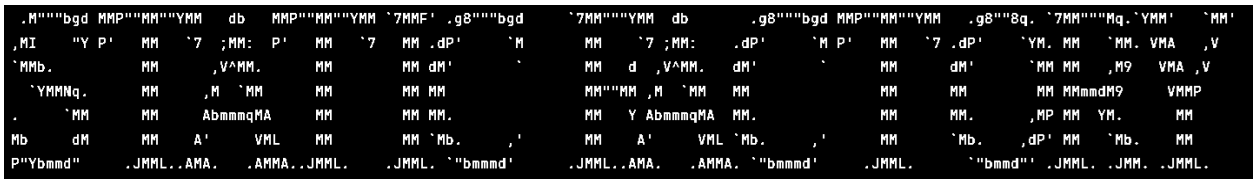


FIT2099 ASSIGNMENT 2 REPORT (DESIGN DIAGRAM AND RATIONALE)

Group: MA_AppliedSession08_Group2

Group Members:

Lee Quan Hong

Teh Jia Xuan

Hoe Jing Yang

Lee Teck Xian

REQ1: The moon's (hostile) fauna II: The moon strikes back

Figure 1 Class Diagram of REQ 1.

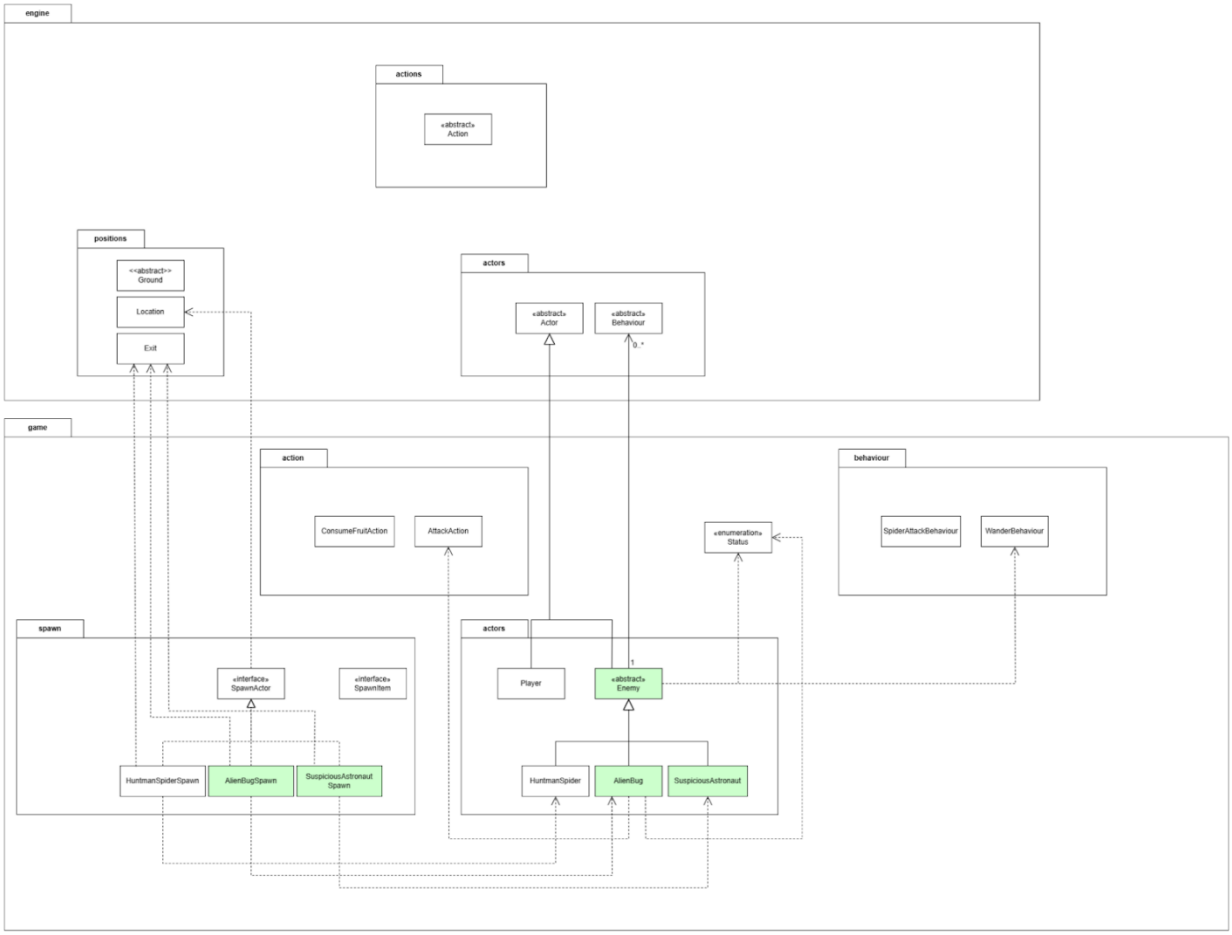
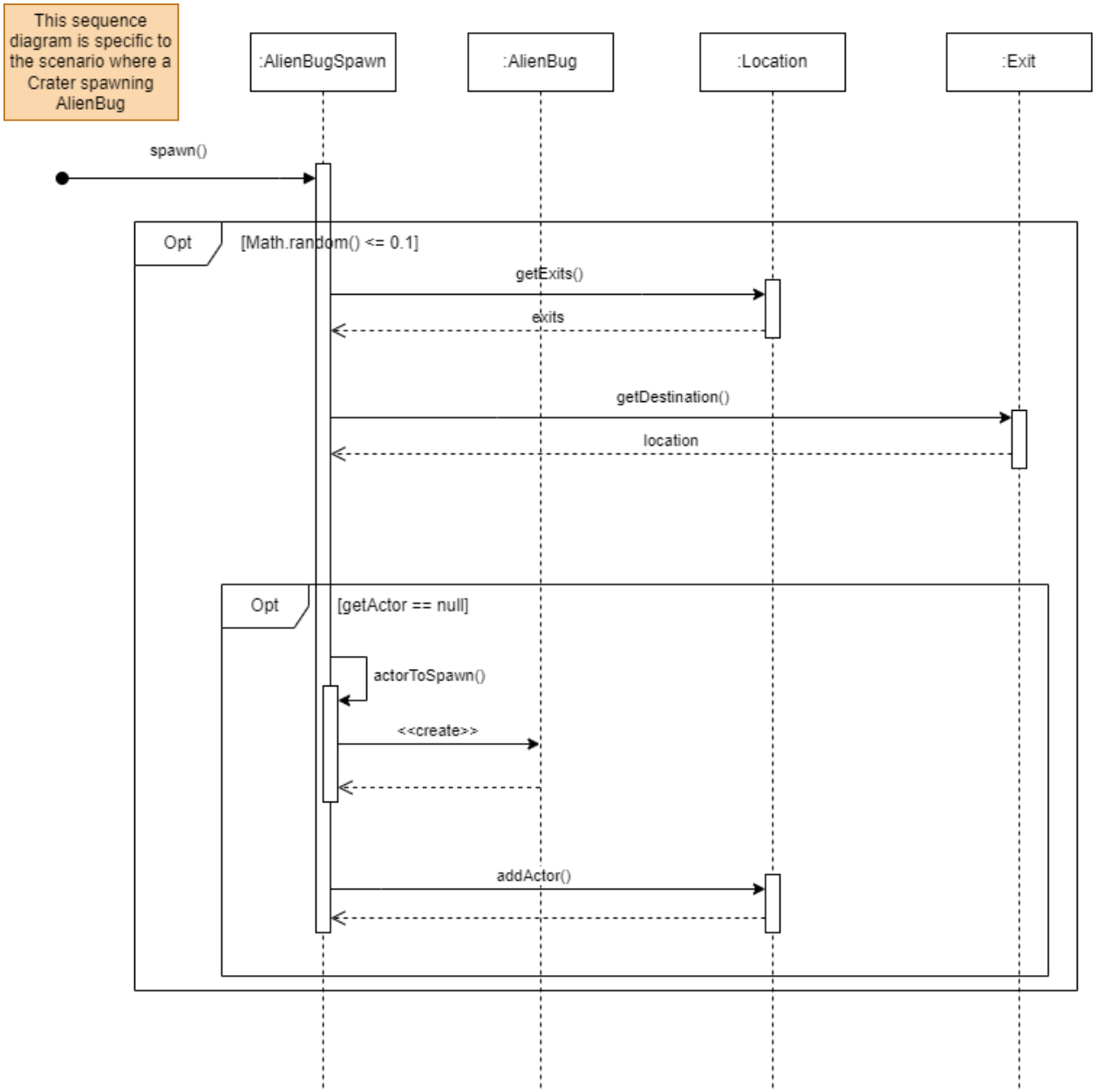


