Assignment 2: Design Rationale

Ang Qiao Xin | Chew Ken Yang | Er Jun Yet | Gan Ruiqi

33520054 | 30881234 | 33521026 | 33521204

FIT2099 – Object Oriented Design and Implementation

9th May 2024

Introduction

In Assignment 2, we have been tasked with working on further modification of the text-based “rogue-like” game on Static Game Factory. There are four key requirements for this game system which are as follows:

Req 1 - The moon’s (hostile) fauna II : The moon strikes back, where new hostiles such as Alien Bug and Suspicious Astronaut spawned from the moon’s craters wanders around.

Req 2 - The imposter among us, where new hostiles such as Alien Bug and Suspicious Astronaut may attack or follow the Intern, or even steal items from the ground.

Req 3 - More scraps, where the Intern finds more scraps on the moon, such as a Jar of Pickles which can heal or hurt the Intern, and a Pot of Gold which can earn golds for the Intern.

Req 4 - Static factory’s staff benefits, where the Intern can access a Computer Terminal to purchase items like Energy Drink, Dragon Slayer Sword and Toilet Paper Roll.

Each requirement above plays a crucial role in enhancing the gameplay experience and contributing to the overall functionality and immersion of the Static Game Factory.

This report aims to delve into a thorough analysis on the chosen code implementation and the design rationale behind this game system implementation. The designs chosen for each requirement have all adhere to one of the most important principle in object-oriented design and programming, known as the SOLID principles, which includes the Single Responsibility Principle (SRP), Open-Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP) and Dependency Inversion Principle (DIP), and also Don't Repeat Yourself (DRY).

Design Rationale

Req 1 - The moon's (hostile) fauna II: The moon strikes back

The first design task revolves around the *Hostile* subclasses especially *AlienBug* and *SuspiciousAstronaut,* as well as the *Spawner* subclasses especially *BugSpawner* and *AstronautSpawner* classes, which represent the new hostile creatures and its spawning on the abandoned moon in the game map. The design is proved to have adhered the SOLID and DRY principles with the following justification below.

AlienBug, SuspiciousAstronaut and HuntsmanSpider subclasses extends Hostile Class

Hostile abstract class encapsulates common attributes and behaviours shared by hostile creatures, including *AlienBug, SuspiciousAstronaut* and *HuntsmanSpider* that extends *Hostile* parent class, promoting cohesion and minimising coupling between classes. This code delegation clearly demonstrates adherence to the SRP because each *Hostile* subclass is responsible for handling their own unqiue attributes and operations like its *playTurn()* and *getIntrinsicWeapon()*, while also inheriting common attributes that all *Hostile* have. This also adheres to the DRY principle that prevents duplication of code across multiple classes and ensures consistency and reducing the likelihood of errors.

As far as a flexible and extensible game framework is concerned, the game system can easily accommodate new hostile creatures by creating additional classes that extend the *Hostile* abstract class. Each new entity can define its unique attributes and behaviours while leveraging common functionality provided by the abstract class. This abstraction approach demonstrates adherence to the OCP because the design allows for extension without modification to support new hostile types by extending new subclasses of *Hostile* parent class without modifying the base code. This promotes code stability and minimizes the risk of introducing bugs when extending the system.

Moreover, the inheritance from *Hostile* abstract class to its subclasses and perhaps other more child classes in the future demonstrates adherence to the LSP, where the subclass can be substituted for their *Hostile* parent class without altering the correctness of the program. In other words, subclasses of *Hostile* like *AlienBug, SuspiciousAstronaut* and *HuntsmanSpider* can be used interchangeably with other hostile entities wherever *Hostile* instances are expected.

BugSpawner, AstronautSpawner and SpiderSpawner subclasses implements Spawner Class

As far as the future game extension is concerned, *Spawner* class is assigned as an interface class because the *Spawner* subclasses like *BugSpawner, AstronautSpawner* and *SpiderSpawner* do not need to implement methods that it do not care about and only care about spawning *AlienBug, SuspiciousAstronaut* and *HuntsmanSpider* respectively. This abstraction design demonstrates adherence to the DIP due to the loose coupling between *Crater* class and *SpiderSpawner* class that relies on abstractions rather than concrete implementation of the spawning of many *Hostile* creatures in the *Crater* concrete class itself. Rather now, *Crater* class is refactored to have a *Crater()* parameterised constructor that takes in the specific *Spawner* class desired to spawn that particular *Hostile* creature.

Spawner interface class is introduced to promote new spawning mechanisms like *BugSpawner, AstronautSpawner* and *SpiderSpawner* by specifying its unique implementation of the standardised *spawnHostiles()* method. This interface design demonstrates adherence to the ISP where the design favours narrow and specific segregation interface, where any new type of *Spawner* like *BugSpawner* and *AstronautSpawner* now can implement *Spawner* interface class by implementing straight the important methods needed for being *Spawner* subclasses, like *spawnHostiles()* method. For example, *BugSpawner* does not need to implement any other methods that it does not care about as it only cares about the spawning of *AlienBug*, enhancing code readability by establishing a clear separation of concerns with ISP.

Besides that, this abstraction approach demonstrates adherence to the OCP because the design allows for extension without modification to support new spawner types when more *Hostile* needed to be spawned by creating new *Spawner* subclass that implements *Spawner* interface class to override the given base method of *spawnHostiles()* method. This promotes code stability and minimizes the risk of introducing bugs when the game design requires extension.

However, it is worth noting that the introduction of multiple abstraction layers and class hierarchies may increase the complexity of the system as more subclasses are implemented, potentially making it harder to comprehend when collaborating with other developers who are unfamiliar with the design.

Req 2 - The imposter among us

Next, the second task's design revolves around *SuspiciousAstronaut* and *AlienBug* classes that have specific behaviour to each of the creatures. For example when the Intern enters *SuspiciousAstronaut* surroundings, it will instantly kill the Intern with *AttackBehaviour*, regardless of the Intern's health, with 100% precision while for *AlienBug* it will pick up scraps on ground with *StealAction* and follow player when Intern is within the surroundings with *FollowBehaviour* until it dies or the game ends. This design adhered to the SOLID and DRY principle with the following justification below.

StealBehaviour and FollowBehaviour Classes implement Behaviour Class

StealBehaviour class represents the actor will figure out a *StealAction* that will steal an item from the ground, while *FollowBehaviour* represents a *MoveActorAction* that will move the actor one step closer to a target Actor.

This design follows ISP because of having many specific interfaces instead of one general-purpose interface. For example, the *Behaviour* interface is implemented by different classes like *StealBehaviour* and *FollowBehaviour*, each providing a specific behavior. This allows easier to add new features to the code without needing to refactor. This makes our system more extensible. *FollowBehaviour* and *StealBehaviour* implement a *Behaviour* interface that will only depend on the necessary method which is *getAction()* to reduce unnecessary dependencies.

Moreover, *StealBehaviour* and *FollowBehaviour* follow SRP as each class in the system has a single responsibility. For instance, the *StealBehaviour* class is only responsible for defining the behavior of stealing items. Similarly, the *FollowBehaviour* class only handles the logic for following another actor. This makes the system easier to maintain and understand. Not only that, the system is open for extension but closed for modification (Open-Closed Principle (OCP)). For example, the *StealBehaviour* and *FollowBehaviour* classes implement the *Behaviour* interface. If we need to add a new behavior in the future, we can simply create a new class that implements the *Behaviour* interface without modifying the existing classes or the *Behaviour* interface itself.

This design also follows Dependency Inversion Principle (DIP) in which high-level modules in the system do not depend on low-level modules. Both depend on abstractions. For instance, the *Actor* class does not depend on the *StealBehaviour* or *FollowBehaviour* classes directly. Instead, it depends on the *Behaviour* interface, which is an abstraction. The system also adheres to the

DRY (Don't Repeat Yourself) principle. For example, the *StealBehaviour* and *FollowBehaviour* classes share the same method signature from the *Behaviour* interface, avoiding code duplication.

In an alternative design without interfaces, we could directly implement the functionality within the *Action* subclasses, thereby reducing abstraction layers. In other words, *StealAction* and *FollowAction* are applied immediately whereby the logic of providing the *StealAction* and *FollowAction* for the *Hostile* are carried out respectively in the *StealAction* and *FollowAction* classes that extend *Action* class. However, this approach may lead to code duplication and reduced flexibility, as each class would need to handle the logic of providing that particular *Action* independently in their own *Action* class, which a *Behaviour* class should be in charge of, violating the SRP and DRY principles. As far as the game design is concerned, all NPCs' *Action* should go through the standardised behaviour logic where the *Behaviour* interface class is introduced to allow new different behaviours like *StealBehaviour* and *FollowBehaviour* to implement the *Behaviour* interface to execute the *getAction()* method. The main disadvantage of the alternative design is that no interface is introduced, resulting in difficult enforcement of consistent behaviour across different classes and to achieve code reuse through polymorphism.

StealAction Class extends Action Class

StealBehaviour depends on *StealAction* which will execute the *StealAction* that will call *PickUpAction* from the engine class. The *StealAction* class depends on the *Action* interface, a high-level module, not on low-level modules. It also does not depend on concrete classes but on abstractions which adhere to the Dependency Inversion Principle (DIP). This principle helps manage coupling by ensuring that high-level modules do not depend directly on low-level modules. Instead, both should depend on abstractions. This reduces the direct dependencies between classes, making the system more flexible and easier to change.

However, there is tight coupling with *PickUpAction*: The *StealAction* class is tightly coupled with the *PickUpAction* class. This means that if the *PickUpAction* class changes, it could potentially affect the *StealAction* class. This could be mitigated by introducing an abstraction (like an interface or abstract class) that both *StealAction* and *PickUpAction* could implement, reducing the direct dependency between them.

Moreover, *StealAction* class is a subtype of *Action* and it can be substituted for an *Action* without causing any issues which demonstrates Liskov Substitution Principle (LSP). It provides implementations for the *execute()* and *menuDescription()* methods, as required by the *Action* abstract class. This principle helps manage dependencies by allowing new functionality to be added without changing existing code, thus not affecting other parts of the system that depend on the *StealAction* class. While the *StealAction* class adheres to LSP, it's important to note that LSP can be restrictive. It requires that subclasses behave in the same way as their parent classes. This might not always be desirable or practical, especially when the behavior of the subclass is fundamentally different from the parent class. In such cases, it might be better to use composition instead of inheritance.

The design of *StealAction* class also follows Single Responsibility Principle (SRP) which means it has a single responsibility, which is to handle the action of stealing an item. It does not concern itself with other actions or behaviors. Thus, this makes the code more readable and maintainable. The drawbacks would be if the responsibility of the *StealAction* class needs to change in the future, it may require significant refactoring. This could involve creating new classes or changing the interfaces between classes, which can be time-consuming and error-prone.

Besides, *StealAction* class is open for extension (can create a new class that inherits from it and override its methods) but closed for modification (don't need to modify the class itself to change its behavior) which adheres to the Open-Closed Principle (OCP). This principle helps manage dependencies by allowing new functionality to be added without changing existing code, thus not affecting other parts of the system that depend on the *StealAction* class.

StealAction class also adheres to the DRY (Don't Repeat Yourself) principle. It does not contain duplicated code. The logic for stealing an item is defined once in the execute method. This is achieved by reuse of *PickUpAction* in *StealAction*: In the *StealAction* class, the execute method reuses the *PickUpAction* to perform the stealing action. This avoids duplicating the logic for picking up an item, which is already defined in the *PickUpAction* class.

SuspiciousAstronaut and AlienBug Classes extends Hostile Class

The *SuspiciousAstronaut* class has a single responsibility, which is to represent a specific type of hostile actor in the game. It defines the behavior and characteristics of a Suspicious Astronaut which follows Single Responsibility Principle (SRP). This makes the class easier to maintain and understand.

Furthermore, the *SuspiciousAstronaut* class is open for extension but closed for modification Open-Closed Principle (OCP). It extends abstract *Hostile* class and overrides the getIntrinsicWeapon() method to define its unique behavior which kills the player instantly regardless of player's health. This allows for new functionality to be added without changing the existing *Hostile* class, reducing the risk of introducing bugs.

The inheritance from *Hostile* abstract class to *SuspiciousAstronaut* class child class demonstrates adherence to the LSP where The *SuspiciousAstronaut* class is a subtype of *Hostile* and can be substituted for a *Hostile* without causing any issues. This ensures that the program remains correct even when a *SuspiciousAstronaut* object is used wherever a *Hostile* object is expected.

The *SuspiciousAstronaut* class adheres to the ISP as it only implements the behaviors that are relevant to it. It doesn't have to depend on methods it doesn't use.

In addition, this abstraction approach demonstrates adherence to the Dependency Inversion Principle (DIP) because *SuspiciousAstronaut* class depends on the *Hostile* abstraction, not on concrete classes. This makes the system more flexible and easier to change.

The *SuspiciousAstronaut* class adheres to the DRY principle. It extends the *Hostile* class and reuses its functionality, avoiding code duplication. It only overrides the *getIntrinsicWeapon()* method to provide a specific behavior for the Suspicious Astronaut. Not only that, *SuspiciousAstronaut* class is reusing the *AttackBehaviour* class by adding an instance of it to its behaviors to reuse the functionality of *AttackBehaviour* class.

Additional Status in Status Enum Class

As enum class is particularly a powerful feature that can make our code implementation more efficient and reliable, *Status* enum class in the *game* package, which initially contains only a *HOSTILE_TO_ENEMY* and *HOSTILE_TO_PLAYER* constants, is decided to add a new constant called *DANGER_TO_PLAYER* while also redefining all constants available now.

First off, *HOSTILE_TO_ENEMY* is typically a status of the *Player* that is enemy to all *Hostile* creatures, where the *Player* can attack all *Hostile*. Secondly, *HOSTILE_TO_PLAYER* is typically a status of a *Hostile* that is enemy to *Player,* where it is just disadvantage to *Player*, like *AlienBug*. Lastly, *DANGER_TO_PLAYER* is typically a status of the *Hostile* that is a danger to *Player,* where *Hostile* can attack *Player*.

These enum constants benefit by enhancing code readability by providing self-descriptive status for different *Hostile.* This enum constant works perfectly when our code implementation requires verification of an *Actor* if it is a *Hostile.*

Req 3 - More Scraps

The third design task revolves around the *JarOfPickles* and *Puddle* classes which are implementing the *Consumable* interface. When a *JarOfPickles* is consumed there is a 50% chance that it is expired which will hurt the player by 1 health with *HurtAction* Class. However, there is also a 50% chance that it will heal the player by 1 health with *HealAction* Class. Moreover, when a puddle is consumed it will add the max health of the player by 1 with *HealAction* class. Moreover, the PotOfGold class is a class that extends *Item* class where it is used to add the balance of the player.

JarOfPickles Class implements Consumable Class (involves HealAction and HurtAction class)

This design adheres to DRY principles by utilizing the existing *HealAction*, originally crafted for the *Fruit* class, and introducing the *HurtAction* class for future damage infliction needs. By repurposing *HealActio*n and proactively creating *HurtAction*, the design fosters code reusability and anticipates potential expansions, promoting efficiency and maintainability in the long term.

JarOfPickles implements the *Consumable* interface and extends the *Item* class, encompassing both *HealAction* and *HurtAction* classes. This design aligns with the Single Responsibility Principle (SRP), as *HealAction* focuses solely on healing while *HurtAction* focuses solely on inflicting damage, ensuring clarity and singular purpose for each action. It also adheres to the Open-Closed Principle(OCP) by allowing the addition of new consumable types without modifying existing code, enabling any *Consumable* implementing class to utilize *HealAction* and *HurtAction* seamlessly. The inheritance from the *Item* abstract class to *JarOfPickles* maintains Liskov Substitution Principle (LSP) compliance, ensuring *JarOfPickles* can substitute its superclass without altering program behavior. *JarOfPickles* adheres to Interface Segregation Principle ISP by implementing only the necessary methods from the *Consumable* interface, avoiding unnecessary dependencies. Implementing the *Consumable* interface demonstrates commitment to the Dependency Inversion Principle (DIP), promoting loose coupling and flexibility in the game's architecture.

Puddle Class implements Consumable Class (involves HealAction class)

This design adheres to DRY principles by utilizing the pre-existing *HealAction* promoting code reusability and efficiency in the design.

Puddle class implements the *Consumable* interface and extends the *Item* class, incorporating the *HealAction* class. This design aligns with the Single Responsibility Principle (SRP) by solely focusing on healing the player's max health. It adheres to the Open-Closed Principle (OCP) by enabling the addition of new consumable types without altering existing code. The inheritance from the *Ground* abstract class to *Puddle* ensures Liskov Substitution Principle (LSP) compliance (LSP), allowing *Puddle* to substitute its superclass without affecting program behavior. Conforming to the Interface Segregation Principle (ISP), *Puddle* relies only on specific methods from the *Consumable interface*, avoiding unnecessary dependencies. This design also adheres to the Dependency Inversion Principle (DIP) by relying on abstractions, fostering loose coupling between *Puddle* and the *Consumable* interface, enhancing system flexibility and extensibility.

In an alternative design without interfaces, we could directly implement the functionality within each class, thereby reducing abstraction layers. Each consumable item class, such as *JarOfPickles* or *Puddle*, would have its own methods for consumption behavior, eliminating the need for a separate *Consumable* interface. For example, *JarOfPickles* could have methods for both healing and inflicting damage directly within the class, removing the need for separate *HealAction* and *HurtAction* classes. Similarly, *Puddle* could directly modify the player's health without relying on a *HealAction* class. However, this approach may lead to code duplication and reduced flexibility, as each class would need to handle its own consumption logic independently. Additionally, without interfaces, it may be harder to enforce consistent behavior across different consumable items and maintain separation of concerns.

The main disadvantage of this design without interfaces is reduced flexibility and scalability. Without interfaces, it becomes harder to enforce consistent behavior across different classes and to achieve code reuse through polymorphism. Each class must handle its own consumption logic independently, leading to code duplication and increased maintenance overhead. Additionally, without interfaces, it becomes more challenging to decouple components and swap implementations, making the codebase less adaptable to changes in requirements or future extensions. Overall, this design may result in less modular, harder-to-maintain code compared to a design that utilizes interfaces and adheres to SOLID principles.

PotOfGold Class (involves CollectAction class)

Creating *CollectAction* ensures adherence to DRY principles, facilitating future collection needs without redundancy for enhanced code efficiency.

The *PotOfGold* class, an extension of *Item*, upholds key principles of object-oriented design. It adheres to the Single Responsibility Principle (SRP) by focusing on containing gold, delegating gold collection to the *CollectAction* class. This approach ensures clear separation of concerns. *PotOfGold* remains open for extension, following the Open-Closed Principle (OCP), while maintaining Liskov Substitution Principle (LSP) compliance for seamless substitution within the inheritance hierarchy. Lastly, by depending on abstractions such as *CollectAction*, it aligns with the Dependency Inversion Principle (DIP), facilitating flexibility in dependency management.

However, it is worth noting that the design's multiple classes and interactions can lead to increased complexity as the codebase expands, posing challenges in understanding, maintaining, and modifying the code. Dependency management complexities arise from relying on abstractions like the Dependency Inversion Principle (DIP), potentially causing issues with understanding dependency flow and debugging. Additionally, implementing SOLID principles may introduce performance overhead due to increased abstraction layers and method calls, impacting performance, particularly in performance-critical areas. These factors collectively raise concerns about codebase scalability, maintenance, and performance optimization.

Req 4 - Static Factory's staff benefits

Finally, the fourth design task is about creating a computer terminal that allows the player to purchase items or weapons that would benefit the player. A few new classes are created such as the *Terminal* class, *DragonSlayerSword* class, *EnergyDrink* class, *ToiletPaperRoll* class and *Purchasable* interface. This design is proved to have adhered the SOLID and DRY principles with the following justification below.

Terminal Class extends Ground Class

As far as the object-oriented design is concerned, *Terminal* class exhibits common key properties and functionalities like *Ground* abstract class in the provided *engine* package, where the *Player* can purchase weapons or items using it. This design adheres to the DRY principle by allowing shared functionality to be implemented in the *Ground* abstract class and reused by its child classes.

Ground abstract class encapsulates common attributes shared by all grounds in the future, including Terminal that extends *Ground* parent class, promoting cohesion and minimise coupling between classes. This code delegation clearly demonstrates adherence to the SRP because each *Ground* subclass is responsible for handling their own unique definition and attributes like *setUpPurchasableItems()* for the setting the items that are purchasable to the *Player* in *Terminal* class, while also inheriting common attributes and methods that all grounds should have, such as *allowableAction()* and *canActorEnter()* for all *Terminal,* which is *BuyAction* for now.

The abstraction of *BuyAction* demonstrates adherence to the DIP due to the loose coupling between the *Terminal* class that relies on abstractions rather than the concrete implementation of the *BuyAction* in the *Termina*l class. Moreover, the implementation of the *BuyAction* class with the *Terminal* class facilitates the decoupling of the purchase mechanics from the user interface logic, and demonstrates adherence to the SRP. This is due to the fact that *Terminal* objects should be only responsible for defining its unique attributes, while the *BuyAction* class is responsible for allowing the *Player* to buy weapon items or items using the computer terminal.

Additionally, this abstraction approach demonstrates adherence to the OCP because the design allows for extension without modification to support new *Ground* or even more types of *Action* by the *Ground*, by creating new *Ground* subclass or overriding new *allowableAction()* method for

the *Ground*. This promotes code stability and minimises the risk of introducing bugs when the game design requires extension.

EnergyDrink Class extends Item Class (implements Purchasable and Consumable Classes)

Item abstract class encapsulates common attributes and behaviours shared by *EnergyDrink* class, promoting cohesion and minimise coupling between classes. This code delegation clearly demonstrates adherence to the SRP because each *Item* subclass is responsible for handling their own unique definition and attributes like *hitPoints* for the computation in *Consumable* interface and *price* for the computation in *Purchasable* interface, while also inheriting common attributes and methods that all items should have, which is *allowableAction()* for all items, which is the *Consumable* and *Purchasable* for now. The introduction of the Purchasable interface class demonstrates adherence to the SRP.

The abstraction of *Consumable* and *Purchasable* demonstrates adherence to the DIP due to the loose coupling between *EnergyDrink* class that relies on abstractions rather than concrete implementation of the *Consumable and Purchasable* in *EnergyDrink* class. Moreover, the implementation of *Consumable* class and *Purchasable* class with *EnergyDrink* class facilitates the decoupling of the healing and purchasing mechanics from the user interface logic, and demonstrates adherence to the SRP. This is due to the fact that *EnergyDrink* objects should be only responsible for defining its unique attributes, while the *Consumable* class is responsible for healing the *Player* for user interaction and the *Purchasable* is responsible for allowing the items to be able to be purchased by the *Player*.

Moreover, *EnergyDrink* class is a subtype of *Purchasable* where it can be substituted for another *Purchasable* subtype such as *ToiletPaperRoll* and *DragonSlayerSword* without causing any issues, which demonstrates Liskov Substitution Principle (LSP). It provides implementations for the *getPrice()*, *paymentProcess()* and *purchase()* methods, as required by the *Purchasable* interface class. This principle helps manage dependencies by allowing new functionality to be added without changing existing code, thus not affecting other parts of the system that depend on the *EnergyDrink* class. While the *EnergyDrink* class adheres to LSP, it's important to note that LSP can be restrictive. It requires that subclasses behave in the same way as their parent classes. This might not always be desirable or practical, especially when the behavior of the subclass is fundamentally different from the parent class. In such cases, it might be better to use composition instead of inheritance.

Additionally, this abstraction approach demonstrates adherence to the OCP because the design allows for extension without modification to support new items or even more types of *Action* that the item can do to *Player*, by creating new *Item* subclass or overriding new *allowableAction()* method for the *Item*. This promotes code stability and minimises the risk of introducing bugs when the game design requires extension.

DragonSlayerSword Class extends WeaponItem Class (implements Purchasable Class)

WeaponItem abstract class encapsulates common attributes and behaviours shared by *DragonSlayerSword* class, promoting cohesion and minimise coupling between classes. This code delegation clearly demonstrates adherence to the SRP because each *WeaponItem* subclass is responsible for handling their own unique definition and attributes like *hitPoints* for the computation in *AttackAction* class and *price* for the computation in *Purchasable* class, while also inheriting common attributes and methods that all items should have, which is *allowableAction()* for all items, which is the *AttackAction* and *Purchasable* for now.

The abstraction of *AttackAction* and *Purchasable* demonstrates adherence to the DIP due to the loose coupling between *DragonSlayerSword* class that relies on abstractions rather than concrete implementation of the *AttackAction* and *Purchasable* in *DragonSlayerSword* class. Moreover, the implementation of *AttackAction* class and *Purchasable* class with *DragonSlayerSword* class facilitates the decoupling of the attacking and purchasing mechanics from the user interface logic, and demonstrates adherence to the SRP. This is due to the fact that *DragonSlayerSword* objects should be only responsible for defining its unique attributes, while the *AttackAction* class is responsible for attacking enemies that are hostile to the *Player* for user interaction and the *Purchasable* is responsible for allowing weapon items to be able to be purchased by the *Player*.

Additionally, this abstraction approach demonstrates adherence to the OCP because the design allows for extension without modification to support new items or even more types of *Action* that the item can do to *Player*, by creating new *WeaponItem* subclass or overriding new *allowableAction()* method for the *WeaponItem*. This promotes code stability and minimises the risk of introducing bugs when the game design requires extension.

In an alternative design without interfaces, we could directly add all the items available manually in the *Terminal* class, thereby reducing abstraction layers. However, this approach may lead to code duplication and reduced flexibility and scalability of the project as the game grows larger and more items to be purchased, violating the OCP and DRY principles. As far as the game

design is concerned, *Purchasable* interface class is introduced to allow new purchasable items like *EnergyDrink*, *DragonSlayerSword* and *ToiletPaperRoll* to implement the *Purchasable* interface to and an *ArrayList* is applied to store all the *Purchasable* instances.

ToiletPaperRoll Class extends Item Class (implements Purchasable Class)

Item abstract class encapsulates common attributes and behaviours shared by *ToiletPaperRoll* class, promoting cohesion and minimise coupling between classes. This code delegation clearly demonstrates adherence to the SRP because each *Item* subclass is responsible for handling their own unique definition and attributes like *price* for the computation in *Purchasable* class, while also inheriting common attributes and methods that all items should have, which is *allowableAction()* for all items *Purchasable* for now.

The abstraction of *Purchasable* demonstrates adherence to the DIP due to the loose coupling between *ToiletPaperRoll* class that relies on abstractions rather than concrete implementation of the *Purchasable* in *ToiletPaperRoll* class. Moreover, the implementation of *Purchasable* class with *ToiletPaperRoll* class facilitates the decoupling of the purchasing mechanics from the user interface logic, and demonstrates adherence to the SRP. This is due to the fact that *ToiletPaperRoll* objects should be only responsible for defining its unique attributes, while the *Purchasable* is responsible for allowing the item to be able to be purchased by the *Player*.

Additionally, this abstraction approach demonstrates adherence to the OCP because the design allows for extension without modification to support new items or even more types of *Action* that the item can do to *Player*, by creating new *Item* subclass or overriding new *allowableAction()* method for the *Item*. This promotes code stability and minimizes the risk of introducing bugs when the game design requires extension.

However, it is worth nothing that the abstractions layer and interfaces might increase the complexity of the code, maintenance overhead and indirections. This may cause the code to be harder to understand or maintain will lead to confusion and inefficiency because the developers may have a hard time understanding the code and debugging it may be hectic.

Conclusion

Overall, through careful analysis and rationalising of the design decisions for these four key requirements, we aim to create a cohesive and immersive gameplay experience that captivates players. By adhering to coding principles such as SOLID and DRY principles, we strive to create a robust and maintainable codebase that facilitates future extensions and enhancements to the game system. In conclusion, the chosen designs above utilise abstract and interface classes that adheres to SOLID and DRY principles more effectively, by promoting code reusability, extensibility and maintainability by encapsulating common behavior with abstract and interface classes and allowing easy integration of new gaming features without modifying the existing code.