MONASH University

FIT2099 - Object-oriented design and implementation

Assignment 3:
Design Rationale

Team Members:
Ong Chong How (33363102)
Desmond Chong Qi Xiang (33338248)
Maliha Tariq (33473692)

MONASH University

Changes made to the design for Assignment 2:

Spawner Class Refactoring:

The feedback from A2 suggested that the classes ForestEnemySpawnableGround and VillageEnemySpawnableGround were not providing substantial benefits and were potentially introducing unnecessary complexity. The introduction of a second abstract middle class was considered to complicate the design further, potentially affecting the overall maintainability.

In response to this feedback, the design has been simplified by eliminating the ForestEnemySpawnableGround and VillageEnemySpawnableGround classes. Instead, the EnemySpawnableGround class has been modified to directly handle the spawning of enemies.

The change reflects a development in the design approach, offering more flexibility and better functionality for handling the spawning of enemy actors within the game. By consolidating the enemy spawning logic into the EnemySpawnableGround class, the code becomes more streamlined and easier to maintain. The modified design aligns with the principles of DRY (Don't Repeat Yourself) and SOLID (Single Responsibility Principle) by reducing redundant code and simplifying the class hierarchy.

Despite the removal of the middle abstract classes, the current design maintains its extensibility through the utilisation of the Spawner interface. This approach allows for easy integration of different enemy types and spawning mechanisms, ensuring a scalable and adaptable architecture for accommodating future enhancements or expansions in the game environment.

MONASH University

REQ1: The Overgrown Sanctuary

Adding a Multi-Destination Gate

The implementation of a multi-destination gate in a game, specifically within the context of a text-based adventure game. The primary goal is to allow players to access new areas by defeating a boss character (Abxervyer) and interacting with a gate, which can have multiple destinations. This design adheres to key software design principles, such as DRY (Don't Repeat Yourself), SOLID principles, and connascence.

Gate Class Refactoring:

The existing Gate class serves as the foundation for implementing multi-destination gates. It is designed as an environment represented by '=' on the game map. It has attributes to keep track of whether it's locked or unlocked, a list of travel actions, and the ability to notify and perform resets. By utilising these existing attributes, we extend the class to support multiple destinations.

Implementation of Multi-Destination Gates:

The player can now access a new area, The Overgrown Sanctuary, in addition to returning to the Ancient Woods. When the boss, Abxervyer, is defeated, the location where the boss last stood is transformed into a gate. The new gate has travel actions that allow players to access both the Ancient Woods and The Overgrown Sanctuary. This flexibility in destinations is achieved without duplicating code by adding multiple travel actions to the same gate.

Pros:	**Cons:**
Code Reusability (DRY Principle): This design promotes code reusability. The same Gate class is used for both destinations. Travel actions are added to the gate, reducing code duplication. **Extensibility (Open-Closed Principle):** The design allows for easy extension to support more destinations in the future. Additional travel actions can be added to an existing gate without modifying the Gate class.	**Complexity:** While the design is flexible, it can become more complex as the number of destinations increases. Proper documentation and organisation of code are essential to mitigate this. **Error Handling:** With multiple destinations, error handling becomes crucial. Players might encounter issues if the gate's state or destinations change unexpectedly.

Simplicity and Maintainability: The design keeps the gate-related logic centralised within the Gate class, making it easier to maintain and understand.	

Ways in Which the Design Can Be Easily Extended:

New Areas: Introduce new areas that players can access through multi-destination gates. These areas can have their unique environments, challenges, and quests.

Quest System: Create a quest system that requires players to complete tasks in different areas accessed through gates, unlocking additional gates and story progression.

Gate System: Implement a dynamic gate connection system where gates can be activated or deactivated based on in-game events or triggers.

Character Control: Allow players to control multiple characters or have companions, each with their unique abilities or limitations, making gate traversal a strategic choice.

Hidden Gates: Scatter hidden gates throughout the game world that lead to secret areas, powerful items, or hidden quests, rewarding explorative gameplay.

Implementing the Eldentree Guardian Enemy and Spawner

The implementation of the "Eldentree Guardian" enemy character and the associated spawner in the game. The design rationale explains the decisions made when implementing the "Eldentree Guardian" enemy character and the associated spawner. It provides insights into why specific attributes, behaviours, and capabilities were chosen for this character and how it fulfils the given requirements.