Figure 1 Sequence Diagram of REQ 1, Crater spawning AlienBug.



Design Goal:

The design goal for this assignment is to add two more creature that can be spawn using the crater from assignment 1 with a different spawn rate and different behaviours while adhering closely to relevant design principles and best practices.

Design Decision:

In implementing the function to add more actors to the game, the decision was made to create an interface dedicated to actor that spawns from the ground(crater) and also create a spawn class for each spawned actor that handles their spawn logic. By having a spawn interface, we get to standardize all the methods that the spawn actor going to use which makes it easier to add more actor in the future. The spawn class dedicated to each actor which implements the interface handles each of their spawn rate and spawn logic makes the code cleaner and allows the crater to spawn different actors easily.

Alternative Design:

One alternative approach could involve not using an interface and abstract class but cramp everything into the actor class itself like AlienBug class but include everything like the spawn logic and etc into the class itself. This method might work but it would make the code unstandardised which might create problems like error from copy-paste, code doesn't fit into previous working one or difficult for future extension. Besides that, cramping everything into a class violates various Design and SOLID principles which makes reading and debugging harder.

Analysis of Alternative Design:

The alternative design is not ideal because it violates various design principles including SOLID and DRY:

1. **Single Responsibility Principle:**
    - By simply cramping all the methods into one class, we violate SRP as its stated that one class should only hold one responsibility.
2. **DRY:**
    - By cramping everything into one class and without the usage of various abstract classes, we would violate the DRY principle by writing too many redundant code as DRY means Don't Repeat Yourself.
3. **Open-closed Principle:**

- Without the usage of interface and abstract classes, this violates the open-closed principle as its hard for extension and requires a lot of modification if new features are introduced to the code.

