Design Rationale

Design Patterns

Command

The command pattern was used for handling actions in the game. It decouples the object that invokes the operation from the object that performs it i.e. the code initiating the action doesn't have to directly interact with the code performing the action. This separation enhances flexibility and maintainability of the game by reducing dependencies between components and makes it easier to extend individual actions without affecting others. Actions can be reused by different senders or multiple senders can call the same command.

There are two categories of actions, dragon card actions and game actions that are in response to events. Dragon card actions are commands that store actions that dragon cards can perform, by having the actions as commands this ensures that actions can be reused by multiple dragon cards and so follows the dry principle. Game actions are commands that subscribe to various game events and can be used to create the game flow. These actions have different purposes but use the same method so it was decided to use the one interface for both. There are contextual differences between them and so to promote easier code readability the game actions package is placed within the *event* package and dragonCard actions placed in *dragonCard* package.

Observer

Observer patterns allow taking action on a set of objects when and if the state of another object changes. It does so by having subscribers subscribe to events, this is managed by an event manager that is the publisher. This promotes a more flexible and modular design by decoupling the subjects from their observers and providing a standardised way for communication by reacting to events.

The advantage is that you can set up the pattern without knowing about the events or subscribers beforehand and can introduce new subscribers without changing the publisher's code. Here, it is used in conjunction with the command design pattern for controlling the game flow using various events. Various commands (actions) subscribe to events and once there events occur they are notified by the publisher and the command is executed. These commands in turn can trigger other events or execute other commands that can trigger events and by doing so a dynamic gameplay can be set up. This setup allows for modifying and even introducing new game flow easily by introducing new events and commands. An example of this is the initial game board setup. The dragonCards manager subscribes to the event that is *GAMESTART,* when the game starts this event is triggered and the dragonCards manager shuffles the cards. This also means that the GAMESTART event can be used to restart the game and none of the code would have to change.

Singleton

Singleton pattern is used to ensure that only a single instance of the Event manager exists throughout the application. By enforcing a single instance of the Event Manager throughout the application, it establishes a centralised hub for managing all game events. This prevents the proliferation of multiple event manager instances, which could lead to inconsistent event handling and potential conflicts. It also means that as a Singleton, the Event Manager instance is globally accessible from any part of the game. This accessibility ensures that all game components, regardless of their location or hierarchy, can subscribe to and trigger events seamlessly.

Key Classes

DragonCardManager class

This class is used to manage dragon cards. The DragonCardManager is responsible for initialising and configuring dragon cards based on the specifications provided in the configuration file. This involves parsing the configuration data, instantiating card objects, and setting their attributes accordingly. Another key responsibility of the DragonCardManager is to shuffle the deck of dragon cards and distribute them as needed during gameplay. This ensures randomness and fairness in card draws, contributing to the overall balance of the game.

MatchedAction class

MatchedAction serves as a critical component for organising the game flow, particularly in response to specific events triggered during gameplay. This class essentially listens to the matched creature event and when that event is triggered it executes the move action, moreover this action has dependency on TileManager and other classes which are going to be injected by dependency injection.

Key Relationships

Composition between Creature and CreatureFactory:

- The CreatureFactory controls the full life cycle of Creature instances. Creatures don't exist without the creature factory which creates new instances of creatures by reading from the configuration file and also stores them.

Inheritance relationship between standard dragon card and abstract dragon card:

- StandardDragonCard inherits from the abstract DragonCard class. This allows for future extension by adding more dragon card types while ensuring they implement the Action interface and can execute actions.

Decisions around inheritance

Not using abstract creatures class

- It was decided to not create abstract creatures and instead use a creature factory. This was because the creatures can be read from the config file and unlike dragon cards there is no requirement for them to do any action. A creature class which acts as the blueprint for new creatures is sufficient and so is used.

Actions interface.

- This was made an interface instead of an abstract class as card actions are dynamic in that they can be made to perform any action that will potentially affect anything in the game. Therefore the only main requirement is that any action must be able to execute that action and what it stores will vary depending on the specific action and so no attributes were required. This allows for more dynamic approach as dependency injection can be used to inject required dependencies and every action may have different dependencies.

Cardinatiles

PlayerManager and Player (1 to 1..*):

- The PlayerManager holds and creates a list of players, storing multiple instances. However, there must be at least one player for the game to work (e.g., playing against an AI).

BoardManager and Tiles (1 to 2..*):

- There must be at least two tiles in the game (e.g., the player starts at a cave, moves to a volcano tile, and back to the cave tile as a base case).