Eldentree Guardian Class:

The EldentreeGuardian class represents a powerful enemy character, the Eldentree Guardian, which is tasked with defending a specific location in the game world. The class extends the Enemy class, inheriting attributes and behaviours related to enemy characters.

Extensibility (Open-Closed Principle):

The design adheres to the Open-Closed Principle. The EldentreeGuardian class can be easily extended to introduce new behaviours, capabilities, or items that the character can drop. This provides a foundation for adding more diversity to enemy characters in the game.

Pros:	**Cons:**
Reusability (DRY Principle): The design encourages code reusability by extending the Enemy class, inheriting common properties and behaviours, and only specifying unique characteristics of the Eldentree Guardian.	**Complexity:** The introduction of multiple behaviours, random loot drops, and capabilities can increase the complexity of the enemy character. Careful testing and documentation are essential to ensure smooth gameplay.
Extensibility (Open-Closed Principle): The design allows for easy extension by introducing new enemy characters with different behaviours and capabilities, promoting variety in gameplay.	**Balancing:** Balancing the chances of loot drops and behaviour transitions might require fine-tuning to provide a fair and challenging gameplay experience.
Abstraction and Polymorphism: The use of the Enemy class as a base class and extending it for specific enemies demonstrates abstraction and polymorphism principles. It provides a structured way to handle various enemy types.	

Behavioural Flexibility: The ability of the Eldentree Guardian to switch between wandering and following behaviours based on player proximity adds depth to its character and interaction.	

Ways in Which the Design Can Be Easily Extended:

New Enemies: Extend the design to introduce new enemy types with diverse behaviours and abilities. This can include enemies with unique attack patterns, special abilities, and interactions with the game world.

Faction System: Create a system where enemies belong to different factions, each with its unique characteristics and relationships with other factions. This can lead to complex interactions and alliances among enemy characters.

Special Abilities: Introduce special abilities for enemies, such as teleportation, summoning reinforcements, or creating environmental hazards. These abilities can add complexity to enemy encounters.

Living Branch Enemy character and its spawner

Living Branch Spawn Mechanism:

The Living Branch is an enemy character represented by the symbol "?". It is designed to spawn from the bushes in the game world. The spawning chance is set to 90% at each turn, making it a frequent occurrence.

Object-Oriented Design (OOD):

The design of the Living Branch enemy and its spawner adheres to object-oriented principles, promoting modularity and reusability.

DRY (Don't Repeat Yourself):

To avoid code duplication, a spawner pattern is employed. The LivingBranchSpawner creates Living Branch instances, centralising enemy creation logic.

Polymorphism:

The Living Branch class extends the Enemy class, demonstrating polymorphism. This allows the Living Branch to be treated as an Enemy while possessing unique attributes and behaviour.

Abstraction:

The Living Branch class encapsulates its attributes and behaviour, abstracting the details from the client code. This simplifies the interface for interacting with enemy actors in the game.

SOLID Principles:

The single responsibility principle (SRP) is followed by ensuring that each class has a single, well-defined purpose, enhancing maintainability and reducing coupling.

Pros:	Cons:
Modular and Extensible: The use of a spawner pattern and OOD principles makes the code modular and extensible. Future enemies with similar behaviours can be added by creating new subclasses. **Gameplay Variety:** The Living Branch introduces variety to the game by offering a unique enemy with different characteristics, including a chance to drop a special item (BloodBerry). **Clear Separation of Concerns:** By employing a spawner, the responsibility for enemy creation is separated from other game logic, simplifying maintenance and additions of new enemies. **Code Reusability:** The Living Branch class inherits from the Enemy class, leveraging existing enemy behaviour. This promotes code reusability and maintains a consistent interface for interacting with enemies.	**Limited Flexibility:** While the current design supports enemies with predetermined attributes, it may be less suitable for enemies with complex, dynamically changing behaviours. **Reduced Variation:** The Living Branch's behaviour is somewhat limited, as it does not wander or follow the player, which might reduce gameplay diversity.

