**FIT3077 - Semester 1, 2024**

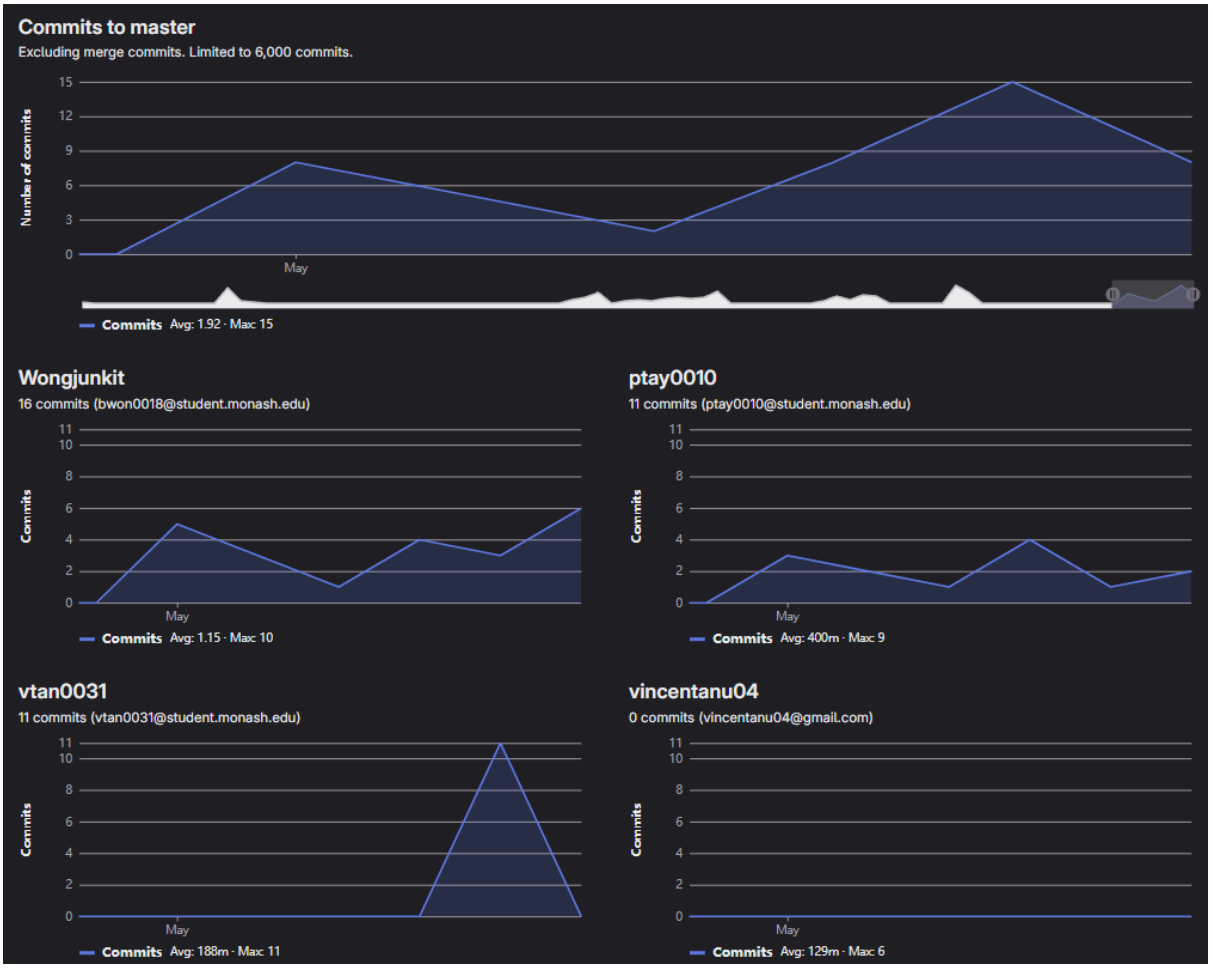
# Sprint Four

**Prepared by:**

**Team MA_Tuesday12pm_Team005**
**(Vincent Tanuwidjaja, Po Han Tay, Bryan Wong)**
{vtan0031, ptay0010, bwon0018}@student.monash.edu