Final Design:

In our design, we closely adhere to closely adhere to SOLID and DRY.

1. **Single Responsibility Principle:**

    - **Application:** By having a dedicated spawn class for each actor like AlienBugSpawn and SuspiciousAstronautSpawn, each spawn class only handles one spawn criteria for their actor which match the definition of single responsibility principle where the spawn class only handles the spawn criteria of their dedicated actor.
    - **Why:** We don't want a class to hold too many responsibilities as it will makes it harder to maintain.
    - **Pros/Cons:** The benefit of implementing this principle is that when error occurs, it is very easy to pinpoint the class that cause the error.

2. **Open-closed Principle:**

    - **Application:** Instead of defining all the methods for each actor, we created an interface name SpawnActor. Every actor that can be spawn from Crater needs to implement this interface. This implementation follows the principle by open for extension but close for modification.
    - **Why:** By adhering to this principle, we are able to standardize our code which reducing errors and make it easier for future extension.
    - **Pros/Cons:** The benefit for implementing this principle, we can add more spawn actor to the game easily without needing to rewrite the codes every time when we need to add a new actor. Also, this reduces the risk of adding bug to the previously working code.

3. **DRY:**

    - **Application:** The newly added AlienBug and SuspiciousAstronaut extends from the Enemy class which also extends from Actor. The Enemy class only overrides needed method from the Actor and add certain new method for the actor type. The newly added actor inherits method from their superclass that are specific for them without repeatedly adding code into it which adheres to the DRY principle.
    - **Why:** By implementing the DRY principle, we are able to design a code with fewer redundancy and higher maintainability.

- **Pros/Cons:** We don't need to rewrite methods for actor every time we want to add a new actor, instead we can just inherit it from the abstract class and override the methods that we need to make changes. This allows us to avoid error that can happen when copy-pasting codes.

How it can be easily extended:

To add a new enemy, it only needs to extend the Enemy abstract class and while the spawn class dedicated to the enemy only needs to implement the spawnActor interface class. The spawn criteria can be easily modified based on the requirements while the characteristic of the enemies can also be override from the abstract class if necessary. This provides an easy and standardize way that allows anybody to add more enemy in the future.

Conclusion:

Overall, our chosen design provides a robust framework for adding new actors and makes it very easy for future extension. By carefully considering various design principles like SOLID and DRY and also the requirement of the assignment, we have developed a solution that is easy to scale, easy to maintain/debug and very efficient for future extension. This paves the way for further enhancement, adding new feature and the ease of optimization.

REQ2: The imposter among us

Figure 3 Class Diagram of REQ 2.

