Covers all required functionalities
- Average - Missing 1-2 minor functionalities
- Poor - Missing one or more key functionalities, or missing many (>2) minor functionalities

Functional Correctness
Definition: The degree to which the system represents what the client wants

Values

- Excellent - Functionalities match what the client required exactly
- Average - Some functionalities differ slightly from what the client required
- Poor - Some functionalities differ substantially from what the client required

1.0.2 Rationale behind the chosen solution direction

Functional Appropriateness
Definition: The degree to which the system facilitates what users want to accomplish

Values

- Excellent - The game contains no unnecessary functionalities
- Average - The game contains some unnecessary functionalities (1-2)
- Poor - The game is convoluted and contains many unnecessary functionalities (>2)

1.0.3 Understandability of the solution direction

Appropriateness Recognizability
Definition: The degree to which the user can identify that the solution is what they need

Values
- Excellent - The user can identify all components and functions of the game
- Average - The user can identify most components and functions of the game, only being unable to identify at most one component
- Poor - The user can only identify a few components and functions of the game (user is not able to identify more than one required component)

1.0.4 Extensibility of the solution direction

Modifiability
Definition: The degree to which the system can be effectively modified without introducing defects to the system

Values
- Excellent - The system can be modified and requires a small amount of refactoring to maintain correctness
- Average - The system can be modified and requires a medium amount of refactoring to maintain correctness
- Poor - The system is hard to modify and requires a huge amount of refactoring to maintain correctness

1.0.5 User Interface

User Engagement
Definition: The degree to which the interface is inviting and motivating for a user

Values
- Excellent - Clean and simple interface and fast
- Average - Slightly messy or convoluted interface or moderately laggy
- Poor - Very messy or convoluted interface or extremely laggy

1.0.6 New assessment criteria

Faultlessness
Definition: The degree to which the system performs the required functionalities without errors (bugs) normally

Values
- Excellent - The game does not have any errors

- Average - The game has at least one non-breaking error
- Poor - The game has at least one error that breaks it (i.e causes a crash)

Reusability
Definition: The degree to which a part of a system can be used in other parts of a system / to build other parts of a system

Values
- Excellent - The majority of the system can be reused
- Average - Some of the system can be reused
- Poor - None/few components of the system can be reused

Learnability
Definition: The degree to which the functions of the system can be learnt by the user in a given period of time

Values
- Excellent - The game is easy to learn and use
- Average - The game is not easy nor hard to learn and use
- Poor - The game is hard to learn and use

1.1 Reviews

In this section, we review each sprint 2 prototype for each team member and assign ratings to each metric defined in section 1.0.

1.1.1 Shen's Prototype

Functional Completeness
Rating: Excellent

The set-up of the board which involves the initialisation of dragon tokens, tiles, caves and chit cards when the game is launched was fully implemented. The other key functionality involving the flipping of chit cards was also fully implemented, with the chit cards being able to be flipped at any time when the user clicks on them.

Functional Correctness
Rating: Poor

Correct number of tiles and chit cards. Chit card positions were randomised as required. Chit cards correctly contained an indicator for the number of symbols, and displayed the symbols on them.

The tiles making up the main sequence were randomised which was against game rules. The tiles should not be randomly shuffled and should be grouped into volcano tiles (considered a minor deviation from what the client wants). The initial board setup needs the four dragon pirate chit cards (considered a major deviation from what the client wants). Lastly, the chit cards can also be unflipped once flipped, which is against game rules.

Functional appropriateness
Rating: Excellent

Current functionalities include only the flipping of chit cards by clicks and initialisation of the game board. Hence, there were no unnecessary functionalities.

Appropriateness Recognizability
Rating: Excellent

All components (i.e. which animals are on the tiles that the dragons move on, which tiles represent the caves, which sprites are the dragons, which sprites represent the chit cards) of the game are easily identifiable. Functionalities such as clicking on a chit card are responsive upon click.

Therefore, users can easily see if the functionalities and interface of the game are appropriate for an implementation of the Fiery Dragons game.

Modifiability
Rating: Excellent

Referring to the class diagram,

There are no circular dependencies between modules meaning that changing one module will not always require changing countless other modules at the same time, improving modifiability,

Most modules don't have many dependencies (at most 5). Because most classes have high cohesion (i.e. the correct functionalities and state are in the correct classes), any changes are not likely to cause any changes to the dependents. Coupling is also low (there are not many strong associations compared to the more numerous weaker dependencies between modules).

Examples: You can relocate the cave tile to any position on the board and determine whether the dragon tokens start in their caves or on normal tiles without causing required modifications to other components. Additionally, the board size is adjustable, offering the ability to make it larger or smaller as desired.

User Engagement
Rating: Excellent

Clean simple visuals. Simple chit card, tile, dragon, animal and dragon graphics. The dragons do not cover the animal on the tile they are on. Finally, the game is fast.

Faultlessness
Rating: Excellent

No game-breaking errors or bugs were found. The game fully works as expected.

Reusability
Rating: Excellent

Abstract classes PlayableCharacter, ChitCard, and Tile reduce repetition, by providing default initialisation behaviour and default implementation of methods useful for any new type of character, chit cards, game boards and tiles. For example, we can easily get character drawing instructions for any tile, which automatically retrieves the correct drawing instructions for drawing characters on top of tiles.