Reasons for Choosing the Current Design:

The design choice to use a spawner pattern and extend the Enemy class was made to ensure a modular and extensible codebase. It allows for the addition of various enemy types with distinct behaviours while maintaining consistency in the codebase.

The decision to avoid wandering or following behaviour for the Living Branch aligns with its role as a unique, stationary enemy that emerges from the bushes. This design choice promotes gameplay diversity and unpredictability.

Ways in Which the Design Can Be Easily Extended:

Dynamic Behaviours: Can implement more dynamic behaviours, such as wandering or following, for other enemy types.

Special Items: Introduce additional special items and enhance the drop system for defeated enemies.

Powerful Enemies: Create enemies with varied immunities, providing additional challenges in different game scenarios.

REQ2: The Blacksmith

The requirement is fulfilled through the implementation of the Blacksmith class and the UpgradeAction class. The Blacksmith class serves as a character enabling players to upgrade specific items and weapons. The UpgradeAction class provides the necessary functionality for upgrading various items, including healing vials, refreshing flasks, broadswords, and great knives, enhancing the player's gameplay experience and character progression.

Reasons for Choosing the Current Design:
The current design is chosen to maintain a clear separation and ensure a modular code which adheres to the principles of DRY and SOLID. The design of the Blacksmith and the associated upgrading mechanism was chosen based on several key principles and desired functionalities. By ensuring that each class and interface in the system has a distinct and clear responsibility, the design remains modular and easier to maintain. Additionally, the design heavily leverages the concept of polymorphism, providing a consistent framework for introducing new types of upgradable items or actions in the future. Through the strategic use of interfaces and abstract classes (e.g. the Upgradable interface), the design offers future additions or modifications without necessitating significant changes to the existing codebase. Hence, this design promotes a more organised and maintainable code structure while minimising code duplication and complexity.

Pros:	**Cons:**
Single Responsibility Principle (SRP): The design adheres to the SRP, ensuring that each class is responsible for a specific set of functionalities, promoting a more focused and manageable codebase. **Encapsulation:** The design encapsulates the upgrade functionalities within the UpgradeAction class, ensuring controlled access to the upgrade process and promoting secure and organized code management. **Liskov Substitution Principle (LSP):** The design adheres to the LSP, ensuring that the UpgradeAction class can be used interchangeably with different upgradable	**Complexity in Upgrade Logic:** The design may introduce complexity in managing the upgrade logic for various items and weapons, potentially leading to intricate code structures that are difficult to maintain and debug.

items, promoting code reusability and interoperability. **DRY Principle:** The codebase utilizes methods and classes efficiently, reducing code repetition. For example, the UpgradeAction class can be reused for various upgradable items	

Ways in Which the Design Can Be Easily Extended:

Introduction of New Upgradable Items: The design can be extended to incorporate new upgradable items, such as accessories or consumables, ensuring they can be upgraded by the Blacksmith. For example, if a new armour type is introduced, implementing Upgradable will make it fit seamlessly into the current upgrading system.

Implementation of Blacksmith Interaction Levels: The design can be extended to introduce varying interaction levels with the blacksmith, enabling players to unlock special benefits based on their upgrade history. The allowableActions method in Blacksmith can be extended to provide new interactions, such as trading or repairing equipment.

Introducing New Upgrade Mechanisms: If there's a need to introduce a different upgrade mechanism in the future, say enchanting, a new action like EnchantAction can be added and the items can implement a new interface like Enchantable.

REQ3: Conversation (Episode I)

Talkable

The Talkable interface represents entities within the game that can engage in a monologue with an actor. It provides methods for adding monologue options and choosing a random monologue from the available options.

Reasons for Choosing the Current Design:
The current design of the Talkable interface is chosen to ensure a modular and extensible approach to managing monologue functionalities within the game. It adheres to the principles of DRY (Don't Repeat Yourself) and SOLID, particularly emphasising the Single Responsibility Principle (SRP) by separating the concerns of managing monologue options and selecting from them. This approach promotes maintainability and scalability in the codebase, enabling easier modifications and extensions as the game evolves.

