Sprint 4

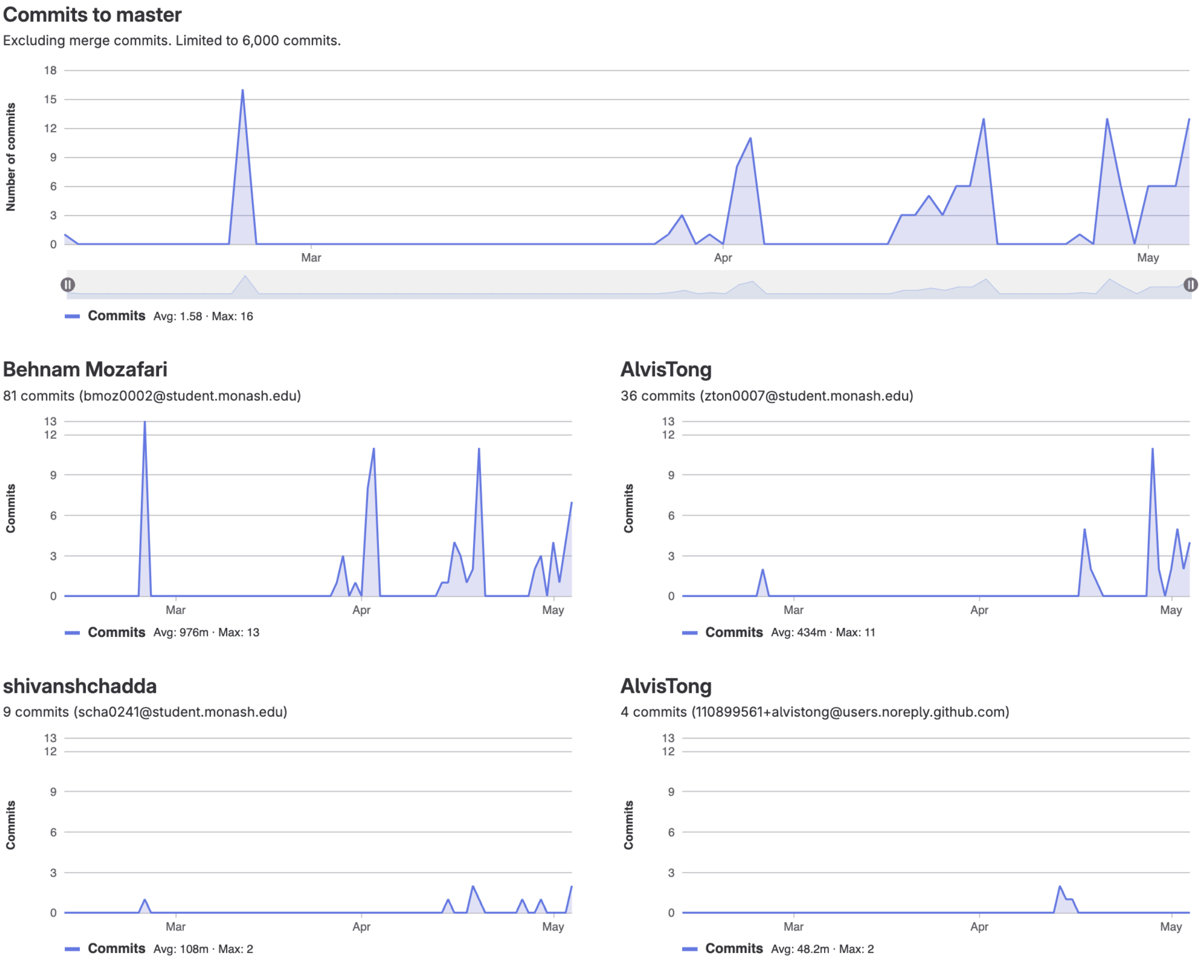
Inferno Innovations

Behnam Mozafari, Shivansh Chadda, Alvis Tong

Assumptions

- In order to ensure the game is competitive and not exploitable, we assumed that the players would not be able to use the pirate dragon card to move back behind their cave to then win, so the 1 and 2 pirate dragon cards only have an affect and are playable from 2 and 3 squares after the cave respectively.
- In order to keep the essence of the swap card, we have assumed that players can swap with another player anywhere on the board (not in a cave) and then be able to win, irrespective of the distance to the cave after swapping.

