Design Rationale

Design Patterns Used

With game programming in general, there are certain design patterns that ensure that there is nice architecture so the code is easier to understand over the lifetime of the project, and future developers can get features built quickly without drastically changing the project.

To help with these tasks I've included the following design patterns to streamline the development process.

Observer Pattern

Majority of the game elements in Fiery Dragons require a certain condition or event to engage. These events must be global since multiple actions can be performed after a certain event.

This is where the Observer Pattern comes into play. We call the game element that triggers this event as the publisher and game elements that need to change their state due to this trigger as subscribers.
At game runtime, all the game elements subscribe to events that they are interested in, when the publisher triggers the event, it also notifies all the subscribers of the trigger. These subscribers then process the trigger in the way they've defined it.

In Fiery Dragons players can only interact with the buttons, but since the buttons and the backend for the logic of Fiery Dragons is neatly uncoupled, we cannot directly call the logic as the button doesn't know beforehand about these classes. Hence, the button notifies all the subscribers of the player turn starting which in turn leads to the game knowing about the user picking a dragon card.

Hence we can introduce new subscriber classes without having to change the publisher's code (and vice versa if there's a publisher interface) and establish relations between objects at runtime.

Command

Reducing the dependencies on each class helps decouple the code and stop classes from having too many responsibilities, but there are multiple functionalities where we need these classes to interact with multiple classes.

Like the Dragon Card which during an execution, will need to access multiple classes like the creature, the player, and game tiles. These responsibilities can be moved from the dragon card with the introduction of the Command classes.

The command classes will now store all the information to perform a particular action and get executed at any time by any class that has all the required information.
A MoveDragonCardCommand will accept the player, the dragon card and the tile, and then execute all the logic within the execute method ensuring single responsibility principle and reducing dependencies on game element classes.

Strategy
Certain objects have different ways of doing the same task according to a particular attribute. For example the player object, depending on whether it is played by a human or an AI, will have different methods of selecting a dragon card.

Instead of manually checking if the player is controlled by a human or the computer, we set up a behaviour strategy for the player which includes the human behaviour and the AI behaviour. The player will now know that it has a behaviour but not know the specifics about it.

We define the different functionalities in the behaviours themselves. This makes the code incredibly extensible for future behaviours ensuring that we don't have to change the player class to reflect these changes.

Singleton
Singleton is a pattern that should be used very carefully as it completely violates the single responsibility principle when used incorrectly.

As singleton provides a global access point to all classes in the game scope, I've only used it in either instances where I can swap it out with static methods or classes that can potentially be accessed by every single class.

For example, the Settings class that provides access to the game config data that needs to be accessed by any class, and the EventManager where classes should have access to the EventManager to subscribe or notify of certain events.

These classes do not allow the game components to access information that they otherwise should not be able to access, thus these classes have a justified reason to implement the singleton pattern.

Explaining the Why's

Key Classes

DragonCardCommand (MovePlayerCommand)

The DragonCardCommand was introduced to get rid of multiple inter class dependencies within the DragonCard classes. Without them, these classes would have to handle multiple responsibilities during run time. The DragonCard would have access to multiple classes and use the functionalities of multiple methods to complete a task. This would lead to an incredible amount of coupling.
To eradicate this, we introduced the DragonCardCommands which now holds information regarding the execution of a particular dragon card, removing all the unnecessary coupling.

EventManager

In the game, multiple classes need to interact with each other after a specific trigger. Originally this was handled by chaining multiple function calls together. This was a huge hassle essentially when we needed to wait for the output to chain all the way back up. With the introduction of the EventManager and the observer pattern, interaction between classes changed drastically. Classes would now only interact with the EventManager and notify all subscribers instead of manually interacting with all entities.

Key Relationships

Aggregation Relationship between GameBoard and PlayerManager:

In the GameBoard class, an instance of PlayerManager is created and used to manage players within the game. This relationship represents an aggregation, where GameBoard contains a PlayerManager instance to manage players. It's an essential part of the game's functionality, but PlayerManager can exist independently of GameBoard. This aggregation relationship allows GameBoard to delegate player management responsibilities to PlayerManager, enhancing modularity and separation of concerns.

Association Relationship between DragonCardButton and DragonCardManager:

The DragonCardButton class interacts with the DragonCardManager to perform actions related to dragon cards, such as flipping cards and managing their state. This relationship represents an association, where DragonCardButton is associated with DragonCardManager to access and manipulate dragon card data. The association allows DragonCardButton to interact with the broader game system through DragonCardManager, enabling dynamic behaviour based on game state and player actions.