Pros:	Cons:
DRY (Don't Repeat Yourself): Promotes code reusability by encapsulating the behavior of actors who can engage in conversations and provide monologue options. **Single Responsibility Principle (SRP):** Adheres to the SRP by defining a clear and focused set of methods related to conversations. Actors implementing this interface are responsible for managing their monologue options. **Encapsulation:** The interface encapsulates the implementation details of monologue functionality, preventing unnecessary exposure of internal workings and promoting a more secure and controlled code structure. **Reusability:** With a standardised interface for managing monologues, the design promotes the reuse of monologue-related code across various entities and scenarios, enhancing overall development efficiency.	**Scope Limitation:** While the Talkable interface provides a basic framework for handling conversations, it might not cover more complex scenarios or features that can arise in a game, such as branching dialogues, or dynamic conversation state changes, potentially necessitating additional extensions or modifications to accommodate advanced monologue features.

Ways in Which the Design Can Be Easily Extended:

Conditional Monologue Options:
By extending the design to include conditional monologue options, the “Talkable” interface can support context-specific dialogue interactions that change based on specific in-game events or the current state of interacting characters.

Implementing Different Conversation Styles:

Can extend the interface to support various conversation styles or modes. For example, we could introduce methods like startFormalConversation() and startInformalConversation() to cater to different types of interactions with NPCs. This allows us to customize the conversation behavior based on the context or character type.

ListenMonologueAction

The ListenMonologueAction class represents an action within the game where an actor listens to a monologue provided by a Talkable speaker. It facilitates communication and interaction between game characters through the exchange of monologues.

Reasons for Choosing the Current Design:
The current design of the ListenMonologueAction class is chosen to ensure a focused and streamlined approach to managing monologue interactions within the game. It aligns with the principles of DRY and SOLID, particularly emphasizing the Single Responsibility Principle (SRP) by concentrating on the specific task of enabling monologue exchanges. The design also emphasizes encapsulation and reusability, promoting a modular and extensible codebase that allows for future enhancements and modifications.

Pros:	Cons:
DRY (Don't Repeat Yourself): The class minimizes code redundancy by consolidating the monologue interaction logic in one place, promoting efficient code maintenance and readability. **Single Responsibility Principle (SRP):** By concentrating on the singular responsibility of managing monologue interactions, the class ensures clear and concise code organization, facilitating easier maintenance and modifications. **Encapsulation:** The class encapsulates the monologue-related functionality, safeguarding the internal implementation details and promoting a more secure and controlled code structure. **Modularity:** The design promotes modularity by encapsulating the monologue-related functionality within the class. This modularity facilitates the separation of concerns and promotes a more organized codebase, making it easier to understand, maintain, and extend the monologue-related features independently from other parts of the game.	**Scope Limitation:** The class might encounter limitations in managing complex monologue interactions that involve intricate dialogue management or dynamic character-specific responses. Extending the class may be necessary to accommodate advanced monologue features, potentially leading to increased development complexity. **Complexity:** While the design emphasizes modularity and extensibility, it might introduce additional complexity overhead, especially when implementing more intricate monologue interactions or integrating advanced dialogue management systems

Ways in Which the Design Can Be Easily Extended:

Character-Specific Responses: By introducing character-specific response mechanisms, the design can be extended to allow the ListenMonologueAction class to generate tailored and contextually appropriate responses based on the unique traits and attributes of the interacting characters. This extension would enable the game to provide more authentic and immersive character interactions, enhancing the overall storytelling and character development aspects of the game.

Interactive Dialogue Trees: Extending the design to include interactive dialogue trees can enhance the complexity and depth of the monologue interactions within the game. By implementing a proper dialogue tree structure, the class can support branching dialogues that offer players various conversation paths and choices, thereby providing a more engaging and personalized narrative experience.

REQ4: Conversation (Episode II)

The addition of the conversation functionality in Episode II builds upon the existing dialogue system established in REQ3. This new feature allows the player to engage in monologues with the isolated traveller found in the Ancient Woods. The design is structured to seamlessly integrate the traveller's monologue options with the pre-existing dialogue options, ensuring a coherent and immersive narrative experience for the player.