The use of the facade PygameScreenController reduces repetition by allowing for the reuse of complicated drawing logic via methods exposed by the facade.

There is also a utility file that can be used to prevent repetition of the code by automatically calculating object position based on data passed in by objects that require it.

Learnability

Rating: Excellent

Simple interface that maps to the real Fiery Dragons game as expected. Chit cards respond as expected when clicked on. All components (chit cards, tiles (and their animals), caves, dragons) are easily identifiable.

1.1.2 Rohan's Prototype

Functional Completeness
Rating: Excellent

All required functionalities are completed.
The initial setup of the board is correct (i.e. tiles, caves, dragons, chit cards) and chit cards can be flipped by clicking on them.

Functional Correctness
Rating: Excellent

The main tiles are made of volcano cards, and stay consistent through multiple game launches which match the rules for the actual game. The number of tiles is correct.

There are 16 chit cards (which is consistent with the game rules), and the distribution of the chit cards in terms of the symbols (dragon pirate/animal) on them and their count matches the game rules. The chit cards also cannot be unflipped once flipped which is correct.

Functional appropriateness
Rating: Excellent

Current functionalities include only the flipping of chit cards by clicks and initialisation of the game board. Hence, there were no unnecessary functionalities.

Appropriateness Recognizability
Rating: Excellent

All components (i.e. which animals are on the tiles that the dragons move on, which tiles represent the caves, which sprites are the dragons, which sprites represent the chit cards) of the game are easily identifiable. Functionalities such as clicking on a chit card are responsive upon click.

Therefore, users can easily see if the functionalities and interface of the game are appropriate for an implementation of the Fiery Dragons game.

Modifiability
Rating: Poor

The GameBoard module essentially represents and handles the entire game. It is a god class. It handles drawing, deciding how to place players on the tiles on game startup, how to draw dragons, chit cards, game's initial state, among others. Cohesion is extremely low within this module as many unrelated components and their state are placed together, and the module is coupled entirely to numerous concrete classes.

Any modification to any module in the game will likely require modification in the GameBoard. Repetition is rampant within the class, meaning any small modification will result in changes in many places within the module. Repetition is also observed in the child modules of the abstract module ChitCard (representing chit cards). Hardcoding of tile positions also means changes likely need to be made repeatedly via trial and error to ensure correctness.

User Engagement
Rating: Poor

Sprites (chit cards, tiles, caves, dragons) are modern and elegant.

The interface is a bit messy with unnecessary black backgrounds showing up when a chit card is flipped. Dragons block the animal on the tile on which they are residing, making it impossible to see which chit card they have to flip to move. The game is extremely laggy, taking ~2 seconds to respond to a click.

Faultlessness
Rating: Poor

The game breaks randomly when flipping chit cards, and breaks if you click something other than the chit card.

Reusability
Rating: Poor

No utility methods. The module representing the game board is a god class and is coupled to many concrete classes. It is highly unlikely that the methods in the game board module can be reused by child classes without overriding many methods to achieve the desired functionality. For example, a new game board type that explores a different way of representing chit cards, tiles, etc. will not be able to easily extend the game board module. With the game board serving to represent the entire game, this has a significant influence on the poor rating.

Lacks any use of interfaces. Little to no utility methods.

Chit card and tiles abstract classes provide default implementations for several methods, allowing for some reuse of logic for new types of chit cards and tiles.

Learnability
Rating: Excellent

Simple interface that maps to the real Fiery Dragons game as expected. Chit cards respond as expected when clicked on (although with a delay). All components (chit cards, tiles (and their animals), caves, dragons) are easily identifiable.

1.1.3 Ian's Prototype

Functional Completeness
Rating: Excellent

All functionality has been completed including setting up the game board (creating the tiles, caves, chit cards, dragons, randomised chit card position) and the flipping of chit cards when they are clicked on.

Functional Correctness
Rating: Excellent

The gameboard is set up correctly with original game functionality (4 caves, 4 dragon tokens, 16 chit cards and the 24 tiles). The individual tiles that compose the volcano tiles are not scrambled on every game launch as expected, and the chit cards are initialised with different positions as expected. The chit cards can also be flipped when clicked on, and cannot be unflipped once flipped as expected.

Functional Appropriateness
Rating: Excellent

Current functionalities include only the flipping of chit cards by clicks and initialisation of the game board. Hence, there were no unnecessary functionalities.

Appropriateness Recognizability
Rating: Average

It is easy to recognise the tiles, the caves, the animals on them and the dragon tokens, allowing the user to judge appropriateness. The animal representing each tile is also clear through the graphics. Functionalities such as chit card flipping are responsive when actioned by the user, making it easier to determine if the functionality is appropriate or not. However, the lack of graphics for the back of chit cards made it more difficult to identify whether they were actually the chit cards or not before clicking on them.

Modifiability
Rating: Poor

The code for creating tiles and chit cards is hard coded. The number of tiles and chit cards, and the position of the tiles cannot be changed easily without needing to alter other parts of the code.

The module for the game board is a god class as it contains three separate responsibilities, drawing the concrete tiles, chit cards and dragons. As it contains drawing responsibilities for three separate elements, changes to drawing logic for one will likely need to propagate to the other two, making it hard to modify without introducing defects. Furthermore, cohesion is very low as the class does not work on most of the state it contains. Changes may need to be made in multiple places in the module that contain relevant logic as a result.