Inheritance

In the game code base, I've employed inheritance wherever classes share anything that can be abstracted, but completely do away with these abstractions when I realise that these classes can be combined into one and exist as objects of the same class. For example the Creature class could be made into an abstract class having various inheritors like the bat, salamander and baby dragon. But these classes do not have much information that can justify branching them out into their own classes. They only have unique data, all functionality is shared among them, leading to the decision of creating a concrete Creature class, which is the reason why I opt to not use the Visitor Pattern in my design.

Key Cardinalities

The relationship between a player and a tile is a 1 and a 0..1 implying that a tile may have a player at any time but the player must always be on a tile, allowing for the possibility of an unoccupied tile.

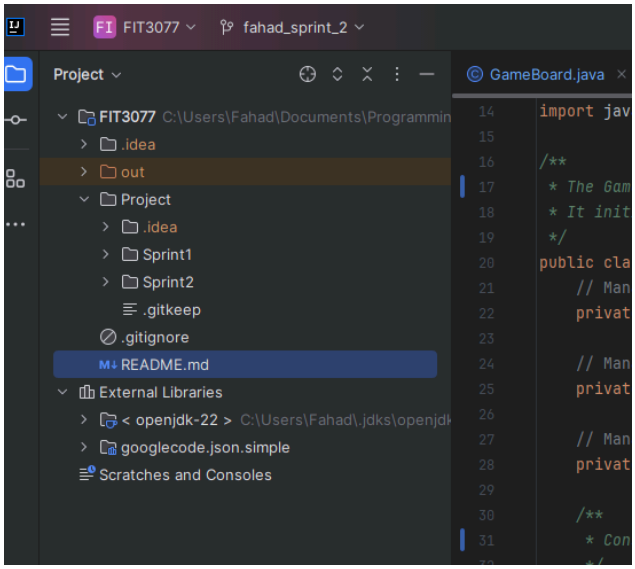
The relationship between managers and their composite entity is always 0..* as they may have 0 or more than 0 depending on the config file used to initialise them.