# Table Of Contents

# 1. Contribution Analytics

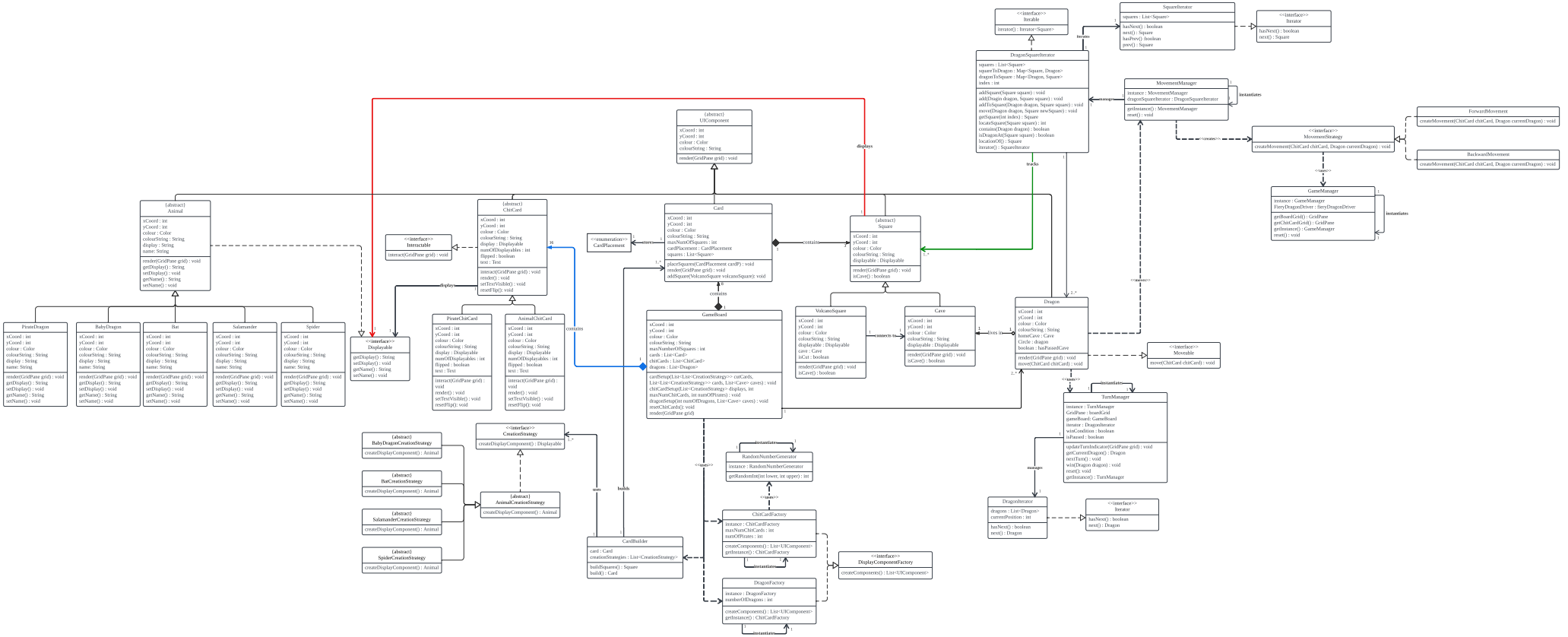



All the commits to the Software Prototype for this Sprint Four. The work distribution on the document is provided in each student's entry within the GitLab Wiki.