Contributor Analytics



Commits to master

Excluding merge commits. Limited to 6,000 commits.

Commits Avg: 1.58 · Max: 16

Behnam Mozafari

81 commits (bmoz0002@student.monash.edu)

Commits Avg: 976m · Max: 13

AlvisTong

36 commits (zton0007@student.monash.edu)

Commits Avg: 434m · Max: 11

shivanshchadda

9 commits (scha0241@student.monash.edu)

Commits Avg: 108m · Max: 2

AlvisTong

4 commits (110899561+alvistong@users.noreply.github.com)

Commits Avg: 48.2m · Max: 2

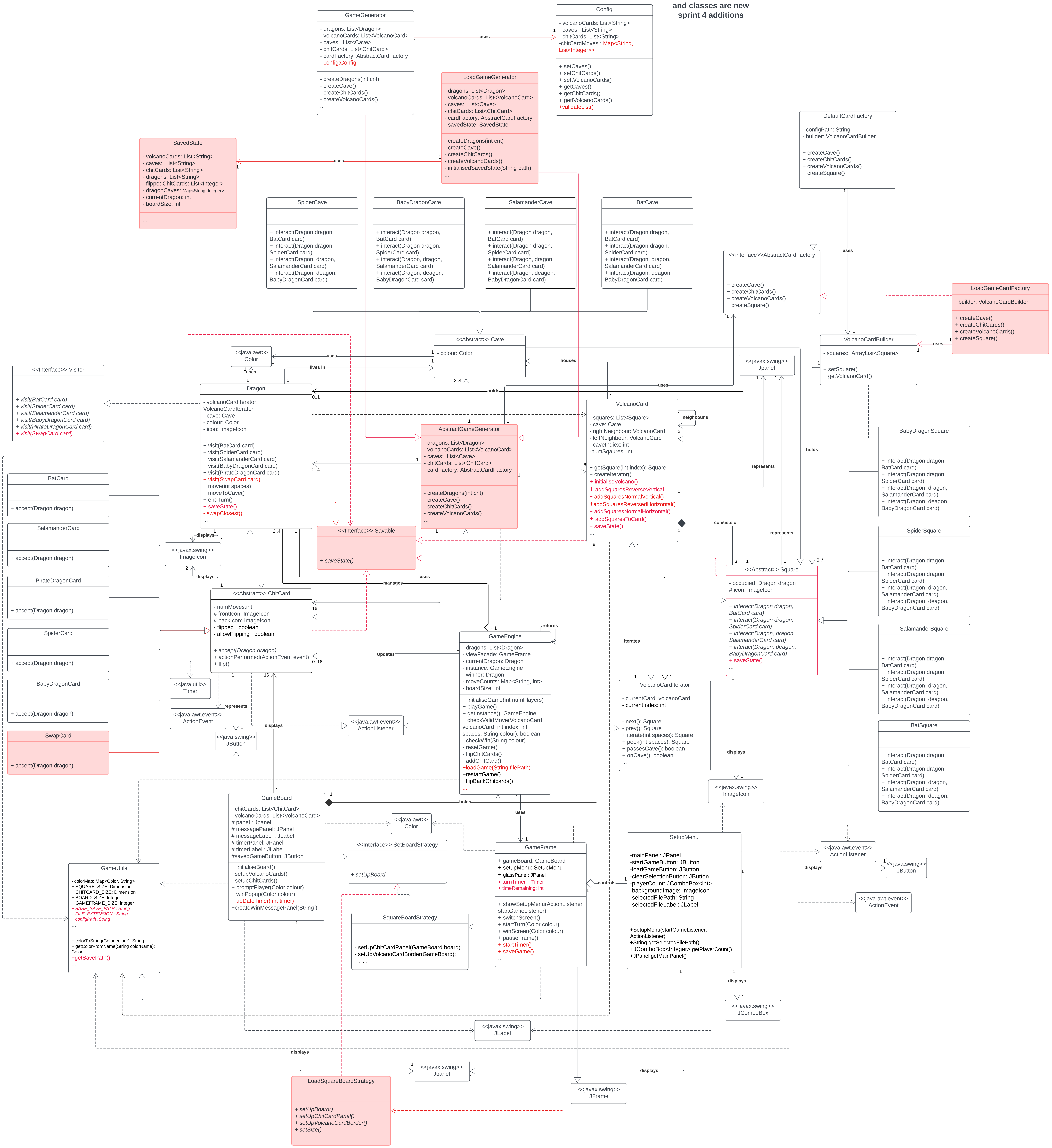
Self-Defined Extension

As a self-defined extension, the Timer feature limits playtime in Fiery dragon games, aligning with the human value which promotes healthy gaming habits. By preventing excessive screen time, the timer reduces the risk of digital eye strain, which can cause symptoms like dry eyes, headaches, and blurred vision. Besides addressing the human value, incorporating a timer makes the game more challenging and engaging. The pressure of the ticking clock can make even simple decisions more challenging, increasing the difficulty level and testing player's memory. Timed challenges can vary from game to game and enhance replayability. Players may return to improve their performance under time constraints, striving for better efficiency and quicker decision-making. This balance of structured gameplay and added difficulty can make the game more enjoyable and rewarding for players of all skill levels.

The second extension is the new 'Swap' Chit Card. It allows players to swap their dragons with the closest dragons. By incorporating this amusing functionality, players are prompted to think and make decisions strategically. When a swap card is uncovered, players are compelled to adapt their strategies to win the game. Players must not only plan ahead but also be prepared for unexpected shifts, making the game both mentally stimulating and entertaining. The unpredictability and possibility of the swap chit card enhance the game's dynamic and keep the gameplay exciting and challenging.

