FIT3077 - Software engineering: Architecture and design

Sprint 2:
Design Rationale

Desmond Chong Qi Xiang(33338248)

- Explain two key classes (not interfaces) that you have included in your design and provide your reasons why you decided to create these classes.

Tile

In adhering to the Single Responsibility Principle (SRP), the decision was made to create a separate class for tiles within the game. This choice ensures that each class is responsible for only one aspect of the game's functionality, thus promoting maintainability and clarity in the codebase. By isolating the functionality related to tiles into its own class, we avoid the pitfall of having overly lengthy methods that handle multiple responsibilities, which could become cumbersome to manage and modify.

Moreover, the creation of the Tile class aligns with the Open-Closed Principle (OCP), which encourages software entities to be open for extension but closed for modification. By encapsulating the behaviour of tiles within a distinct class, we allow for future enhancements and modifications to the functionality of tiles without needing to alter existing code. For instance, if future iterations of the game introduce additional features such as tiles that enable free movement rather than being fixed at specific positions, we can extend the functionality of the Tile class through inheritance or composition, thereby maintaining the integrity of the existing code while accommodating new requirements.

In summary, the decision to create a separate Tile class reflects a commitment to adhering to fundamental software design principles, namely the Single Responsibility Principle and the Open-Closed Principle. This approach enhances the maintainability, extensibility, and clarity of the game code, facilitating future modifications and enhancements with minimal disruption to existing functionality.

GameBoard

The GameBoard class serves the purpose of initialising the game and rendering the screen. This design adheres to the Single Responsibility Principle (SRP) by segregating the responsibilities of game flow management and user interaction from other aspects of the game. By assigning the task of initialising the game and rendering the screen to the GameBoard class, we avoid the creation of a "god class" where all functionalities are consolidated into a single entity.

Moreover, the separation of concerns in this manner promotes code maintainability and clarity. By encapsulating the initialization and rendering logic within the GameBoard class, we ensure that the Game class remains focused on orchestrating the overall game flow without being burdened by the intricacies of screen rendering. This division of responsibilities facilitates easier code maintenance and future enhancements.

Furthermore, the design of the GameBoard class aligns with the Open-Closed Principle (OCP), which encourages software entities to be open for extension but closed for modification. The GameBoard class provides a clear interface for initialising the game and rendering the screen,

allowing for potential extensions or modifications to be implemented through subclassing or composition without the need to alter existing code. This design flexibility enables the GameBoard class to adapt to evolving requirements or additional features without disrupting the core functionality of the game.

In summary, the GameBoard class exemplifies a design approach that promotes adherence to fundamental software design principles, including the Single Responsibility Principle and the Open-Closed Principle. By separating concerns and encapsulating responsibilities within dedicated classes, we enhance code maintainability, clarity, and extensibility, ultimately contributing to the robustness and longevity of the game codebase.

- Explain two key relationships in your class diagram

The relationship between the ChitCard class and the DragonToken class is characterised by dependency rather than association. This signifies that the functionality of the ChitCard class is reliant on the presence and behaviour of instances from the DragonToken class, without directly possessing them as attributes. Interactions between ChitCards and DragonTokens are facilitated through function arguments within the Game class. Consequently, modifications to the DragonToken class are anticipated to impact the ChitCard class, as ChitCard is reliant on the behaviour of DragonToken instances, notably for actions such as relocating the DragonToken during gameplay.

The relationship between the Game and GameBoard classes is best described as association. In an association relationship, one class (Game) is aware of another class (GameBoard) and interacts with it to perform certain tasks, but there is no strong ownership or containment between them. In the context of a game, the Game class serves as the orchestrator of the game's logic and flow, while the GameBoard class handles the initialization of the game and rendering of the game screen. Although the Game class may create instances of the GameBoard class and call its methods to perform specific actions, such as setting up the game environment or updating the display, the GameBoard objects exist independently of the Game class and can be reused or modified without directly affecting the Game class itself. This loose coupling allows for greater flexibility and modularity in the design, as changes to the GameBoard class do not necessitate modifications to the Game class, and vice versa. Overall, the association relationship between the Game and GameBoard classes facilitates collaboration between them while maintaining their distinct responsibilities and independence within the game architecture.

- Explain your decisions around inheritance

My decision to utilize inheritance in the design involved creating an abstract ChitCard class, with ForwardChitCard and BackwardChitCard classes inheriting from it. This approach was justified by the existence of two distinct types of chit cards in the game: those that move the player forward and those that move the player backward. While these chit cards share common attributes such as animal type and quantity, their effects on the player differ. By implementing inheritance in this manner, I aimed to adhere to the principles of Liskov Substitution and Dependency Inversion.