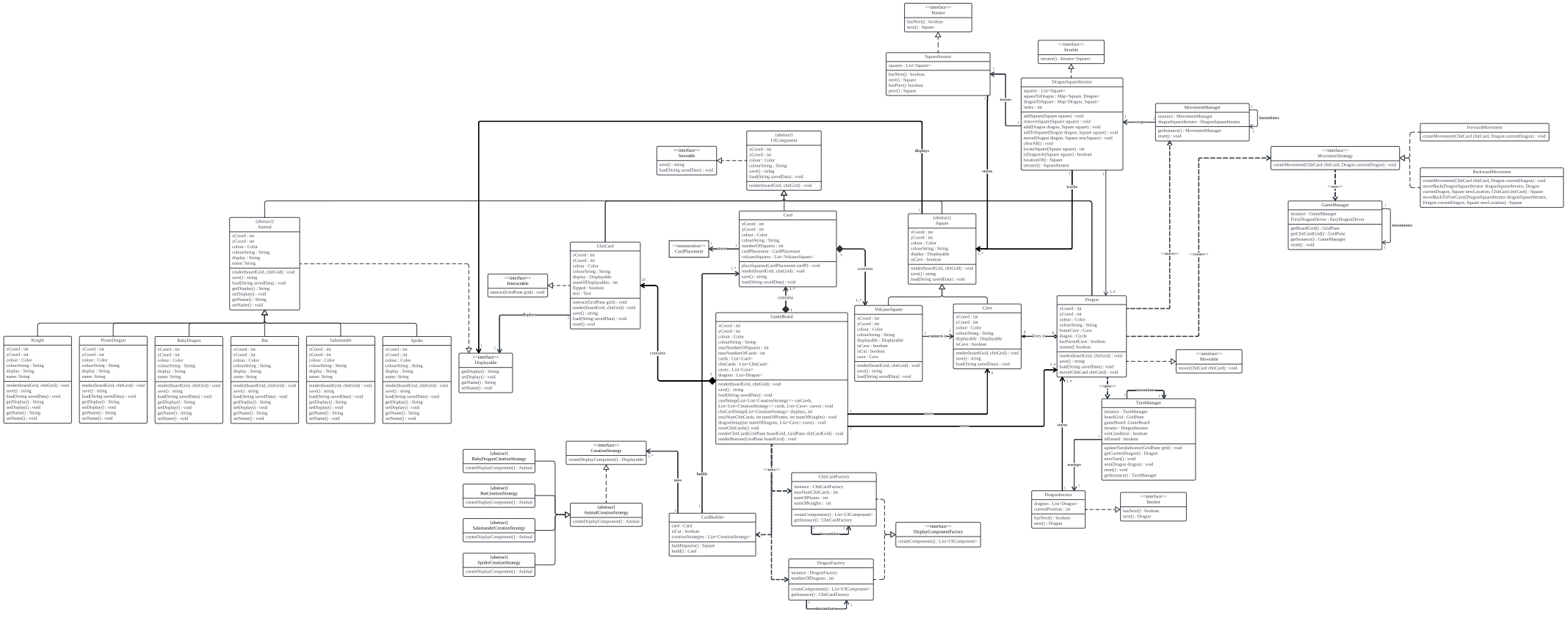
# 2. Class Diagram

(look at next page for both Sprint 3 and Sprint 4 Class Diagram)

# 2.1 Sprint 3 Class Diagram



# 2.2 Sprint 4 Class Diagram



## 2.3 Differences between Class Diagrams

Implementation of New Dragon Card 1 (Knight):

- The addition of **the Knight** class extends from the **Animal** base class as it behaves similarly to the other **Animal Displayables**.
- The **BackwardMovement** class contains 2 additional methods to simulate the normal backward movement with the **PirateDragon** (*moveBack()*) and the new backward movement with the **Knight** (*moveBackToFreeCave()*).
- Cardinality at **ChitCard** for the composition relationship between it and the **GameBoard** is 17 as there will be a total of 17 **ChitCards** with this implementation.

Implementation of saving and loading:

- Addition of a **Saveable** interface, which enables a saving and loading feature to the game. It enforces the *save()* and *load()* methods for its child classes.
- **UIComponent** class implementing the **Saveble** interface, which means all its subclasses will implement the *save()* and *load()* method
- Added association relationship between **SquareIterator** and **Square** with a cardinality of 1 and 1..* respectively. This is because the **SquareIterator** stores the varying number of Squares in the game, which can be more than 1.
- Cardinality at **Card** for the composition relationship between it and **GameBoard** is 1..* as this implementation supports the varying **Cards** in the game.
- Cardinality at **VolcanoSquare** for the composition relationship between it and **Card** is 1..* as this implementation supports the varying **VolcanoSquares** in a **Card**.
- **GameBoard** class stores a list of **Caves** as its class variable used for saving and loading purposes.
- **GameBoard** class contains an additional method to display the save and load buttons in the game using the *renderButtons()* method.
- **DragonSquareIterator** class contains 2 additional methods used for the loading feature: *removeSquares(),* which removes **Squares** from the game and *clearAll()*, which clears all the content within its class variables.

Implementation of Self-Defined Extension - Attack and Stun Dragons (Skip Turn):

- Addition of *stunned* boolean attribute along with its getter and setter method in the **Dragon** class, allowing its value to be set based on the ‘attacking/stunning’ logic and obtaining its stunned state (True or False).

Non-requirement implementation changes include:

- Removal of redundant **RandomNumberGenerator** class as the **ChitCards** are shuffled randomly using the *shuffle()* method from the Collections framework.
- Removal of redundant **PirateChitCard** and **AnimalChitCard** classes as it already uses the Displayable interface to set varying **Animal Displayables**.
- Added association relationship between **DragonIterator** and **Dragon** with a cardinality of 1 and 2..4 respectively. This is because the **DragonIterator** stores the varying number of **Dragons** in the game, which can be 2 to 4.