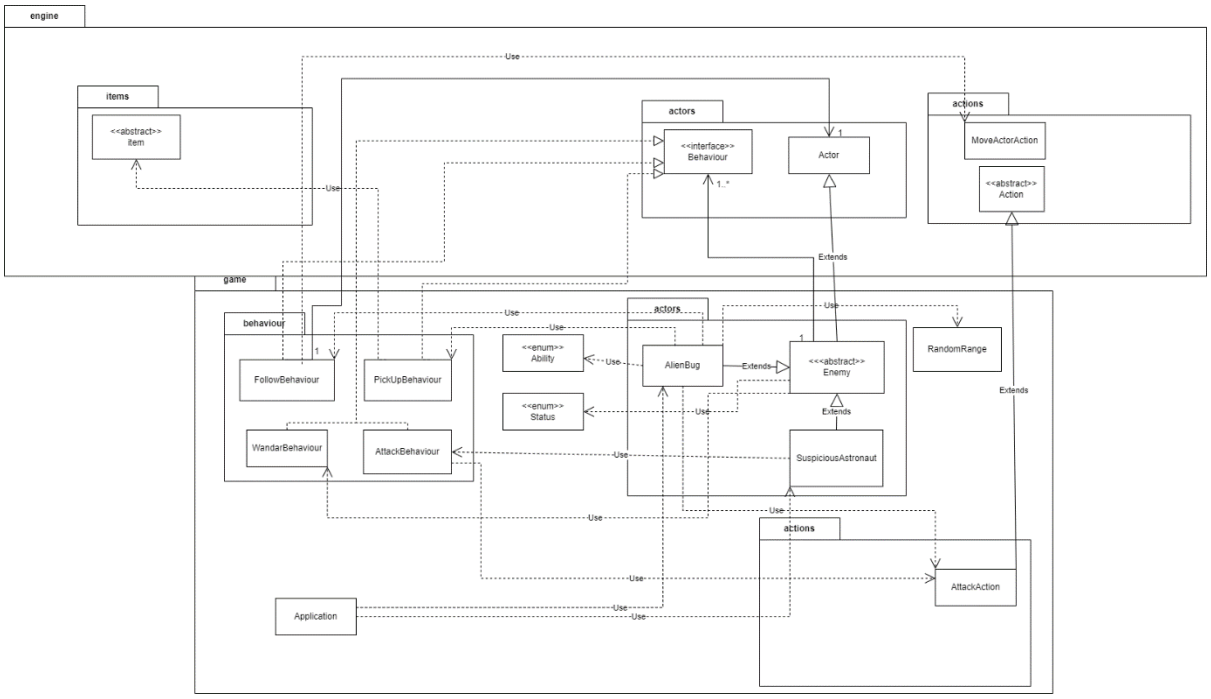
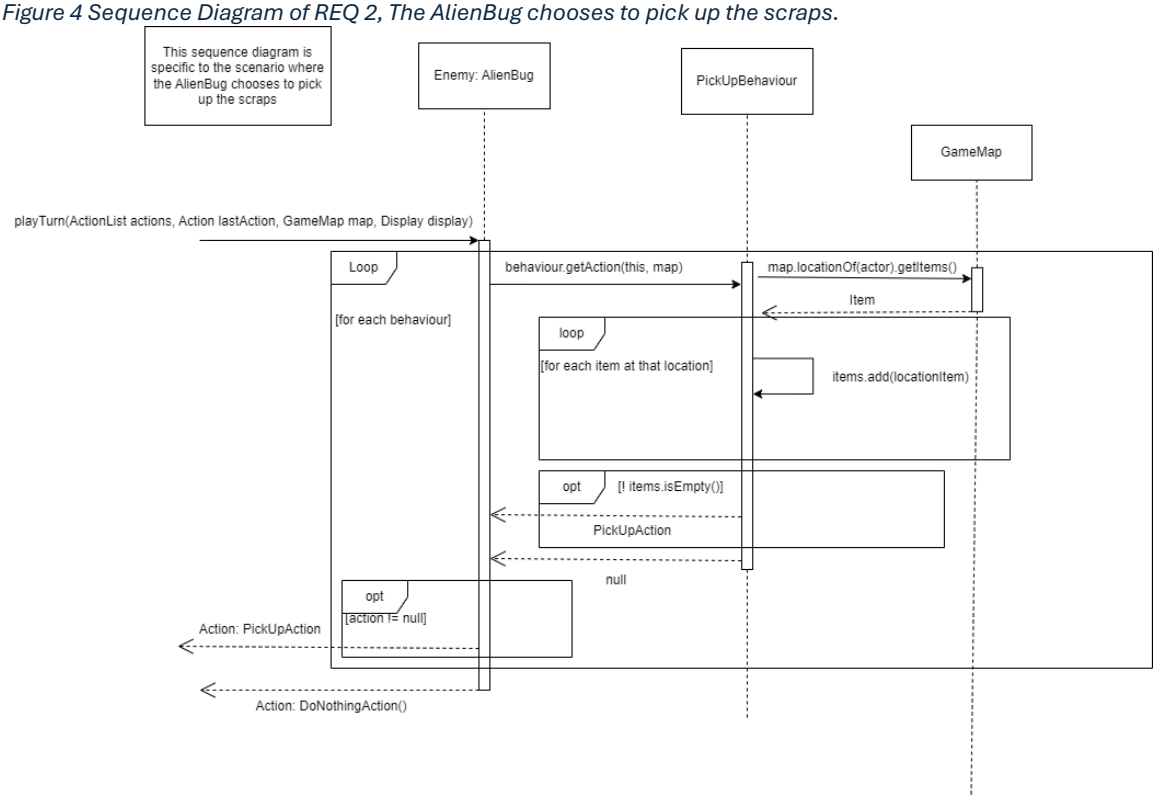


Figure 4 Sequence Diagram of REQ 2, The AlienBug chooses to pick up the scraps.



Design Goal:

The design goal is to implement the new behaviour to SuspiciousAstronaut and AlienBug with high maintainability and easy for extension in the future.

Design Decision:

We created an enemy abstract class for all the player's enemy to inherit. Since all enemy has Wandering Behaviour so we added WanderBehaviour into enemy abstract base class, which then can be extend to those enemy actors. By letting SuspiciousAstronaut able to attack the player we simply add AttackBehaviour into its behaviour and we override the weapon to determine the damage and hit rate. In this design it makes our code more efficiency as we able to utilise the AttackBehaviour that we created in our previous implementation. Furthermore, we have created a RandomRange for the AlienBug's name. We can also reuse this with other feature in the future. Thus, we can avoid repetition of code.

Similarly, to enable AlienBug to Pick up scraps and follow the player we developed PickUpBehaviour and FollowBehaviour integrating them into AlienBug's behaviour. Moreover, the AlienBug is able to enter the spaceship, so we just added "able to enter spaceship" ability in the Ability class and AlienBug add to its capability. The decision was made to let the code understandable and easy for extension in the future. For example, if there is a creature needed these behaviours in the future, we can simply add it into their behaviour. Similarly, if there is creature that allows to enter the spaceship it can add the ability into its capability. Therefore, this design has high extensibility and is preventing code repetition.

Alternative Design:

One alternative approach could involve creating the behaviour method in the SuspiciousAstronaut and AlienBug. However, when there are more behaviours, the class will become lengthy and hard to maintain. Not to mention, there will be repetitive code when there is same behaviour of the creatures as we need to implement in their own class. But the advantage will be the code can be easily understood as all the behaviour related to this creature is in its own class.

Not to mention, initially our design for SuspiciousAstronaut attack behaviour was creating another KillAttackBehaviour that killed player instantly. The disadvantage of this is there will be repetitive code in the behaviour which can get complicated when there are two attack behaviour class.