User Engagement
Rating: Average

Slightly messy interface due to the very pixelated animals on the tiles. Hard to identify chit cards due to missing back graphics. Animals on the chit cards are also stretched, making it harder to identify the exact animal on them. Dragons block the animal on the tile on which they are residing, making it impossible to see which chit card they have to flip to move. The game is very fast, however.

Faultlessness
Rating: Excellent

No game-breaking bugs or errors. Clicks flip the chit cards as expected, and clicking at locations other than the chit cards does not break the game.

Reusability
Rating: Poor

The game board which forms the majority of the game is hard coded, making it extremely difficult for new types of game boards to extend from it without having to override a lot of behaviour. As the game board forms a large portion of the game's functionality, this has a significant impact on the poor rating.

Abstract classes for tiles allow for the reuse of state initialization logic, but other than that there is no behaviour defined in them that can be reused for any new child modules. Chit cards also have an abstract class, and this class contains flipping behaviour and state initialization that can be reused by new types of chit cards.

Learnability
Rating: Excellent

Simple interface that maps to the real Fiery Dragons game as expected in terms of the location of elements. Tiles (and their animals), caves and dragons are easily identifiable. Chit cards

respond as expected when clicked on (although with a delay). Though the chit cards are harder to identify, the chit cards can still be discovered quickly with little experimentation.

1.1.4 Desmond's Prototype

Functional Completeness
Rating: Excellent

All functionality has been completed including setting up the game board (creating the tiles, caves, chit cards, dragons, randomised chit card position) and the flipping of chit cards when they are clicked on.

Functional Correctness
Rating: Average

The gameboard is set up correctly with original game functionality (4 caves, 4 dragon tokens, 16 chit cards and the 24 tiles).

On every game launch, the individual tiles that compose the volcano tiles are not scrambled whilst the chit card positions are scrambled as expected. The distribution of the chit cards in terms of the symbols (dragon pirate/animal) on them and their count matches the game rules. The chit cards can also be flipped when clicked on, but can be unflipped once flipped which is against the game rules.

Functional Appropriateness
Rating: Average

Key functionalities such as the flipping of chit cards by clicks and initialisation of the game board were included. However, the inclusion of a player selection was not something the client had requested thus far and does not facilitate what the client wants.

Appropriateness Recognizability
Rating: Excellent

All components (i.e. which animals are on the tiles that the dragons move on, which tiles represent the caves, which sprites are the dragons, which sprites represent the chit cards) of the game are easily identifiable. The chit cards flip responsively when clicked on, although with a slight delay.

Therefore, users can easily see if the functionalities and interface of the game are appropriate for an implementation of the Fiery Dragons game.

Modifiability
Rating: Average

The game board module depends on hard coded constant values, of which the meanings of each constant are hard to discern due to its location in a separate file that contains many unrelated constants. These constants seem to be implementation details (e.g. there exists one constant that specifies the location of each of the tiles on the game board) of the game board module, which means changing them will likely cause game-breaking errors.

However, the drawing logic has been delegated to the corresponding each of game elements (e.g. chit cards or tiles), meaning the game board does not handle unrelated drawing responsibilities. Chit cards also have some behaviour (e.g. flipping) that is delegated to them by the game board. This makes changing functionalities for these game elements more forgiving.

User Engagement
Rating: Poor

Sprites used for the animals are blurry, making it harder to distinguish which animal is which.
The background on each of the tiles is excessive, making it difficult to make out the animal on each tile.
The dragons cover the animal on the tile they are on, making it impossible to know which chit card they need to flip next to move (if they forget).
The game is very laggy (~1-second delays on average).

Faultlessness
Rating: Average

When flipping chit cards quickly, the wrong chit cards tend to be flipped (minor error). However, everything else works as expected. The initialisation of the game board does not cause errors, and clicking in places other than the chit cards does not break the game in any way.

Reusability
Rating: Average

The game board specifically is coupled to many concrete modules (e.g. animal chit cards, dragon pirate chit cards, dragon tokens, etc.), and hence will not be able to be easily extended by game boards who want to handle chit cards, playable tokens and tiles differently.

However, there is some degree of reusability in other modules. The chit cards implement common drawing and flipping functionality. New chit cards can extend from these classes and not have to re-implement these functionalities. Tiles and caves also implement common drawing functionality, allowing new types of tiles or caves to specify other responsibilities without having to write out the drawing logic again.

Learnability
Rating: Excellent

Simple interface that maps to the real Fiery Dragons game as expected in terms of the location of elements. Tiles (and their animals), caves, chit cards and dragons are easily identifiable. Chit cards respond as expected when clicked on (although with a delay).

1.1.5 Review Summaries

Here we list a summary of the key findings of each person's prototype.

Shen's prototype

Key findings:

- There were no god classes, and the cohesion of each module was high
- The game's modules do not depend on hardcoded values, making the game more flexible
- Due to the use of multiple modules to separate concerns, the game was found to be highly modifiable
- The use of many abstractions assists in making extensions to existing functionality easy
- Tiles making up the volcano cards were randomised, which was against game rules
- Missing pirate chit cards.
- Chit cards can be unflipped once flipped which is against game rules
- Simple, modern interface. Dragons do not block animals on tiles

Overall a very robust design, but will require some modifications to maintain functional correctness.