Reasons for Choosing the Current Design:
The current design is selected to ensure seamless integration of the new dialogue functionality for the isolated traveller within the existing game framework. It aligns with the principles of DRY and SOLID, particularly emphasizing the Single Responsibility Principle (SRP) by focusing on the distinct responsibility of managing dialogue interactions. The design also emphasizes encapsulation and reusability, facilitating the incorporation of new dialogue options and characters while minimizing code redundancy and promoting a modular and maintainable codebase

Pros:	**Cons:**
Single Responsibility Principle (SRP): The design adheres to the SRP by focusing on the specific task of managing the isolated traveller's dialogue interactions, ensuring a clear and well-defined purpose for the implemented dialogue system. **Encapsulation**: The design encapsulates the dialogue-related functionality, safeguarding the internal implementation details and promoting a more secure and controlled code structure, contributing to improved code organization and maintenance. **Reusability**: By leveraging the modular structure and the integration of the new dialogue options, the design promotes code reusability, facilitating the seamless integration of new characters and dialogue features into the existing game framework without necessitating significant modifications.	**Potential Increased Complexity**: The integration of the new dialogue functionality for the isolated traveller might introduce increased complexity, especially when managing multiple character interactions and conditional dialogue options. This complexity could potentially impact the overall development process, requiring additional resources for comprehensive testing and maintenance.

Ways in Which the Design Can Be Easily Extended:

Dynamic Dialogue Expansion:
This extension allows the dialogue system to adapt and present new dialogue options based on the player's progress or choices. For instance, as the player reaches specific milestones, new dialogue options could become available, branching the narrative based on the player's actions.

Advanced Dialogue Management Support:
This extension allows the dialogue system to handle complex dialogue structures, enabling multi-path conversations, decision-making, and context-driven dialogue triggers. It improves the player's interaction with the game world, providing a more immersive and interactive storytelling experience.
Example: During pivotal plot points, the isolated traveller's dialogue options could branch into different narrative paths, offering the player choices that impact the outcome of subsequent events, leading to diverse consequences and story developments.

REQ5: A Dream?

Requirement 5 entails several tasks to be performed upon a player's death, which include dropping corresponding Runes at the player's death location, respawning the player at their initial starting point, and clearing out both enemy spawns and dropped runes from players and enemies. Additionally, it's essential to ensure that the Forest Watcher is restored to full health. To achieve these objectives, we've devised a design that involves the creation of two interfaces and one class, providing a structured framework for implementing this functionality. The purpose of these interfaces and the class will be further explained below.

ResetNotifiable interface

We've established a "ResetNotifiable" interface to signal classes like EnemySpawnableGround, ForestWatcher, Gate, and Runes about the player's demise, prompting them to initiate specific reset actions within their respective "tick" or "playTurn" methods in the subsequent turn.

Pros:	**Cons:**
Dependency Inversion Principle (DIP): This is applied through the utilisation of the ResetNotifiable interface. This enables the system to employ the notifyReset() method without having to be concerned with the specifics of the actions required when announcing a player's demise. Instead, the system can rely on high-level modules or abstractions to handle the announcement process. This abstraction helps decouple the lower-level implementation details from the higher-level modules, promoting flexibility and easier maintenance in the system. **Extensibility (Open-Closed Principle):** The design exhibits extensibility in line with the Open-Closed Principle. This means that introducing new entities which require a reset notification when the player dies can be seamlessly accommodated. As a result, the class responsible for announcing the game's reset requirement does not need to undergo modification to accommodate these new entities. Instead, we ensure that the new	**Violence DRY(Don’t Repeat Yourself) Principle:** While it's true that the classes implementing the ResetNotifiable interface share similar code within their notifyReset() methods, we've chosen not to adhere strictly to the DRY (Don't Repeat Yourself) Principle in this context. This decision is based on considerations of system maintainability and extensibility. By allowing some repetition in the code, we maintain the flexibility to easily extend the announcement functionality in the future, which outweighs the redundancy in the present implementation.

entities implement the resetNotify() method independently. This approach enhances the system's adaptability and minimises the need for extensive modifications when adding new features or entities. **Abstraction and Polymorphism:** Through the implementation of the ResetNotifiable interface, classes requiring interaction with reset notifications are shielded from the specifics of the classes they interact with. Instead, they can simply invoke the notifyReset() method of any class that implements the interface. This demonstrates the principles of Abstraction and Polymorphism, allowing for a more flexible and adaptable system.	