The Liskov Substitution Principle ensures that subclasses can be substituted for their base class without affecting the correctness of the program. In this case, both ForwardChitCard and BackwardChitCard can be treated as instances of the ChitCard class, allowing for seamless interchangeability in the game logic. Moreover, by abstracting common attributes and behaviors into the base class, the Dependency Inversion Principle is upheld, as dependencies are now on abstractions rather than concrete implementations.

Furthermore, the design aligns with the Open-Closed Principle by allowing for extensibility and future enhancements. For instance, if additional types of chit cards with different effects are introduced in the future, they can simply extend the ChitCard class, ensuring that the existing code remains closed for modification but open for extension.

- Explain how you arrived at two sets of cardinalities

The cardinality between the Game class and the Occupiable entities is 1 to 28, representing that each instance of the Game class controls all 28 Occupiable entities in the game. These entities include 24 tiles and 4 caves. With the Game class implemented as a singleton, only one instance exists throughout the game, ensuring centralised control over all Occupiable entities.

The cardinality between the Game class and the Drawable entities is 1 to 48. This means that each instance of the Game class is responsible for managing and drawing a total of 48 Drawable entities during gameplay. These entities include 4 caves, 4 DragonTokens, 16 ChitCards, and 24 Tiles. The rendering of these entities is exclusively handled by instances of the GameBoard class, with the Game class delegating this responsibility.

- Explain why and where you have applied a particular design pattern in your design.

Observer pattern:
In this game, each player's win condition is determined by that player entering back into their own cave. Since there are four players, each with their own cave, and the win condition is specific to each player, the Observer pattern might not be the most suitable choice.

The Observer pattern is typically used for establishing a one-to-many relationship, where multiple observers (subscribers) are interested in changes to a single subject (publisher). However, in this scenario, the win condition for each player is tied directly to their individual cave. There is not a clear one-to-many relationship where multiple entities are observing changes to a single subject.

Instead, the game logic revolves around tracking the movement and state of each player individually, specifically monitoring whether each player has entered their respective cave. This can be more effectively managed through direct state management within the game logic, rather than implementing the Observer pattern, which might introduce unnecessary complexity and overhead.

In summary, while the Observer pattern is useful for scenarios where multiple entities observe changes to a single subject, it may not be the most appropriate choice for situations where each entity has its own distinct win condition, as in this game where each player must enter their own cave to win.

Strategy pattern:
In the design of the ChitCard class, the Strategy pattern has been implemented to manage variations in movement behaviour for different types of ChitCards. This design choice was made to ensure flexibility and maintainability in handling diverse movement rules without necessitating modifications to existing code. The Strategy pattern allows encapsulating specific movement logic within separate strategy classes, such as ForwardChitCard and BackwardChitCard, which represent different types of ChitCards. Within the ChitCard class, the move_dragon_token method serves as the context for the Strategy pattern, delegating movement logic to the appropriate strategy class based on the type of ChitCard being processed. By employing the Strategy pattern, the ChitCard class achieves a clear separation of concerns, promotes code reusability, and facilitates seamless addition of new ChitCard types in the future. Overall, the Strategy pattern enhances the maintainability and extensibility of the ChitCard functionality by encapsulating variations in movement behaviour within dedicated strategy classes.

Singleton pattern:
The Singleton pattern has been applied to the Game class to ensure that only one instance of the class exists throughout the game's lifecycle. This design choice was made because the game requires a centralised controller to handle all aspects of game flow, such as managing player turns, updating game state, and determining win conditions. By restricting the instantiation of the Game class to a single object, the Singleton pattern ensures that all game logic remains consistent and coherent, preventing issues such as conflicting game states or unintended multiple instances.

Additionally, the Game class contains attributes such as the screen, which is required by all drawable entities to render themselves on the game board. By using the Singleton pattern, these attributes can be accessed globally by all parts of the game without the need for passing them explicitly as parameters. This promotes code simplicity and encapsulation, as the Game class serves as a centralised hub for accessing shared resources and managing game state.
In summary, the Singleton pattern has been applied to the Game class to enforce a single instance throughout the game, ensuring consistent game flow and providing a centralised point of access for shared resources such as the screen. This design choice enhances code organisation, promotes encapsulation, and simplifies resource management within the game.