Sprint 4 Class Diagram

Note: Red coloured lines, text and classes are new sprint 4 additions


GameGenerator
- dragons: List<Dragon>
- volcanoCards: List<VolcanoCard>
- caves: List<Cave>
- chitCards: List<ChitCard>
- cardFactory: AbstractCardFactory
- config:Config
- createDragons(int cnt)
- createCave()
- createChitCards()
- createVolcanoCards()
...
uses
Config
- volcanoCards: List<String>
- caves: List<String>
- chitCards: List<String>
-chitCardMoves : Map<String, List<Integer>>
+ setCaves()
+ setChitCards()
+ settVolcanoCards()
+ getCaves()
+ getChitCards()
+ gettVolcanoCards()
+validateList()
LoadGameGenerator
- dragons: List<Dragon>
- volcanoCards: List<VolcanoCard>
- caves: List<Cave>
- chitCards: List<ChitCard>
- cardFactory: AbstractCardFactory
- savedState: SavedState
- createDragons(int cnt)
- createCave()
- createChitCards()
- createVolcanoCards()
- initialiseSavedState(String path)
...
uses
SavedState
- volcanoCards: List<String>
- caves: List<String>
- chitCards: List<String>
- dragons: List<String>
- flippedChitCards: List<Integer>
- dragonCaves: Map<String, Integer>
- currentDragon: int
- boardSize: int
...
DefaultCardFactory
- configPath: String
- builder: VolcanoCardBuilder
+ createCave()
+ createChitCards()
+ createVolcanoCards()
+ createSquare()
SpiderCave
+ interact(Dragon dragon, BatCard card)
+ interact(Dragon dragon, SpiderCard card)
+ interact(Dragon, dragon, SalamanderCard card)
+ interact(Dragon, deagon, BabyDragonCard card)
BabyDragonCave
+ interact(Dragon dragon, BatCard card)
+ interact(Dragon dragon, SpiderCard card)
+ interact(Dragon, dragon, SalamanderCard card)
+ interact(Dragon, deagon, BabyDragonCard card)
SalamanderCave
+ interact(Dragon dragon, BatCard card)
+ interact(Dragon dragon, SpiderCard card)
+ interact(Dragon, dragon, SalamanderCard card)
+ interact(Dragon, deagon, BabyDragonCard card)
BatCave
+ interact(Dragon dragon, BatCard card)
+ interact(Dragon dragon, SpiderCard card)
+ interact(Dragon, dragon, SalamanderCard card)
+ interact(Dragon, deagon, BabyDragonCard card)
<<interface>>AbstractCardFactory
+ createCave()
+ createChitCards()
+ createVolcanoCards()
+ createSquare()
uses
LoadGameCardFactory
- builder: VolcanoCardBuilder
+ createCave()
+ createChitCards()
+ createVolcanoCards()
+ createSquare()
uses
VolcanoCardBuilder
- squares: ArrayList<Square>
+ setSquare()
+ getVolcanoCard()
<<Abstract>> Cave
- colour: Color
...
<<java.awt>> Color
uses
lives in
houses
uses
<<javax.swing>> Jpanel
<<Interface>> Visitor
+ visit(BatCard card)
+ visit(SpiderCard card)
+ visit(SalamanderCard card)
+ visit(BabyDragonCard card)
+ visit(PirateDragonCard card)