Ways in Which the Design Can Be Easily Extended:

Adding New ResetNotifiable Entities: Introducing new entities that require reset notifications can be seamlessly integrated by having them implement the ResetNotifiable interface. This allows them to participate in the reset notification process without requiring significant modifications to existing code.

Resettable interface

We've introduced a "Resettable" interface to facilitate the implementation of concrete reset actions in classes like Enemy, ForestWatcher, Gate, Player, Runes, and EnemySpawnableGround. This enables us to invoke the reset action within the "tick" or "playTurn" methods as needed.

Pros:	**Cons:**
Interface Segregation Principle: The Interface Segregation Principle is demonstrated in this design. While some classes implement both the Resettable and ResetNotifiable interfaces and can effectively perform both the reset and notifyReset methods, there are also classes that either only utilize one interface or cannot perform both methods simultaneously. This indicates the importance of segregating the interfaces, ensuring that classes adhere to a specific set of methods relevant to their functionality. This approach promotes a more streamlined and cohesive system design. **Dependency Inversion Principle (DIP):** While it's possible to perform the reset action directly without invoking the reset method, adhering to the Dependency Inversion Principle (DIP) requires avoiding dependencies on low-level modules and instead relying on abstractions. This is why we've introduced the interface, allowing classes to implement this method. As a result, the tick and playTurn methods are relieved from concerning themselves with the specific implementation details, and can simply rely on these abstractions. This approach fosters a more flexible and modular system design.	**Violating Open-Closed Principle:** While there are currently no classes dedicated to handling the reset of resettable game entities, this may potentially violate the Open-Closed Principle due to the addition of reset actions in the tick and playTurn methods. However, the introduction of this interface enables any new game entities to implement it, thereby providing a specialised method for reset actions. This allows for easy adaptation to incorporate new entities with unique reset requirements without necessitating substantial modifications to existing code. **Increased Complexity:** The introduction of this interface may lead to an increased level of complexity within the system. It's worth noting that the system can still function without this interface by directly implementing specific reset actions in each resettable class. However, in the long run, prioritizing maintainability and extensibility may warrant this trade-off in complexity brought about by the interface. This decision serves to ensure a more adaptable and manageable system in the future.

Abstraction and Polymorphism: While there isn't currently a class capable of handling all resettable entities at once, the potential for achieving abstraction and polymorphism remains readily accessible. This may be realized in the future if a class is introduced to manage these entities comprehensively.	

Ways in Which the Design Can Be Easily Extended:

Expanding Reset Actions: If additional actions are needed during the reset process, they can be easily incorporated within the reset method of the relevant classes. This provides a straightforward way to introduce new reset behaviours without disrupting the existing system.

Customizing Entity-Specific Reset Logic: The interface provides a framework for implementing specialised reset logic for specific entities. This allows for tailored reset actions that cater to the unique requirements of individual game elements.

ResetNotifiableManager:
We've established a ResetNotifiableManager to efficiently manage and simultaneously notify all classes implementing the ResetNotifiable interface.

Pros:	**Cons:**
Singleton pattern: The Singleton pattern is applied to this class by restricting the constructor to private access and allowing it to be created only once when the getInstance method is invoked. This design choice guarantees that the system maintains a single instance of the ResetNotifiableManager. Moreover, the ResetNotifiableManager is conventionally available globally within the application, enabling any part of the code to easily access and share resources or data across various components. **Single-responsibility Principle (SRP):** The Single-responsibility Principle (SRP) is observed by segregating the notify reset functionality into a dedicated class, rather than directly managing it within the player class when a player dies. This separation ensures that each class has a distinct and specific responsibility, making the functionality more maintainable and extensible in the long run.	**Global State:** Since the Singleton is accessible globally, it can introduce shared state across different parts of the application. This can lead to unexpected interactions and make it harder to reason about the code. **Overuse:** There's a potential risk of overusing the Singleton pattern, which could result in an excessive dependency on global state. This may lead to a more intricate and less modular codebase. However, this trade-off can alleviate the challenge of manually adding the ResetNotifiable to the list managed by the ResetNotifiableManager.

Ways in Which the Design Can Be Easily Extended:

Optimizing Notification Performance: The introduction of new methods or strategies can be employed to streamline the notification process and improve overall system efficiency.