- Cardinality at **Dragon** for the association relationship between it and the classes, **DragonSquareIterator** and **GameBoard**, are 2..4 as there will only be 2 to 4 **Dragons** in the game.

# 3. Reflection on Sprint 3 Design

## 3.1 New Dragon Card 1 (Knight)

### 3.1.1 Pros

**Animal Abstract Class**

One positive was the inclusion of the **Animal** abstract class and its subclasses in Sprint 3, allowing the **Knight** to inherit the abstract class and implement the common functionalities shared between all **Animal Displayables** without modifying existing code, like rendering the **Animal** in the UI. This adheres to OCP, promoting code reusability and maintainability.

**Movement Creation Strategy Design Pattern**

In addition, the **Knight ChitCard** utilises the **BackwardsMovement** Strategy from Sprint 3 to move the **Dragon** back into the closest **Cave**. The **BackwardMovement** utilises the **DragonSquareIterator** design pattern to iterate through a list of **Squares** backwards (anti-clockwise) based on the flipped **ChitCard.** Because of this, adding the new extension was simple as minor modifications were required and existing code can be reused, showcasing the extendability of our codebase from Sprint 3.

### 3.1.2 Cons

**Unused ChitCardFactory Design Pattern**

The current **ChitCardFactory** is designed for specific **Animal ChitCards** (**BabyDragon**, **Bat**, **Salamander**, **Spider**) and cannot handle **Knight** or other future **Animals**. This limits the reusability of the factory design pattern for new **ChitCards**. The factory pattern's initial goal was to centralise object creation and allow for easy addition of new types for future Sprints. However, due to time constraints in Sprint 3, the creation of **Animal ChitCards** was fixed and hardcoded, resulting in the factory becoming inflexible. Introducing the **Knight** required manual creation within the **GameBoard** class, defeating the purpose of the factory pattern and hindering future extensibility.

**Fixed Frontend Layout**

As Sprint 3 made no mention of more than 16 **ChitCards**, the UI was fixed to display them in a 4x4 grid. Although the back-end allowed a varying number of **ChitCards**, the front-end was fixed in order to save time. This rigidity makes adding new **ChitCards** like the **Knight** problematic. Scaling the UI to accommodate a different number of **ChitCards** and displaying them in a different layout required significant adjustments, which violates the OCP.

### 3.1.3 Lessons Learned and Moving Forward

**Avoid Hardcoding**

Our decision to fix the **ChitCard** UI layout to a 4x4 grid backfired on us, resulting in additional work to fit the **Knight ChitCard** (17 **ChitCards**) in the UI. This demonstrates that

despite us wanting to save time and effort by hardcoding the layout, it resulted in additional work in the end. If we made accommodations for the dynamic placement of **ChitCards** in the UI in Sprint 3, this extension would have been easier. Furthermore, we left this addition to the last minute, preventing us from making major overhauls in our current codebase. As the requirements evolve, our current implementation may not fit the new standards, so in the future, we should schedule regular code reviews and refactoring sessions to ensure that the codebase remains flexible and adaptable to new requirements.

**Proper usage of Design Patterns**
Due to our haphazard planning, we "cut corners" in the implementation of the **ChitCardFactory**, not taking into account future extensions that may reuse this design pattern. As a result, this increased the difficulty of our extension implementation in Sprint 4. This has taught us a valuable lesson on how to think beyond the immediate requirements and short-term solutions can lead to long-term complications. In the future, we should consider future changes and code with the assumption of future extensions and plan for scalability by anticipating changes in design requirements.

## 3.2 Saving and Loading

### 3.2.1 Pros

**Division of Classes**
The decomposition of the specific board **UIComponents** like **GameBoard, Card**, **Square**, **Cave**, **Dragon** and the **ChitCards** proved to be a good choice in Sprint 3 when implementing the saving and loading extension. Initially, this was done so that future extensions or modifications could be done on specific components without creating side effects and affecting the entire system. As each class manages and oversees its own saving and loading logic, it simplifies our extension, making the overall process modular and easier to maintain. This also adheres to OOP design like the Open-Close Principle (OCP).