Rohan's prototype

Key findings:

- The user interface for the two key functionalities (flipping of chit card and initialization of game board) matches all elements of the Fiery Dragons game rules exactly
- The game's modules depend on hardcoded values, making the game likely to break upon modification of those values
- Extremely laggy
- Crashes when clicking on elements other than chit cards, and randomly when flipping them
- Dragon tokens block the animal of the tiles they are on
- Changing a portion of the code requires major changes to other parts of the system due to misplaced responsibilities and a large number of concrete dependencies
- Most of the code base cannot be reused, making it hard to extend from

Overall the design is not well thought out and will require large amounts of refactoring to address its issues and improve the developer experience. The game's performance will also need to be improved.

Ian's prototype

Key findings:

- The user interface for the two key functionalities (flipping of chit card and initialization of game board) matches all elements of the Fiery Dragons game rules exactly
- The game's modules depend on hardcoded values, making the game likely to break upon modification of those values
- Changing portions of the code requires lots of effort to maintain correctness due to misplaced responsibilities
- Reusability is poor due to the lack of behaviour in abstract classes
- The prototype has no game breaking or minor errors
- It is hard to identify the chit cards on the game board as the chit cards lack graphics when not flipped
- The UI is slightly messy due to the pixelated animals on the tiles and stretched graphics on the chit cards
- The game is not laggy

Overall, the design is not well thought out. The prototype will require a large refactoring effort to address its issues before we can start adding new features.

Desmond's prototype

Key findings:

- The user interface contains an unnecessary player selection screen
- The game's module depends on hardcoded values, making the game likely to break upon modification of those values
- Moderate degree of reusability. The drawing functionality for chit cards and tiles can be reused for new chit cards and tiles
- Modifications to the drawing functionalities of the game can be made easily without breaking the game due to the code's modular nature
- Chit cards can be unflipped once flipped, which is against game rules
- The wrong chit cards are flipped when flipping chit cards quickly
- The UI is quite messy. The sprites used for the animals on the tiles and chit cards are blurry, and the background on the tiles makes it harder to discern the animals on the tiles
- The game is laggy

Overall the design has some structure that allows for the extension of functionality, but the prototype will need to be refactored to fix its issues. The game's performance will also need to be improved.

1.2 Sprint 3 solution direction

In this section, we go through our plan for sprint 3.

1.2.1 Chosen prototype and justification

For sprint 3, we have decided to proceed with one person's solution: Shen's.

The main reasons are as follows:

- Far more flexible than all the others as it was the only one that was built from the ground up not relying on hard coding.
- Effective use of abstractions, where the abstractions contain lots of behaviour that can be reused by new modules. This is unlike the other prototypes which do not utilise abstractions to reduce repetition to the same degree.
- Fully follows the dependency inversion principle where concrete classes and abstractions depend solely on abstractions (i.e abstract classes and interfaces), decoupling classes and making it much easier to modify concrete implementations & functionalities. All the other prototypes depended on concrete classes within concrete classes and as a result, had a high degree of coupling.
- Fully follows the single responsibility principle and separates concerns effectively using numerous modules, reducing the complexity when modifying or extending the functionality of the game. All the other prototypes except one contained god classes.
- Follows the open/ended principle (due to the game's modular nature), liskov's substitution principle (through the use of abstract classes and interfaces), and the interface segregation principle (no classes are implementing methods they do not use).
- Use of interfaces to enforce contracts, separate concerns and reduce cognitive load. For example, any implementers of the DrawableByAsset interface can be drawn onto the pygame screen. All the others except one do not use interfaces.
- Well-written documentation for every single method and class, and type hints for everything, making it easier to learn, use and understand the code base. All the other prototypes lack type hinting.
- Code encapsulates modules by using visibility modifiers (python's double and single underscore prefixes), unlike the other prototypes.

Overall, these reasons listed make the selected prototype far superior compared to the other prototypes. Its robust structure and effective documentation will make it much easier to add new functionality for future sprints.

1.2.2 Ideas/elements to be taken from each prototype

All ideas from the other prototypes are to be disregarded except for the ideas present in Shen's prototype due to its vastly superior design as justified in section 1.2.1.

1.2.3 New ideas/elements to improve the chosen prototype

This section describes the ideas/elements that will be used to improve and fix the problems that the chosen prototype has.

Problem 1: Missing dragon pirate chit cards
Solution: Introduce a new module that will extend from the chit card abstract class to reduce repetition

Problem 2: Randomisation of tiles making up the volcano cards
Solution: Introduce a new method that will provide a preconfigured tile sequence to match the sequences seen in the specific volcano cards that are available in the Fiery Dragons game

2 CRC Cards

2.0 Chosen Classes

In this section, we provide Class-Responsibility-Collaboration (CRC) cards for six of the main classes in our design. These classes are the Tile, ChitCard, GameBoard, PlayableCharacter, GameWorld and PygameScreenController.

2.0.1 CRC Cards Diagram

2.1 Description of CRC Cards

In this section, we provide a brief description of the purpose of each chosen class specified in section 2.0.

2.1.1 Tile

The primary role of the ‘Tile’ is to represent an occupiable space within the game. It manages its own rendering on the game board by utilising the information contained in the ‘DrawProperties’ class and is also responsible for retrieving any animal or players (‘PlayableCharacter’) located on it, which is essential for tasks like comparing the animal on a chit card with the one on the tile when a player flips a chit card or when determining if a tile is already occupied.