Besides that, an alternative approach for the functionality of entering spaceship is overriding the floor class when the creature has capability of Hostile to Player then it

allows the creature to enter the floor. But it doesn't work when there is more than one creature and the other creature could not enter the floor.

Analysis of Alternative Design:

The alternative design is not ideal because it violates various Design and SOLID principles:

1. **Single Responsible Principle:**
   - In the design since we have all the behaviour in one class, it is breaking the single responsible principle as one class only can have one responsiblility

2. **Open-Closed Principle:**
   - When we need to add new behaviour, we need to modify the method to accommodate the new behaviour so it will be violating the open-closed principle as it mentioned that when we are adding new functionality, we shouldn't be modifying the previous code.

3. **Dependency Inversion Principle:**
   - Since the class is putting all behaviour together, it will have multiple dependency towards the concrete class and it will break this principle as it says concrete class should depends on abstract class instead of another concrete class.

4. **Don't Repeat Yourself**
   - Since there are two attack behaviour. Thus, there will be similar code in the attack behaviour, hence violating this principle.

Final Design:

In our design, we closely adhere to SOLID and DRY principles:

1. **Don't Repeat Yourself (DRY):**
   - **Application:** Add wander behaviour to enemy abstract class for the player's enemy to extension to previous redundancies of wandering behaviour in all enemy class. Moreover, we added new behaviours like PickUpBehaviour and FollowBehaviour.

   - **Why:** Whenever there is a new enemy, we can simply inherit it from the enemy class and it has wander behaviour automatically. Similarly, whenever there is a creature has pick up behaviour or follow behaviour, we can simply add the behaviours to their capability that we created in our previous implementation. Thus, we won't be having any redundant code.

- **Pros/Cons:** The pros of this design are it promotes code reusability and reduce redundancy of the code as every enemy can inherit the enemy abstract class and add the behaviours. Moreover, introducing new enemies becomes simpler and more efficient as they automatically gain the necessary behaviours and method without additional code. Similarly, it promotes consistency by centralising common behaviours like wander behaviour can maintains a consistency across different enemy types. Furthermore, it reduces the risk of errors and makes it easier to debug.

  The cons will be inheriting behaviours from a base class can limit the ability to customize behaviours for specific enemies. Besides that, adding multiple behaviours to a single enemy can increase complexity.

2. **Single Responsibility Principle:**
   - **Application:** Each class has its own responsibility. In this design each class only assigned with one responsibility.

   - **Why:** The FollowBehaviour is solely accountable for following actions, while the PickUpBehaviour is exclusively responsible for pickup actions. The SuspiciousAstronaut class and AlienBug class are dedicated to their essential methods without overlapping responsibilities.

   - **Pros/Cons:** The pros of this design are since each class has its own responsibilities the codebase becomes more modular and easier to manage. Furthermore, by having separate responsibilities it enhances code readability and makes it clear which part of the code is responsible for which action. Similarly, the design allows for reusing behaviours in different classes since each behaviour class has single responsibility this allow us to reuse the behaviour in different classes and know each responsibility of the behaviour class.
     The con of this design is since every behaviour needs to have its own class so there will be multiple classes and make the codebase harder to maintain.

3. **Open-Closed Principle:**
   - **Application:** Rather than incorporating the behavior method within their individual classes, we separate them by assigning each behavior its own class. Moreover, for the enemy that unable to enter the spaceship we add ability enter spaceship in the ability enum class. Furthermore, since SuspiciousAstronaut kill the player instantly, we can simply put integer max value into AttackBehaviour's AttackAction of SuspiciousAstronaut. Similarly, we created a random range class for the alien bug's name

- **Why:** This approach allows us to easily integrate new functionalities directly into their respective behavior classes. Consequently, we avoid the need to modify the code within the actor's class to accommodate these new functions. Similarly, when there is new actor that have the ability to enter spaceship, we simply add it to its capability. Thus, we no need to modify the floor function every time we have a new actor. Additionally, putting integer max value into SuspiciousAstronaut's AttackBehaviour AttackAction rather than creating other behaviour like KillPlayerBehaviour to prevent overcomplicated in creating too many different of classes. Here we are reusing AttackBehaviour class and add it into SuspiciousAstronaut's constructor. Likewise, if there are other functionality needs random range it can reuse the class.

- **Pros/Cons:** The pros will be easier to manage and understand the codebase as separating behaviours into their own classes. Furthermore, it enhances reusability. The individual behaviour classes can be reused across different actors which reduces code duplication. Likewise, adding new functionalities becomes more straightforward as they can be integrated into their respective behaviour classes without modifying other's code.

  The con is managing multiple behaviour classes alongside actor classes can add complexity to the overall system. Additionally, this design is kind of hardcoded. As we just put integer max value damage into AttackAction, rather than creating a new Behaviour class that has KillPlayerAction which then execute unconcious() method upon player approaching to the SuspiciousAstronaut's surrounding.