**Render() Template Method Design Pattern**
The "render()" template method design pattern (implemented by **UIComponent** subclasses) was initially created in Sprint 3 to render the board into the UI and create the UI elements in the **GridPane**. In Sprint 4, this method simplifies loading significantly. After loading the saved data of the **UIComponents** (coordinates, colours, etc.), the **GameBoard** "render()" method can be called to recreate the entire board into the UI. This saves development time and reduces code duplication as instead of re-rendering the individual **UIComponent** subclasses one by one, a single "render()" method is called to recreate the whole board in the UI.

### 3.2.2 Cons

**Render() Method Overhead**
Each **UIComponent** "render()" call will create new UI objects for the **GridPane**, instead of modifying pre-existing ones. Meaning that additional logic is required to remove the old UI elements every time the game is loaded (clearing **GridPane** and resetting the UI), resulting in

increased complexity, longer development time and larger overhead costs. Originally, the render() method was intended only for creating and displaying UI elements during game setup. In Sprint 4, we reused the “render()” method but had to write extra code to make it work in this new context.

**Iterator Design Pattern for Movement**
Our current implementation from Sprint 3 relies on an Iterator Design Pattern (**DragonSquareIterator**). Hence, when loading different configurations of **Cave** positions and number of **Squares**, the **SquareIterator** list has to be modified to accommodate these new changes. For example, if a **Card** previously had 3 **Squares** and it’s reduced to 2 once loaded, the **Square** has to be removed from the **DragonSquareIterator**, adding another layer of complexity and introducing potential errors. Similarly, if the **Cave** was relocated, the **SquareIterator** needs to be modified to move the **Cave** to the new position.

**Hardcoded Front-end**
Another major limitation in our approach in Sprint 3 was the hardcoded front-end for the Game Board and the UI. This means the layout is designed for a specific number of **Squares** per **Card** (3), and a specific number of **Cards** (8). Although our back-end is flexible enough to accommodate a varying number of **Squares** per **Card** and number of **Cards**, the same cannot be said for the UI. As a result, visual inconsistencies are created such as gaps between Squares/Tiles, which may disrupt gameplay.

### 3.2.3 Lessons Learned and Moving Forward
The shortcomings identified in Sprint 3's design and implementation faced when extending our game offer valuable lessons for the future. From this, we learn a few lessons such as:

**Prioritising Flexibility**
We underestimated the importance of a flexible UI. To save time, we opted to “cut corners” by hardcoding the front-end for a specific number of **Squares** and **Cards**, limiting our ability to handle different game configurations. In the future, we'll prioritise designing UIs that can adapt dynamically based on the underlying data. This might involve using layouts like **FlowPane** or employing data-driven techniques where the UI elements are generated based on the loaded game data. Thus, we learnt a valuable lesson to consider the long-term implications of our design choices. What may seem like a “simpler” way in the beginning may end up costing us more in the long run.

**Unsuitable Design Pattern**
The choice of the **DragonSquareIterator**, although sufficient for Sprint 3's needs, proved to be a limiting factor when extending the game. Design patterns effectiveness depends on the specific context and they aren’t a one-size-fits-all solution. In this case, it struggled with dynamic changes in the number of **Squares** and the different **Cave** positions. In hindsight, we should have made the iterator a Linked List instead. As a Linked List is a dynamic data structure, it aligns perfectly with our need to handle varying numbers of **Squares** per **Card**.

New **Squares** can be added or removed from the list effortlessly by just modifying the links of each **Square**. One key takeaway from this is that we need to choose the right method for the job, instead of trying to shoehorn a design pattern into everything as while design patterns offer valuable tools, they should be evaluated in the context of the specific problem.

## 3.3 Self-Defined Extension - Attack and Stun Dragons (Skip Turn)

Our extension enables the Player to prevent other **Dragons** from progressing their turn. When a Player lands on a tile occupied by another Dragon, it attacks said **Dragon** and stuns them, forcing them to skip a turn. The attacking **Dragon** will then return to their previous **Square** before the attack, ending their turn. This extension addresses the Human Value category 'Power', specifically the Human Value 'Authority' as this allows Players to gain more control over the game by being able to disrupt their opponents' progress. The extension introduces an element of risk and reward. Players need to weigh the potential benefits of attacking another **Dragon** instead of moving across the **GameBoard** and closer to their home **Cave** to win the game.