+ visit(SwapCard card)
Dragon
- volcanoCardIterator: VolcanoCardIterator
- cave: Cave
- colour: Color
- icon: ImageIcon
+ visit(BatCard card)
+ visit(SpiderCard card)
+ visit(SalamanderCard card)
+ visit(BabyDragonCard card)
+ visit(PirateDragonCard card)
+ visit(SwapCard card)
+ move(int spaces)
+ moveToCave()
+ endTurn()
+ saveState()
+ swapClosest()
...
holds
AbstractGameGenerator
- dragons: List<Dragon>
- volcanoCards: List<VolcanoCard>
- caves: List<Cave>
- chitCards: List<ChitCard>
- cardFactory: AbstractCardFactory
- createDragons(int cnt)
- createCave()
- createChitCards()
- createVolcanoCards()
...
VolcanoCard
- squares: List<Square>
- cave: Cave
- rightNeighbour: VolcanoCard
- leftNeighbour: VolcanoCard
- caveIndex: int
-numSqaures: int
+ getSquare(int index): Square
+ createIterator()
+ initialiseVolcano()
+ addSquaresReverseVertical
+ addSquaresNormalVertical()
+addSquaresReversedHorizontal()
+ addSquaresNormalHorizontal()
+ addSquaresToCard()
+ saveState()
...
neighbour's
represents
holds
BabyDragonSquare
+ interact(Dragon dragon, BatCard card)
+ interact(Dragon dragon, SpiderCard card)
+ interact(Dragon, dragon, SalamanderCard card)
+ interact(Dragon, deagon, BabyDragonCard card)
BatCard
+ accept(Dragon dragon)
SalamanderCard
+ accept(Dragon dragon)
PirateDragonCard
+ accept(Dragon dragon)
SpiderCard
+ accept(Dragon dragon)
BabyDragonCard
+ accept(Dragon dragon)
SwapCard
+ accept(Dragon dragon)
displays
<<javax.swing>> ImageIcon
displays
<<Interface>> Savable
+ saveState()
consists of
represents
<<Abstract>> Square
- occupied: Dragon dragon
# icon: ImageIcon
+ interact(Dragon dragon, BatCard card)
+ interact(Dragon dragon, SpiderCard card)
+ interact(Dragon, dragon, SalamanderCard card)
+ interact(Dragon, deagon, BabyDragonCard card)
+ saveState()
...
SpiderSquare
+ interact(Dragon dragon, BatCard card)
+ interact(Dragon dragon, SpiderCard card)
+ interact(Dragon, dragon, SalamanderCard card)
+ interact(Dragon, deagon, BabyDragonCard card)
SalamanderSquare
+ interact(Dragon dragon, BatCard card)
+ interact(Dragon dragon, SpiderCard card)
+ interact(Dragon, dragon, SalamanderCard card)
+ interact(Dragon, deagon, BabyDragonCard card)
BatSquare
+ interact(Dragon dragon, BatCard card)
+ interact(Dragon dragon, SpiderCard card)
+ interact(Dragon, dragon, SalamanderCard card)
+ interact(Dragon, deagon, BabyDragonCard card)
manages
uses
<<Abstract>> ChitCard
- numMoves:int
# frontIcon: ImageIcon
# backIcon: ImageIcon
- flipped : boolean
- allowFlipping : boolean
+ accept(Dragon dragon)
+ actionPerformed(ActionEvent event)
+ flip()
returns
GameEngine
- dragons: List<Dragon>
- viewFacade: GameFrame