4. **Liskov Substituition Principle**
   - **Why:** since I did not use any downcasting, the behavior of the base class which is enemy remain consistent and its child class can replace their parent class without changing the behavior of the program

   - **Pros/Cons:** The pros are by avoiding downcasting I can ensure the behaviour of the base class remains consistent across all instances and subclasses. Moreover, downcasting can introduce potential errors or unexpected behaviours. Not using downcasting will increase the robustness of the codebase. Furthermore, not using downcasting it leads to a cleaner and more straightforward code structures, making it easier for developers to understand.

     The cons are without downcasting, we have experience limitations in our implementation as some design patterns may need downcasting for extensibility. Likewise in certain scenarios avoiding downcasting may need alternative solutions that can introduce additional complexity to the codebase.

5. **Dependency Inversion Principle**
    - **Why:** In my design my wander behaviour dependent on enemy abstract class. So, it does not violate the dependency inversion principle as it dependent on abstract class instead of another concrete class

    - **Pros/Cons:** By depending on abstractions rather than concrete class it reduces coupling between components and make the codebase more resilient to changes

      The cons will be in some cases the existing codebases may not align perfectly with the principle so it leads to challenges when implementing.

What Has Been Done and Why:

SuspiciousAstronaut kill player instantly

- The SuspiciousAstronaut has the AttackBehaviour and the getIntrinsicWeapon() is override when the new damage and hit rate. So that it can kill player instantly.

Added FollowBehaviour

- FollowBehaviour is added so the actor who added this into its behaviour able to follow specific actor when the actor is around.

Added PickUpBehaviour

- PickUpBehaviour is added so the actor can pick up the scraps when it is standing on it.

AlienBug added follow and pick up behaviour

- The FollowBehaviour and PickUpBehaviour is added to AlienBug behaviour, the AlienBug will prioritise picking up the scrap when it is standing on one. Then after picking up, the AlienBug will follow the player until it dies.

AlienBug can enter spaceship

- Add ability enter spaceship in Ability enum and add to AlienBug capability

AlienBug random range name

- Create a random range class for the AlienBug name

Overall Pros and Cons of the Design:

Following the SOLID principles in the implementation enhances code debuggability and extendability. The utilisation of abstraction and inheritance minimises redundancy and promote code reusability and extensibility.

The cons will be when there are many behaviours added in the future, we will be having large number of behaviour classes and it increases the complexity of the code and it is harder to maintain.

How it can be easily extended

- If we need to add more behaviour, we can create the respective behaviour class and add to the actor's behaviour.

Conclusion:

Overall, our chosen design provides a robust framework for implementing this requirement. By carefully considering relevant factors such as design principles, future extension and constraints we have developed a solution that is efficient, maintainable and extensible. This made a foundation for future enhancements, extensions and optimizations of the program.

REQ3: More scraps

Figure 5 Class Diagram of REQ 3.