Moreover, the ‘Tile’ class handles placing players on it and any associated effects when a player is placed on it. For instance, the ‘CaveTile’ notifies the game of a winning player when it becomes occupied.

Additionally, the ‘Tile’ itself provides the instructions (‘DrawAssetInstruction’) on how to draw itself when its drawing instructions are requested by the ‘GameBoard’. This guarantees a unified presentation across numerous tiles on the game board, accommodating potential variations in animals and characters placed upon them.

One alternative considered was to shift most of these responsibilities to the ‘GameBoard’ class. In this alternative, ‘Tile’ would only manage its own drawing on the board, while the ‘GameBoard’ would take over tasks such as storing information about the player and animal enums on each tile, executing effects triggered by player movement, and obtaining drawing instructions.

This idea was ultimately rejected because it would result in an overly simplistic ‘Tile’ class and burden the ‘GameBoard’ with excessive complexity. Such a design shift would diminish the modularity and adaptability of the system because the 'GameBoard' would absorb responsibilities originally designated for the 'Tile' class. Moreover, any adjustments to tile functionality would require developers to navigate through the 'GameBoard' class, complicating maintenance and extension efforts. By concentrating too many responsibilities in the ‘GameBoard’, the design would become harder to maintain and extend.

2.1.2 Chit Card

The 'ChitCard' class is responsible for managing the state and behaviour of individual chit cards. Its primary responsibilities include setting and tracking the 'flipped' state of a chit card to determine if it has been turned over, defining and managing the drawing properties for how the chit card should be visually represented, and establishing a connection with the game board delegate to inform about events related to the chit cards (such as clicking).

Additionally, the 'ChitCard' class is responsible for providing methods to retrieve information about the chit card, the number of chit symbols on the card, and whether the card has been flipped. This allows other components in the game, such as the 'ModularClickableSprite’ for

making the card clickable and 'GameBoard' for handling the game board events such as the movement of characters upon flipping a chit card successfully through the method ‘on_move_action_fired()’, to interact with the 'ChitCard' object effectively. Finally, the chit card works with 'DrawProperties' to allow specific implementations of chit cards to access their visual appearances such as their size and positions through the class’s methods ‘get_size()’ and ‘get_coordinates()’ respectively.

Instead of managing the flipped state internally, a separate ‘CardStateManager’ class could have been created to manage the state of the chit cards. This would have decoupled state management from the ‘ChitCard’ class. However, this approach might introduce unnecessary complexity for managing the state of the chit cards, as each individual chit card would need to interact with the ‘CardStateManager’ to update its state. This could lead to decreased cohesion and increased coupling between the ‘ChitCard’ class and ‘CardStateManager’ class. Additionally, this approach might be too abstract, as each individual chit card would not be able to specify when and how it wants to respond to a click or other events. This could limit the flexibility of the chit cards and make it harder to implement specific behaviours of different types of chit cards.

Another alternative could have been to have a separate ‘EventDispatcher’ class responsible for managing event dispatching for chit cards. However, if the event dispatcher is not well integrated with the game board delegate, it might also introduce unnecessary complexity and make the game prone to technical errors. For example, if the event dispatcher is not properly synchronised with the game board delegate, it could lead to race conditions or inconsistent game state. Additionally, if the event dispatcher is too tightly coupled with the game board delegate, it could lead to a fragmented design where the responsibilities of each class are not clearly defined.

These alternative distributions of responsibilities were rejected because the focus was keeping the ‘ChitCard’ class dedicated to managing the state and behaviour of individual chit cards. By centralising responsibilities such as state management, drawing properties, and event handling within the ‘ChitCard’ class, it allows for a more simplified design while avoiding unwanted complexity.

2.1.3 Game Board

The 'GameBoard' class encapsulates the central logic and state management for the game board. It is responsible for managing the movement of a character by a specified number of steps and providing access to the tile on which a character is currently standing. It also facilitates the rendering of the game board by providing the clickable assets and drawable assets to draw when requested. The 'GameBoard' class also knows crucial information about the game state, including when a player's turn ends, the currently active player, the chit cards present and the tiles on the board. For example, when a player’s turn ends, the ‘GameBoard’ class can use this information to trigger events such as resetting chit cards.

To fulfil its responsibilities, such as moving a character, it must have access to that character class 'PlayableCharacter'. The 'GameBoard' class returns objects that conform to the 'DrawableByAsset' interface and objects of the 'DrawAssetInstruction' class which is required to draw the game elements on the game board. The 'GameBoard' class also retrieves the tiles based on the character's position. Hence, its collaboration with the 'Tile' class. The

‘ModularClickableSprite’ interface enables assets to be clickable, allowing objects like chit cards to respond to clicks. Consequently, when a chit card is clicked, it flips, and the ‘GameBoard’ moves the character accordingly if the move is valid. The ‘GameBoard’ class interacts with the ‘ModularClickableSprite’ interface to enable this functionality.

Instead of the ‘GameBoard’ class handling movement logic directly, a separate ‘MovementManager’ class could have been created to handle character movement. However, this could lead to increased complexity and potential data synchronisation issues between the ‘MovementManager’ and ‘GameBoard’ classes, as movement is closely tied to the game board’s state and tiles.

Another approach could have been to distribute state management responsibilities across multiple classes instead of centralising them in the ‘GameBoard’ class. However, this could lead to a fragmented design, making it harder to maintain and reason about the game’s state.