- currentDragon: Dragon
- instance: GameEngine
- winner: Dragon
- moveCounts: Map<String, int>
- boardSize: int
+ initialiseGame(int numPlayers)
+ playGame()
+ getInstance(): GameEngine
+ checkValidMove(VolcanoCard volcanoCard, int index, int spaces, boolean): boolean
- checkWin(String colour)
- resetGame()
- flipChitCards()
- addChitCard()
+loadGame(String filePath)
+restartGame()
+flipBackChitcards()
...
Updates
iterates
VolcanoCardIterator
- currentCard: volcanoCard
- currentIndex: int
- next(): Square
- prev(): Square
+ iterate(int spaces): Square
+ peek(int spaces): Square
+ passesCave(): boolean
+ onCave(): boolean
...
<<java.util>> Timer
<<java.awt.event>> ActionEvent
represents
displays
<<java.awt.event>> ActionListener
<<java.swing>> JButton
displays
<<javax.swing>> ImageIcon
holds
GameBoard
- chitCards: List<ChitCard>
- volcanoCards: List<VolcanoCard>
# panel : Jpanel
# messagePanel: JPanel
# messageLabel : JLabel
# timerPanel: JPanel
# timerLabel : JLabel
#savedGameButton: JButton
+ initialiseBoard()
- setupVolcanoCards()
- setupChitCards()
+ promptPlayer(Color colour)
+ winPopup(Color colour)
+ upDateTimer(int timer)
+createWinMessagePanel(String)
...
<<java.awt>> Color
<<Interface>> SetBoardStrategy
+ setUpBoard
uses
GameFrame
+ gameBoard: GameBoard
+ setupMenu: SetupMenu
+ glassPane : JPanel
+ turnTimer : Timer
+ timeRemaining: int
+ showSetupMenu(ActionListener startGameListener)
+ switchScreen()
+ startTurn(Color colour)
+ winScreen(Color colour)
+ pauseFrame()
+ startTimer()
+ saveGame()
...
controls
SetupMenu
-mainPanel: JPanel
-startGameButton: JButton
-loadGameButton: JButton
-clearSelectionButton: JButton
-playerCount: JComboBox<int>
-backgroundImage: ImageIcon
-selectedFilePath: String
-selectedFileLabel: JLabel
+SetupMenu(startGameListener: ActionListener)
+String getSelectedFilePath()
+JComboBox<Integer> getPlayerCount()
+JPanel getMainPanel()
<<java.awt.event>> ActionListener
displays
<<java.swing>> JButton
<<java.awt.event>> ActionEvent
GameUtils
- colorMap: Map<Color, String>
+ SQUARE_SIZE: Dimension
+ CHITCARD_SIZE: Dimension
+ BOARD_SIZE: Integer
+ GAMEFRAME_SIZE: integer
+ BASE_SAVE_PATH : String
+ FILE_EXTENSION : String
+ configPath :String
...
+ colorToString(Color colour): String
+ getColorFromName(String colorName): Color
+getSavePath()
...
SquareBoardStrategy
- setUpChitCardPanel(GameBoard board)
- setUpVolcanoCardBorder(GameBoard);
. . .
displays
<<javax.swing>> JComboBox
<<javax.swing>> JLabel
displays
<<javax.swing>> Jpanel
displays
<<javax.swing>> JFrame
LoadSquareBoardStrategy
+ setUpBoard()
+ setUpChitCardPanel()
+ setUpVolcanoCardBorder()
+ setSize()
...