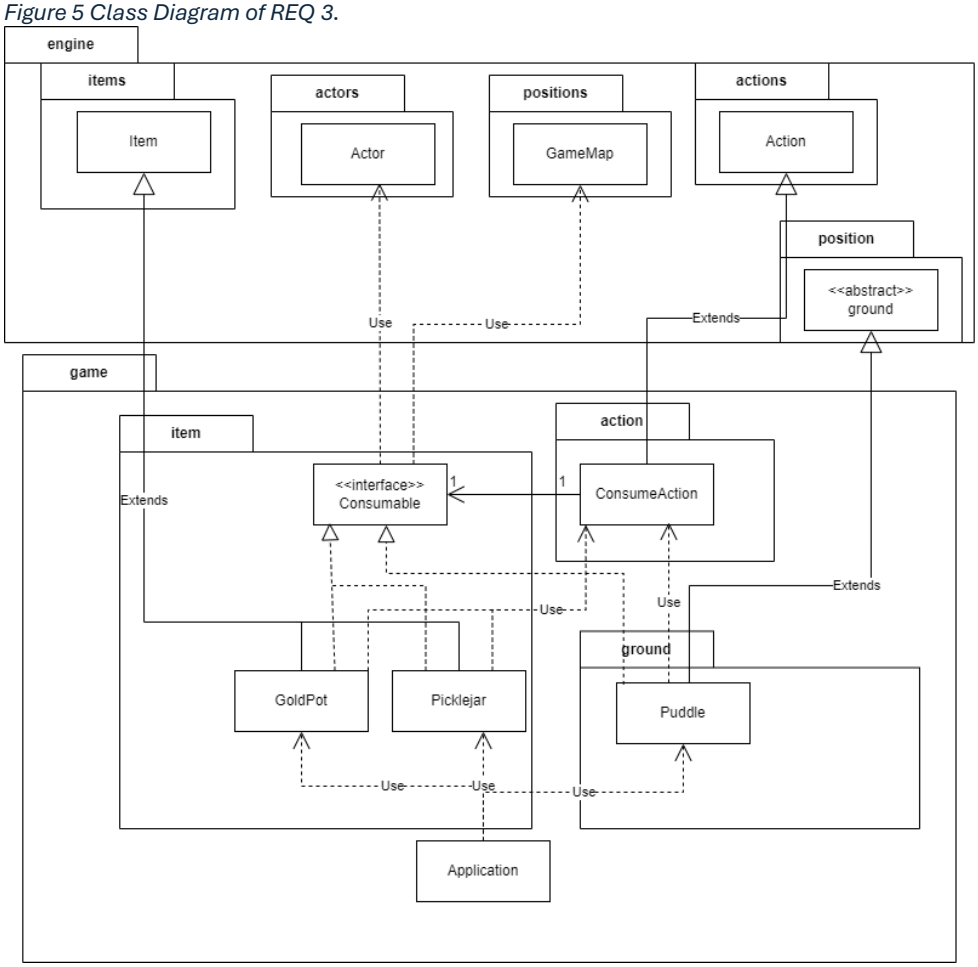
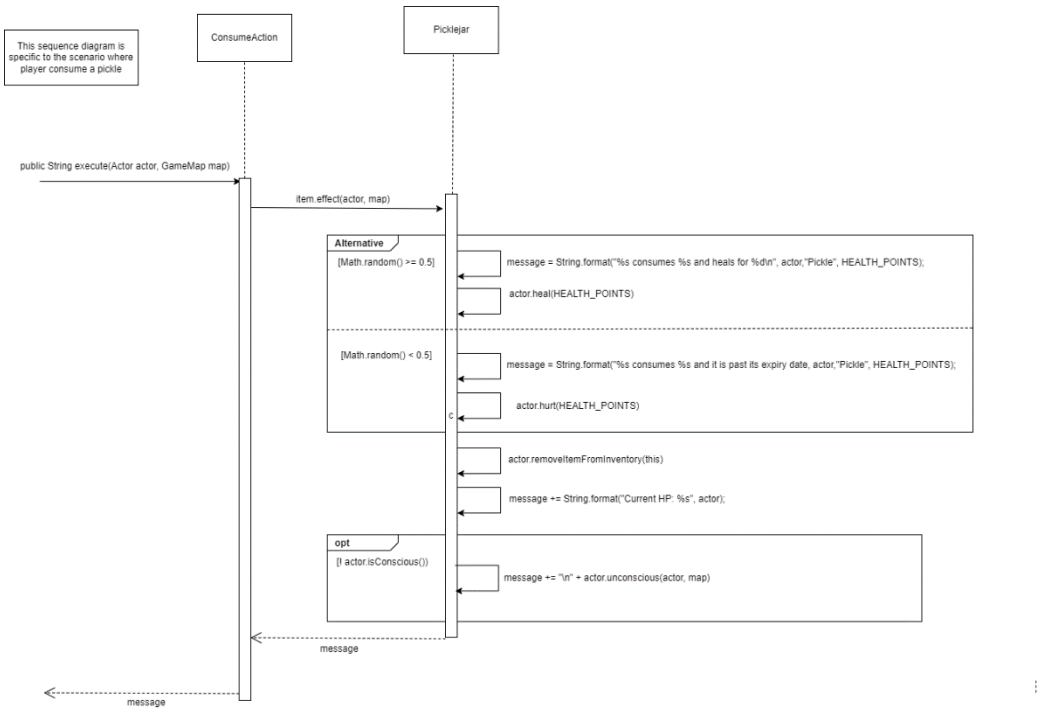


Figure 6 Sequence Diagram of REQ 3, When player consume a pickle.



Design Goal:

The design goal for this assignment is to make implementing new scraps to the game easily while adhering closely to relevant design principles and best practices.

Design Decision:

When implementing the features for puddle, goldpot, and picklejar, I made the decision to convert the "consumable" concept from an abstract class to an interface. This shift was necessitated by the fact that puddle already extends from the "ground" class, and since a class can only have one parent class, it made sense to utilize the consumable functionality through an interface. This way, consumable acts as an optional add-on feature for the item, making extension much simpler and more straightforward. Additionally, each item has its own specific action that occurs when the player consumes it. To handle this, I created a ConsumeAction class that extends from Action and accepts a Consumable object. Within its execute method, it performs the appropriate action based on the consumable item being used. For instance, in the case of the goldpot, when the player consumes it, the ConsumeAction will call the effect method in goldpot class and add $10 to the player's balance. This design eliminates the need to create multiple action classes tailored to each consumable item.

Alternative Design:

An alternative approach might be to create a separate "consume" action for each distinct consumable item and then add these actions to the list of allowable actions for each item. However, this method will increase the code complexity when there are multiple consumable item to be added in the future. It causes the code more complex and harder to maintain. Besides that, there might be redundancy of the code and inefficiency when creating consume action for each of the item. A part from that, the advantage having dedicated actions for each consumable item can make the code more understandable as each consumable has its own action.

Analysis of Alternative Design:

The alternative design is not ideal because it violates various Design and SOLID principles:

1. **Do not repeat yourself (DRY):**
   Due to having each action class for each consumable there will be redundant code as the action will be almost similar.

Final Design:

In our design, we closely adhere to the SOLID principle and DRY principle

1. **Do not repeat yourself:**
    - **Application:** Each consumable item has its own unique action method within its individual class. For example, puddle has its own method called effect that when player drinks the puddle the ConsumeAction it will execute its own effect which is add player's maximum health.

    - **Why:** This setup allows us to utilise a single ConsumeAction class and enable each consumable item to execute its own action. Instead of each consumable item create its own action class and that will cause repetition of code

    - **Pros/Cons:** The pros of this design will be when we are adding new consumable items in the future, we just need to implement Consumable class and define its own action method without requiring any additional classes. So, the ConsumeAction class promoting code reusability.