The rejection of these alternatives stems from the desire to keep the ‘GameBoard’ class cohesive and responsible for the core game board functionalities. Centralising these responsibilities in a single class can simplify the design, improve readability and make it easier to maintain and extend the game board logic in Sprint 4.

2.1.4 Playable Character

The 'PlayableCharacter' class represents a character that can be controlled and interacted with in a game. It manages the drawing properties, including size, coordinates, and rotation (which is used to highlight an active player) to ensure that the character can be rendered visually on the screen. It also determines whether the character should continue its turn for the next tick of the game loop.

The 'PlayableCharacter' class interacts with several collaborators to fulfil its responsibilities. It uses a 'PlayableCharacterVariant' to represent different variants of playable characters within the game. The variants differ by colour to differentiate between the playable characters on the game board. It also relies on 'DrawProperties' to manage the drawing properties of the character and 'DrawableByAsset' to allow for the retrieval of the drawing instructions for the playable character.

Instead of managing drawing properties directly, a separate ‘RenderingManager’ class could have been created to handle all rendering-related data for the classes. This would have decoupled rendering logic from the ‘PlayableCharacter’ classes, potentially making it easier to manage different rendering aspects of the game. However, this approach might introduce unnecessary complexity due to the ungodly number of dependencies between the ‘RenderingManager’ class and all concrete classes. Managing rendering properties specific to each character could become more complicated as a result of this increased dependency complexity.

The second alternative could have been to have a separate ‘TurnManager’ class for determining whether a player should continue its turn rather than managing turn logic within the ‘PlayableCharacter’ class. This would have separated concerns related to turn management from the ‘PlayableCharacter’ class, but it might also lead to increased coupling between the

two classes, as all the required states to determine the next turn is in ‘PlayableCharacter’, potentially decreasing cohesion.

The rejection of these alternatives again stems from the desire to keep the ‘PlayableCharacter’ class focused on representing individual characters in the game and maintaining its cohesion. Centralising responsibilities related to rendering and turn management within the ‘PlayableCharacter’ class simplifies the design and makes it easier to understand and maintain.

2.1.5 Game World

The primary responsibility of the ‘GameWorld’ class is to manage the game's flow. It begins by initialising the game's user interface through calls to ‘PygameScreenController’ and ‘GameBoard’ classes. Continuously, it monitors the player's actions, ensuring the game closes correctly if the player chooses to exit. Additionally, it tracks interactions with all clickable objects like chitcards, updating their states when clicked.

Within its loop, the ‘GameWorld’ class also manages player’s turns. If a player should end their turn, then it processes the turn transition to the next player. Moreover, the ‘GameWorld’ class implements the WinEventListener interface so it can be notified when a winning player appears. When notified of a winning player, it displays the winning message and disables further clicks to prevent further actions. The GameWorld then quits the game after a small delay. GameWorld also can enable/disable user interactions with the game via mouse clicks, which is used to prevent user actions from overlapping with another player’s turn once a player’s turn ends.

Another alternative proposed creating a separate ‘GameEndManager’ class responsible for game-ending logic, such as stopping the game and displaying the win message. This was rejected because it introduced unnecessary complexity into the system. The ‘GameWorld’ already has a complete context of the current game application state and hence is the most appropriate place to handle game-ending responsibilities. Introducing an additional class for this purpose would fragment the logic, lowering cohesion and complicating the overall design.

2.1.6 Pygame Screen Controller

The ‘PygameScreenController’ class is a facade that simplifies the process of drawing assets on the screen for other classes by abstracting away the complex interactions with the external drawing library Pygame.

To enhance system performance and prevent redundant loading, it employs dictionaries to cache loaded images. Additionally, it is responsible for filling the game’s background colour, especially at the game's outset.

This class handles drawing specific assets by simply inputting the image path, size (width & height), and rotation degrees. Instead of manually drawing each image,multiple images can be automatically drawn via the use of a function that takes in a list of ‘DrawAssetInstruction’ objects (i.e a list of instructions). It also facilitates the drawing of clickable assets by taking a list of ‘ModularClickableSprite’ objects. This functionality is crucial in the ‘GameWorld’ class, where it's necessary to store both the rectangle (hitbox) and the clickable object itself to manage mouse click events and the object’s response to them. Additionally, the class provides

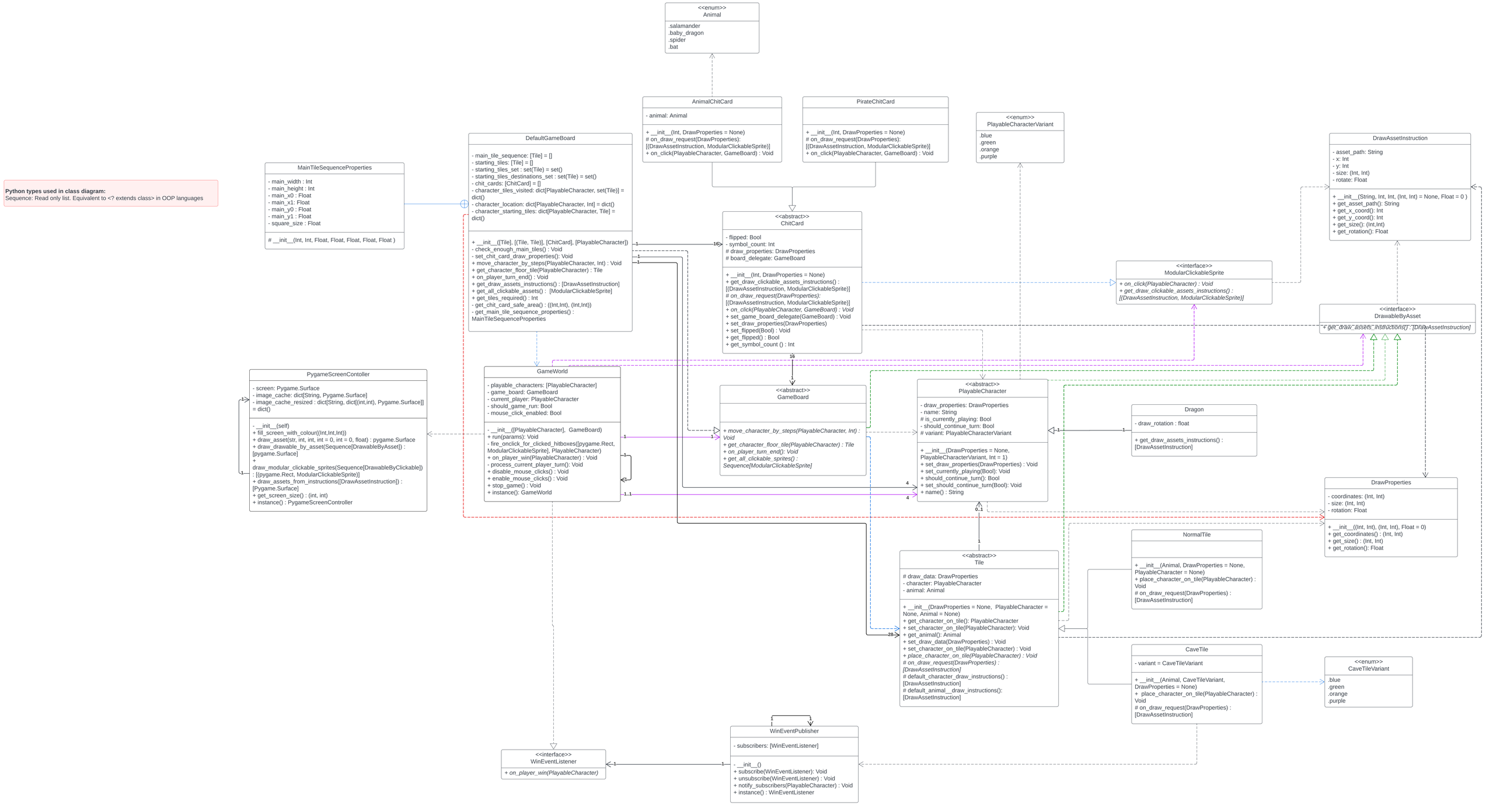
a method to retrieve the screen size, aiding in determining the tile size based on the screen dimensions.

An alternative distribution of responsibilities for the ‘PygameScreenController’ could have been to separate the caching of loaded images into a separate ‘ImageCache’ class. This class would be responsible for managing the loading and caching of images, while the ‘PygameScreenController’ would focus solely on drawing assets on the screen such as the game board, dragon tokens, chit cards, etc.

However, this alternative was rejected because it would again introduce unnecessary complexity and reduce the cohesion of the ‘PygameScreenController’ class. The class remains focused on its primary responsibility of managing the screen and drawing assets. Additionally, having the caching logic directly within the ‘PygameScreenController’ class allows for easier management of cached images and ensures that they are only loaded once, improving performance.

3 Class diagram

The class diagram is attached directly below this page.