Reflection

Timer Feature:
Incorporating the timer feature was moderately challenging. The use of design patterns, like the Strategy pattern for board setup, made this task easier. The clear separation between the UI logic (handled by GameFrame and GameBoard) and the game logic (handled by GameEngine) also helped. However, some tight coupling between classes made it a bit harder to update the timer throughout the system. If I were to redo this, I would focus on a more modular design with clear interfaces for game states and UI updates to make such integrations smoother.

Load and Save Functionality:
Incorporating the load and save functionality was relatively straightforward, thanks to our design approach. Implementing the Savable interface standardised the method by which each game entity is saved, and the SavedState class provided methods for serialising and deserialising game objects to and from JSON format. This ensured a consistent approach to data persistence across various game components. The use of the Savable interface allowed any class to be easily serialised to JSON within the SavedState, simplifying the process of loading and saving the game.

The Abstract Factory pattern played a significant role in creating game components (such as chit cards, volcanoes, and caves) without needing to specify the exact classes. This design pattern decoupled code by separating object creation from usage, making the codebase more modular and easier to manage. It also simplified the integration of new game variants or components, reducing dependencies and enhancing extensibility.

The AbstractGameGenerator class centralised the initialisation and configuration of the game by utilising the provided factory and loading state functionalities. It used the abstract factory to create and set up new game components, read saved game states from JSON, and initialised the components accordingly. This approach clearly separated game setup logic, improving maintainability and extensibility, and effectively handled both the creation of new games and the loading of saved states.

New ‘Swap’ Chit Card:
Incorporating the new Dragon Card extension into the Sprint 3 design and implementation was relatively straightforward due to the pre-existing abstract ChitCard class, which facilitated extension. The use of the iterator pattern with the volcano card iterators to move and track the position of each dragon proved highly beneficial, greatly simplifying the swapping of positions by merely requiring us to switch the dragons’ volcano card iterators. Additionally, the visitor pattern with double dispatch made the interaction with ChitCards easy to manage. However, recalculating the winner presented a challenge since the previous method based on the number of moves was incompatible with the new position-switching mechanic. This necessitated updating the system to check if a player was at a cave of the same colour to determine a win. If I could revisit Sprint 3, I would incorporate more flexible win-checking mechanisms to accommodate

such future extensions more seamlessly. This experience underscores the importance of anticipating potential future changes and designing systems with extensibility in mind, leveraging design patterns and modular code to ease future integrations.

Overall Project Reflection:
The clear distribution of responsibilities among GameFrame, GameBoard, and GameEngine was beneficial. It allowed us to add features like the timer without major refactoring. However, further reducing coupling in some areas could have made extensions even smoother.

The codebase was mostly clean, but some areas had higher coupling than ideal. Reducing these dependencies and adhering more strictly to the Single Responsibility Principle (SRP) would have facilitated smoother integration of new features.

Using design patterns like Abstract Factory, Strategy, Visitor, Iterator, as well as Double Dispatch greatly assisted in adding new functionalities. These patterns and design techniques allowed us to extend the system's capabilities without modifying existing code, adhering to the Open/Closed Principle (OCP).

Reflecting on these experiences, if I were to start over with Sprint 3, we would emphasise a more modular design from the outset, focusing on clear interfaces and reducing class dependencies. This approach would ensure that the system remains flexible and easily extendable, accommodating future changes and new features with minimal disruption. By leveraging design patterns and adhering to best practices in software design, we can build a more robust and maintainable codebase that stands the test of time. *we would try plan for possible extensions to facilitate them in the design*

Description of source code and executable

Project Structure

The project is structured to ensure clarity and ease of navigation for developers and users interested in exploring the source code and resources.

`Docs/` Directory

contains all documentation

`src/` Directory

This directory contains all the files organised into subdirectories reflecting different components
of the game:

`game/`: Contains all Java classes organised further into packages:

- `entities/`: Includes Java classes defining different game entities like Dragon, Player, etc.
- `engine/`: Contains classes related to the game mechanics and logic, such as GameEngine.
- `tiles/`: Houses classes that represent different types of tiles in the game, such as VolcanoCard.
- `utils/`: Includes classes for general utility functions for the game
- `view/`: Includes classes for the UI and handling of the game board and setup menu.
- `chitcards/`: Contains classes for the different types of cards used within the game.

`resources/`: This directory includes all non-Java files needed by the game, organised by type:

- `images/`: Contains all image files used in the game, such as icons for dragons, background /images for game boards, etc.

`configFiles/`: Contains all config json files for the game setup.

Running the Game

To run the game, navigate to the src directory and compile the Java files using your
preferred development environment

Ensure that you have the necessary Java JDK installed on your system, it may have to be JDK 22.0.1.

The Jar file was tested on an M1 mac using Intellij.

License

Images

all images were made by Alvis Tong with resources from
https://boardgamegeek.com/boardgame/23658/fiery-dragons