### 3.3.1 Pros

**Clear Responsibility Separation**

In Sprint 3, we had a clear separation of responsibilities for the moving logic. The "move()" method in **DragonSquareIterator** was responsible only for moving and placing a dragon to its final next **Square** on the **GameBoard** while the "move()" method in **Dragon** calls **ForwardMovement** and **BackwardMovement** to manage the logic on where to move. The "nextTurn()" method in **TurnManager** managed the changing turn logic. This clear separation of concerns made the code modular and easier to manage and extend. Through this, adding the additional logic for attacking **Dragons** on occupied **Squares** is relatively straightforward and simple as minor modifications can be made in the **ForwardMovement** "createMovement()" method to implement this extension without creating any side effects in the underlying pre-existing moving logic. Furthermore, to end the **Dragon** turn after attacking, the preexisting "nextTurn()" method can be reused.

### 3.3.2 Cons

As this extension was relatively easy to implement, there are not many negatives of the old design/implementation that are discovered for this specific extension.

**Scalability Concerns**

As the game evolved and introduced new mechanics in Sprint 4, such as different **Dragon** behaviours or statuses, for example, the stunned attribute, the "createMovement()" functionality became increasingly complex as evidenced by the multitude of conditional statements. This could lead to scalability issues, making it difficult to manage and maintain the codebase over time, resulting in more side effects and bugs. During Sprint 3, when there were only a few movement behaviours (forward and backwards), a patchwork of if statements were used due to its simplicity. While it didn't hinder the implementation of this extension, it may result in unwieldy and hard-to-debug code in the future. Hence, the current

implementation is a ticking time bomb and may result in unintentional complications in the future.

### 3.3.3 Lessons Learned and Moving Forward

**Clear Responsibility Separation**

The clear separation of responsibilities facilitated by the distinct roles of the "move()" method in **DragonSquareIterator**, **Dragon**, and **TurnManager** significantly contributed to the implementation of this self-defined extension. By doing this, it became easier to introduce the new functionality of skipping an attacked **Dragon's** turn when it is attacked by another **Dragon**. This modular design allowed for targeted modifications to specific components without affecting the overall system, enabling a smoother integration of the new feature into the existing codebase. Moving forward, this approach should be further emphasised and extended to other areas of the codebase, as well as future modifications. By identifying and separating clear responsibilities for each component, future extensions can be implemented with greater ease. This has taught us a valuable lesson in thinking in the long-term instead of our immediate goals, as we were rewarded with easier implementations now for doing so in Sprint 3.

**Scalability Concerns**

As the game evolves with new requirements, such as new dragon-related mechanics, scalability concerns may arise, potentially impacting maintainability and performance as mentioned earlier. We discovered that code refactoring and optimisation are essential to address these challenges, ensuring the codebase remains flexible and adaptable. Future incorporation of comprehensive unit tests and automated test suites can also further enhance stability, reliability and ease of extensions. Following these will help ensure the project's long-term maintainability.

## 4. Executable Description

A .jar file located in the ‘executable’ folder of the ‘Sprint Four’ folder

## 4.1 Steps to compile the JAR file

- Ensure this is run on a Windows device
- SDK coretto-17 (Amazon Coretto 17.0.11)
- Ensure Java Runtime Environment (JRE) is at least version 8
- JavaFX SDK version 17.0.11

1. Open Project Structure:
   - Go to File → Project Structure or press Ctrl+Alt+Shift+S.

2. Configure Artifacts:
   - In the Project Structure dialog, select Artifacts under Project Settings.
   - Click the + (plus sign) to add a new artifact.
   - Choose JAR → From modules with dependencies.

3. Select Main Class:
   - Select the Main.java class located in src/com/fierydragon

4. Extract to the Target JAR:
   - Choose to Extract to the target JAR to include your dependencies in the JAR.
   - Click OK to save the configuration.
   - Click the + (plus sign) at the Output Layer section and click File.
   - Locate the folder where JavaSDK is installed.
   - Select everything in the bin folder.
   - Click OK to save the bin folder location.
   - Click Apply and then OK again to close the Project Structure dialog.

5. Build Artifact:
   - To build the JAR, go to Build → Build Artifacts.
   - Select the artifact you configured and choose Build.

6. Locate JAR File:
   - The JAR file will be saved in the out/artifacts/ directory within your project folder.

## 4.2 Steps to run the JAR file

1. Locate the executable folder
   - Open your terminal and cd into the folder where the .jar file is.
2. Execute the .jar file
   - Run the command `java -jar MA_Tuesday12pm_Team005.jar` in the terminal

OR

1. Locate the executable folder and double-click the .jar file to run it.