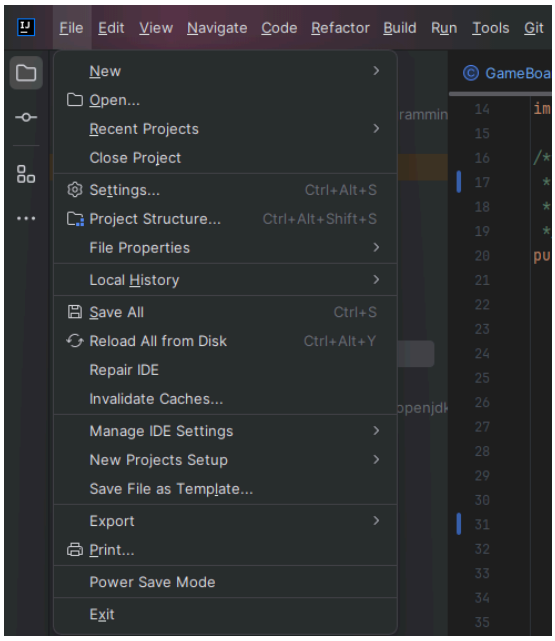
Executable Deliverable

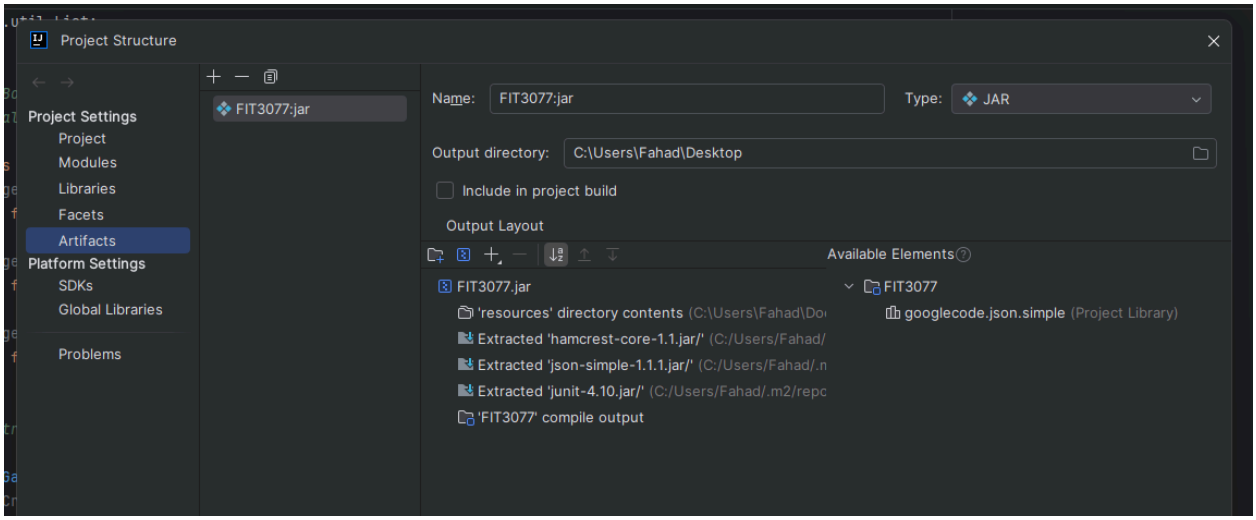
Branch: fahad_sprint_2

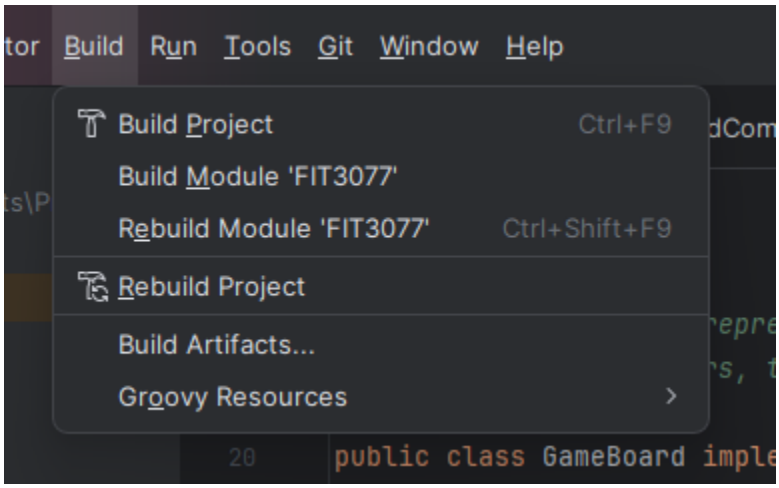
VideoLink: https://youtu.be/-L2oJd8Asck mp4 file format.

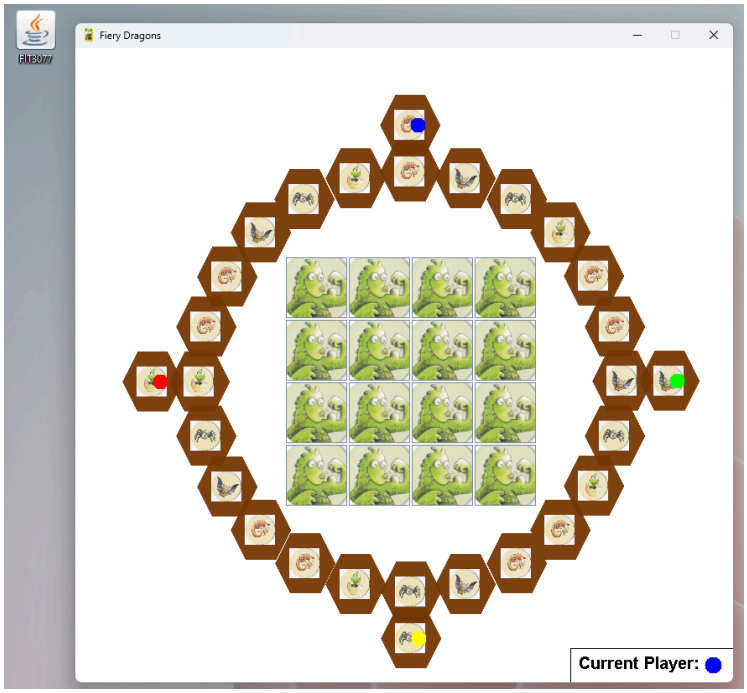
Installation instruction:






Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
FIT3077:jar
Name: FIT3077:jar
Type: JAR
Output directory: C:\Users\Fahad\Desktop
Include in project build
Output Layout
Available Elements
FIT3077.jar
'resources' directory contents (C:\Users\Fahad\Do
Extracted 'hamcrest-core-1.1.jar/' (C:/Users/Fahad/
Extracted 'json-simple-1.1.1.jar/' (C:/Users/Fahad/.n
Extracted 'junit-4.10.jar/' (C:/Users/Fahad/.m2/repc
'FIT3077' compile output
FIT3077
googlecode.json.simple (Project Library)


Build
Run
Tools
Git
Window
Help
Build Project Ctrl+F9
Build Module 'FIT3077'
Rebuild Module 'FIT3077' Ctrl+Shift+F9
Rebuild Project
Build Artifacts...
Groovy Resources
20
public class GameBoard imple


Fiery Dragons
Current Player: