Assignment 1: Design Rationale

Er Jun Yet | 33521026 | FIT2099 – Object Oriented Design and Implementation

19th April 2024

Introduction

In Assignment 1, we have been tasked with working on a 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 Intern of the Static Factory, where the player plays the role of the Intern in the game that can collect scraps like metal sheets and large bolts on the abandoned moon.

Req 2 - The moon’s flora, where the growth of Inheritrees in the game map happens by aging and producing fruits.

Req 3 - The moon’s (hostile) flora, where a hostile called HuntsmanSpider spawned from the moon’s craters wanders around and may attack the Intern.

Req 4 - Special scraps, where metal pipe as the Intern’s weapon and fruit as Intern’s healing source in the game.

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 Intern of the Static Factory

The first design task revolves around the *MetalSheet* and *LargeBolt* classes, which represent the regular scrap items scattered 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.

MetalSheet and LargeBolt Classes

MetalSheet class represents a metal sheet scrap scattered in the game map that is a regular scrap that can be picked up by the *Player* intern and dropped inside the spaceship. It upholds the responsibility of defining the properties of a metal sheet scrap, such as the name of metal sheet, the display character (%) and its true portability. *LargeBolt* class represents a large bolt scrap scattered in the game map that is a regular scrap as well that can be picked up by the *Player* intern and dropped inside the spaceship too. It upholds the responsibility of defining the properties of a large bolt scrap, such as the name of large bolt, the display character (+) and its true portability. This code delegation demonstrates adherence to the SRP as a *MetalSheet* object and a *LargeBolt* object focus solely on its own definition of properties that they should be only responsible for and not having any unrelated behaviour, enhances code clarity and maintainability.

As far as the object-oriented design is concerned, both *MetalSheet* and *LargeBolt* classes exhibit common key properties like *Item* abstract class in the provided *engine* package, where they can be picked and dropped by *Player* intern, thus both *MetalSheet* and *LargeBolt* classes are child classes that extend the *Item* parent class, promoting code reusability and future extensibility. This design decision adheres to the DRY principle by allowing shared functionality to be implemented in the *Item* abstract class and reused by its child classes.

Accordingly, this design approach allows *Player* class to interact with *MetalSheet* and *LargeBolt* as they extend *Item* abstract class which implements *getPickUpAction()* and *getDropAction()* methods that provide *PickUpAction* to *Player* when *Player* is on top of the specific *Item* and *DropAction* when *Player* owns the specific *Item*. The inheritance from *Item* abstract class to *MetalSheet* and *LargeBolt* child classes demonstrates adherence to the LSP, where a subclass object like *MetalSheet* or *LargeBolt* can be substituted for superclass object like *Item* without affecting the behaviour of the program. In other words, operations like *PickUpAction* and *DropAction* can be applied to instances of *MetalSheet* and *LargeBolt* without altering the expected outcome. This design ensures that subclasses can be used interchangeably with their parent classes, promoting polymorphism and modularity.

Furthermore, this abstraction approach demonstrates adherence to the OCP because the game system can be easily extended and modified to support new scrap items by extending new subclasses of *Item* parent class without modifying the base code, promoting code maintainability and extensibility. Moreover, the implementation of *Item* abstract classs provided in *engine* package also adheres the DIP, as it serve as an abstraction between different *Item* subclasses and the game system, where changes made in *MetalSheet* or *LargeBolt* or any other subclasses would not affect the game system because they are decoupled to each other.

As far as the *Item* class is concerned, *MetalSheet* and *LargeBolt* classes are only limited to *Item*-related functionalities, where they are primarily focused on defining behaviors related to scrap items. While this specification enhances clarity and maintainability, it may limit their applicability for more complex game mechanics that extend beyond item management, resulting a potential for overloading them with unrelated functionalities.

Req 2 - The Moon's Flora

Next, the second design task revolves around the *Inheritree* and *Fruit* classes that represent elements of a plant system within the game world, creating a dynamic and immersive ecosystem that adds depth and complexity to the game environment. This design is proved to have adhered the SOLID and DRY principles with the following justification below.

YoungTree and OldTree Classes extend Inheritree Class

Inheritree class represents a general Inheritree plant on the moon, while *YoungTree* class represents the sapling stage of the Inheritree plant and *OldTree* class represents the mature stage of the Inheritree plant. This code delegation clearly demonstrates adherence to the SRP because each *Inheritree* subclass is responsible for managing a specific stage of the Inheritree plant's lifecycle.

From the code implementation, the *Inheritree* parent class encapsulates common functionalities related to fruit production with *dropFruit()* method, shared by both *YoungTree* and *OldTree* classes, and centralised in the *Inheritree* parent class, adhering to the DRY principle that prevents duplication of code across multiple classes and ensures consistency and reducing the likelihood of errors.

Inheritree Subclasses produces Fruit Classes

YoungTree is responsible for handling the entire growth of sapling stage of *Inheritree* that helps in production of *SmallFruit* objects by *dropFruit()* method and its transitioning to *OldTree* by *tick()* method when it reaches its mature age, while *OldTree* is responsible for handling the entire growth of mature stage of *Inheritree* that helps in production of *LargeFruit* objects by *dropFruit()* method and its aging process by *tick()* method.

Due to the clear and evident hierarchy plant system that is determined, *Ground* abstract class in the *engine* package, introduced a new type of Inheritree terrain by having *Inheritree* class to extend *Ground* abstract class, while *Inheritree* class is also assigned to be an abstract class that is extended by *YoungTree* and *OldTree* child classes. For the fruit production, *Item* abstract class in the *engine* package, introduced a new item in the game environment by having *Fruit* class to extend *Item* abstract class, while *Fruit* class is also assigned to be an abstract class that is extended by *SmallFruit* and *LargeFruit* child classes.

The inheritance from *Inheritree* to *YoungTree* and *OldTree* child classes and *Fruit* abstract classes to *SmallFruit* and *LargeFruit* child classes demonstrates adherence to the LSP, where the subclasses like *YoungTree*, *OldTree*, *SmallFruit* and *LargeFruit* can be substituted for their respective parent class without affecting the behaviour of the program. This means that objects of the parent class can be replaced with objects of its subclasses without altering the correctness of the program. For example, *dropFruit()* method that accepts a *Fruit* object as a parameter can also accept instances of *SmallFruit* or *LargeFruit* without any issues.

Furthermore, this abstraction approach demonstrates adherence to the OCP because the design allows for extension without modification to support new plant type or new fruit type by extending new subclasses of *Inheritree* or *Fruit* parent classes without modifying the base code. This promotes code stability and minimizes the risk of introducing bugs when extending the system.

While the design promotes modularity, there is a risk of tight coupling between classes if not carefully managed. Dependency interactions between *YoungTree* with *SmallFruit* and *OldTree* with *LargeFruit* may become tightly coupled, leading to dependencies that are difficult to untangle. To mitigate this risk, it's essential to maintain loose coupling between classes and adhere to established design principles, such as the DIP.

Req 3 - The Moon's (Hostile) Fauna

The third design task revolves around the *HuntsmanSpider* and *Crater* classes that introduce a hostile creature of HunstmanSpider and its spawning habitat from the moon's crater, creating a challenging and dynamic gameplay experience that keeps players engaged and immersed in the game world. This design is proved to have adhered the SOLID and DRY principles with the following justification below.

HuntsmanSpider Class extends Hostile Class

Hostile abstract class encapsulates common attributes and behaviours shared by hostile creatures, including *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 *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 hotile type 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 *HuntsmanSpider* and perhaps other more child classes in the furture demonstrates adherence to the LSP, where the *HuntsmanSpider* subclass can be substituted for their *Hostile* parent class without altering the correctness of the program. In other words, *HuntsmanSpider* as a subclass of *Hostile* can be used interchangeably with other hostile entities wherever *Hostile* instances are expected.

SpiderSpawner Class 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 *SpiderSpawner* does not need to implement methods that it does not care about. 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 HuntsmanSpider in the *Crater* concrete class itself.

Spawner interface class is introduced to promote new spawning mechanisms beyond *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* can implement *Spawner* interface class by implementing straight the important methods needed for being *Spawner* subclasses, like *spawnHostiles()* method. *SpiderSpawner* does not need to implement any other methods that it does not care about, 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 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, potentially making it harder to comprehend when collaborating with other developers who are unfamiliar with the design.

AttackBehaviour Class implements Behaviour Class

As far as the possible game extension is concerned, *Hostile* like *HuntsmanSpider*, is a non-player character (NPC) that exhibits a wide range of behaviours, thus necessitating the importance of *Behaviour* interface class provided in the *engine* package. As all *Hostiles* are NPCs, the attacking by *Hostiles* should be assumed to be a *Behaviour* and not an *Action*, thus the *AttackBehaviour* is introduced by implementing the *Behaviour* interface class. The abstraction of *AttackBehaviour* demonstrates adherence to the DIP due to the loose coupling between *Hostile* class and *AttackAction* class that relies on abstractions rather than concrete implementation of the *AttackAction* in *Hostile* class or even *HuntsmanSpider* class.

Moreover, the implementation of the *Behaviour* interface class with *AttackBehaviour* class facilitates the decoupling of the combat mechanics from the user interface logic, demonstrates adherence to the SRP. This is due to the fact that *AttackBehaviour* objects should be only responsible for defining when the attack behaviour happens, while the *AttackAction* class is responsible for performing the *Hostile* attack to *Player* for user interaction and menu display. This separation of concerns enhances code clarity and maintainability, adhering to the ISP as well due to the usage of interface implementation.

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

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* constant, is decided to add a new constant called *HOSTILE_TO_PLAYER* that work particularly for *Hostile's* status. This enum constant benefits by enhancing code readability by providing self-descriptive status for *Hostile.* This enum constant works perfectly when our code implementation requires verification of an *Actor* if it is a *Hostile.*

Req 4 - Special Scraps

Finally, the fourth design task revolves around the *MetalPipe* and *Fruit* classes that introduce a weapon for *Player* to use for attacking a *Hostile* and fruits for *Player* to gain health, creating an interactive and interesting gameplay experience for players to be immersed in the game world. This design is proved to have adhered the SOLID and DRY principles with the following justification below.

MetalPipe Class involves AttackAction Class

As far as the object-oriented design is concerned, *MetalPipe* class exhibit common key properties and functionalities like *WeaponItem* abstract class in the provided *engine* package, where it can be picked and dropped by *Player* intern and allows *Player* to perform attack on *Hostile*, thus *MetalPipe* is a child classes that extend the *WeaponItem* parent class, promoting code reusability and future extensibility. This design decision adheres to the DRY principle by allowing shared functionality to be implemented in the *WeaponItem* abstract class and reused by its child classes.

WeaponItem abstract class encapsulates common attributes and behaviours shared by all weapons in the future, including *MetalPipe* that extends *WeaponItem* parent class, promoting cohesion and minimising coupling between classes. This code delegation clearly demonstrates adherence to the SRP because each *WeaponItem* subclass is responsible for handling their own unqiue definition and attributes like *damage* and *hitRate* for the computation in *AttackAction* class, while also inheriting common attributes and methods that all weapons should have, such as *allowableAction()* for all *MetalPipe*, which is *AttackAction* for now.

The abstraction of *AttackAction* demonstrates adherence to the DIP due to the loose coupling between *MetalPipe* class that relies on abstractions rather than concrete implementation of the *AttackAction* in *MetalPipe* class. Moreover, the implementation of *AttackAction* class with *MetalPipe* class facilitates the decoupling of the combat mechanics from the user interface logic, demonstrates adherence to the SRP. This is due to the fact that *MetalPipe* objects should be only responsible for defining its unique attributes, while the *AttackAction* class is responsible for performing the attack by *Player* onto *Hostile* for user interaction.

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

method for the *WeaponItem*. This promotes code stability and minimizes the risk of introducing bugs when the game design requires extension.

Fruit Class involves HealAction Class

Fruit abstract class encapsulates common attributes and behaviours shared by *SmallFruit* and *LargeFruit* classes, promoting cohesion and minimising coupling between classes. This code delegation clearly demonstrates adherence to the SRP because each *Fruit* subclass is responsible for handling their own unqiue definition and attributes like *hitPoints* for the computation in *HealAction* class, while also inheriting common attributes and methods that all fruits should have, which is *allowableAction()* for all fruits, which is the *HealAction* for now.

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

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

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.