Python types used in class diagram:
Sequence: Read only list. Equivalent to <? extends class> in OOP languages
<<enum>>
Animal
.salamander
.baby_dragon
.spider
.bat
AnimalChitCard
- animal: Animal
+ __init__(Int, DrawProperties = None)
# on_draw_request(DrawProperties): [(DrawAssetInstruction, ModularClickableSprite)]
+ on_click(PlayableCharacter, GameBoard) : Void
PirateChitCard
+ __init__(Int, DrawProperties = None)
# on_draw_request(DrawProperties): [(DrawAssetInstruction, ModularClickableSprite)]
+ on_click(PlayableCharacter, GameBoard) : Void
<<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
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)
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()
+ __init__([Tile], [(Tile, Tile)], [ChitCard], [PlayableCharacter])
- check_enough_main_tiles() : Void
- set_chit_card_draw_properties(): Void
+ move_character_by_steps(PlayableCharacter, Int) : Void
+ get_character_floor_tile(PlayableCharacter) : Tile
+ on_player_turn_end() : Void
+ get_draw_assets_instructions() : [DrawAssetInstruction]
+ get_all_clickable_assets() : [ModularClickableSprite]
+ get_tiles_required() : Int
- get_chit_card_safe_area() : ((Int,Int), (Int,Int))
- get_main_tile_sequence_properties() : MainTileSequenceProperties
<<abstract>>
ChitCard
- flipped: Bool
- symbol_count: Int
draw_properties: DrawProperties
board_delegate: GameBoard
+ __init__(Int, DrawProperties = None)
+ get_draw_clickable_assets_instructions() : [(DrawAssetInstruction, ModularClickableSprite)]
 # on_draw_request(DrawProperties):
[(DrawAssetInstruction, ModularClickableSprite)]
+ on_click(PlayableCharacter, GameBoard) : Void
+ set_game_board_delegate(GameBoard) : Void
+ set_draw_properties(DrawProperties)
+ set_flipped(Bool) : Void
+ get_flipped() : Bool
+ get_symbol_count () : Int
<<interface>>
ModularClickableSprite
+ on_click(PlayableCharacter) : Void
+ get_draw_clickable_assets_instructions() : [(DrawAssetInstruction, ModularClickableSprite)]
<<interface>>
DrawableByAsset
+ get_draw_assets_instructions() : [DrawAssetInstruction]
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[DrawableByClickable]) : [(pygame.Rect, ModularClickableSprite)]
+ draw_assets_from_instructions([DrawAssetInstruction]) : [Pygame.Surface]
+ get_screen_size() : (int, int)
+ instance() : PygameScreenController
GameWorld
- playable_characters: [PlayableCharacter]
- game_board: GameBoard
- current_player: PlayableCharacter
- should_game_run: Bool
- mouse_click_enabled: Bool
- __init__([PlayableCharacter], GameBoard)
+ run(params): Void
- 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
<<abstract>>
GameBoard
+ move_character_by_steps(PlayableCharacter, Int) : Void
+ get_character_floor_tile(PlayableCharacter) : Tile
+ on_player_turn_end(): Void
+ get_all_clickable_sprites() : Sequence[ModularClickableSprite]
<<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
Dragon
- draw_rotation : float
+ get_draw_assets_instructions() : [DrawAssetInstruction]
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
<<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(): Animal
+ set_draw_data(DrawProperties) : Void
+ set_character_on_tile(PlayableCharacter) : Void
+ place_character_on_tile(PlayableCharacter) : Void
 # on_draw_request(DrawProperties) : [DrawAssetInstruction]
default_character_draw_instructions() : [DrawAssetInstruction]
default_animal__draw_instructions(): [DrawAssetInstruction]
NormalTile
+ __init__(Animal, DrawProperties = None, PlayableCharacter = None)
+ place_character_on_tile(PlayableCharacter) : Void
# on_draw_request(DrawProperties) : [DrawAssetInstruction]
CaveTile
- variant = CaveTileVariant
+ __init__(Animal, CaveTileVariant, DrawProperties = None)
+ place_character_on_tile(PlayableCharacter) : Void
# on_draw_request(DrawProperties) : [DrawAssetInstruction]
<<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 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.

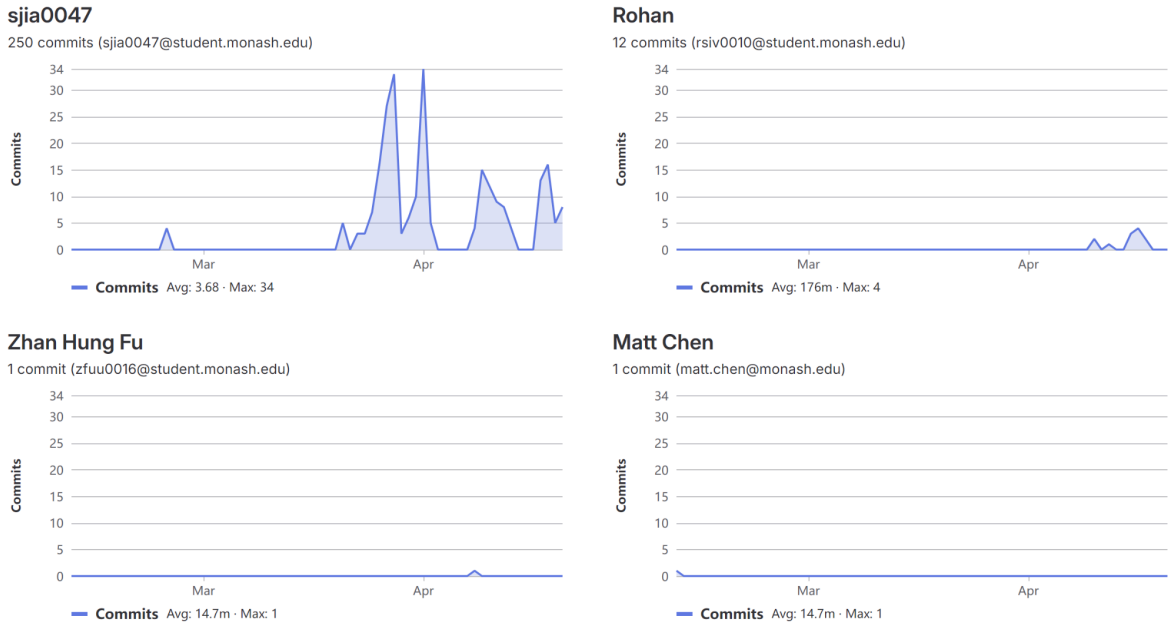
Steps:

1. Open up the source code folder (called Sprint3_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 Sprint3_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.

5 Contributor analytics

These contributor analytics do not reflect the true distribution of work. Team members were split amongst multiple different deliverables, and most of the deliverables were not tracked through git.

I did not have any issues with the workload distribution - sjia0047



6 References

https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?start=5
https://agilemodeling.com/artifacts/crcmodel.htm