      Cons of this design is when the consumable item having a large effect then the code of the class will become lengthy and harder to maintain.

2. **Single Responsible Principle:**
    - **Application:** Each class is assigned its own responsibility. In my design, each class is designed with a single specific responsibility.

    - **Why:** The Consumable interface is meant for consumable items to implement, requiring them to implement the consumable method to define their unique effect and description. This allows consumable functionality as an optional addition for classes that wish to be consumable. Additionally, every consumable item is responsible for defining its own consumption effect by the player. Furthermore, the ConsumeAction is responsible for enabling consumable items to execute their effects.

    - **Pros/Cons:** The pros will be each class has its own specific purpose, making the codebase easier to understand and maintain. Moreover, it promotes reusability as every class have their own responsible and we can reuse it for other code when we need to implement similar functionality in the future. Furthermore, unit testing and debugging will become more straightforward as each class can be tested in isolation.

      The cons are there will be large number of classes in the system as having separate class for each responsibility it might increase complexity of the codebase. Moreover, when having strictly single responsibilities for each class

might lead to some redundancy or duplication of code if similar functionality with different item.

3. **Liskov substitution principle:**
   - **Why:** since I did not use any downcasting, the behavior of the base class which is item remain consistent and its child class can replace their parent class without changing the behavior of the program

   - **Pros/Cons:** liskov substitution principle promotes codebase be replaced without causing unexpected side effects or breaking existing code.

     Liskov substitution principle may impose constraints on how subclasses can extend or modify the behavior of their based classes which could limit certain design choices.

4. **Open-closed principle:**
   - **Application:** Instead of including each consumable item's effect method within the ConsumeAction class, each consumable item has its own effect method within its respective class.

   - **Why:** This approach allows for easier addition of new effects, simply adding them to the method of the specific consumable item class instead of making multiple modifications to ConsumeAction whenever a new consumable item is added.

   - **Pros/Cons**: Placing each consumable item's effect method in its own class promotes a modular design. Each item's behaviour is in its own class so it makes it easier to understand. Besides that, we can avoid problems like modifying ConsumeAction whenever a new consumable class is added. As in my initial design my ConsumeAction class handles all the effect of consumable item but when a puddle which is a ground was added to the implementation. I need to modify my ConsumeAction or create another new class of ConsumeAction as puddle can't be remove from the inventory since it is not an item. Now in my current design I able to use only one ConsumeAction to accommodate the need of all consumable item. This makes the codebase more simply and easy to implement

     For the cons is if multiple consumable items share similar effects there may be some redundancy in code as each item's effect method is separate. This can lead to duplicated code.

5. **Interface Segregation Principle:**
   - **Why:** I have different interfaces for different types of purpose. For example, Consumable interface is implemented for the consumable item so if the item is consumable then it can implement the interface. We ensured that each

class implements only the methods that are directly relevant to its functionality. This followed the Interface Segregation Principle.

- **Pros/Cons:** By having separate interfaces for different types of functionality so that I can ensure each interface is focused on a specific purpose. This promotes clearer and organized design. Moreover, since classes only need to implement interfaces relevant to their functionality, it can reduce unnecessary dependencies. Furthermore, the maintenance becomes easier. Changes or update to a particular feature can be made in the relevant interface.

  The cons will be when there are multiple interfaces it may lead to a larger number of interfaces in the system which can make the interface hierarchy more complex and harder to manage. Moreover, developers need time to familiarise themselves with the interfaces and their specific purposes when there are numerous interfaces.

What Has Been Done and Why:

Picklejar created player able to consume

- Picklejar extend item and implement consumable interface to make it consumable. There is effect method in picklejar it stated the effect to the player which is heal or hurt on the probability of 50 50 and remove from player's inventory after that print an appropriate message when the player consumes and there is ConsumeAction added to Picklejar allowable action. So, it can be consumed and execute the respective effect.

GoldPot created player able to consume

- GoldPot extend item and implement consumable interface. The effect method is adding player's balance by 10 and remove from player's inventory after that print an appropriate message. Moreover, the ConsumeAction is added to its AllowableAction so it able to execute its effect after player consume.

Puddle created player able to consume

- Puddle extend ground and implement consumable interface. The effect method is player able to add 1 to their maximum health and print a appropriate message. Similarly, the ConsumeAction is added to its AllowableAction so it able to execute its effect after player consume.

Overall Pros and Cons of the Design:

Following the SOLID principles in the implementation enhances code debuggability and

extendability. The utilisation of abstraction and inheritance minimises redundancy and promote code reusability and extensibility.

The cons will be when the item have many effects on player it might get lengthy code in the class. Hence, it will increase the complexity of the code and harder to maintain

How it can be easily extended

- If there is a new consumable item, it simply needs to extend item abstract class and implement consumable to get all the necessary method to be consumed by the player. Not to mention, the item can implement its own effect method and add ConsumeAction to its allowable action This makes the codebase easy to understand.

Conclusion:

Overall, our chosen design provides a robust framework for implementing all the new features as well as extending them. By carefully considering the SOLID principle and not violating the principle we have developed a solution that is efficient and easy to understand. So we can add new functionality to the system easily in the future.

REQ4: Static Factory's Staff Benefits

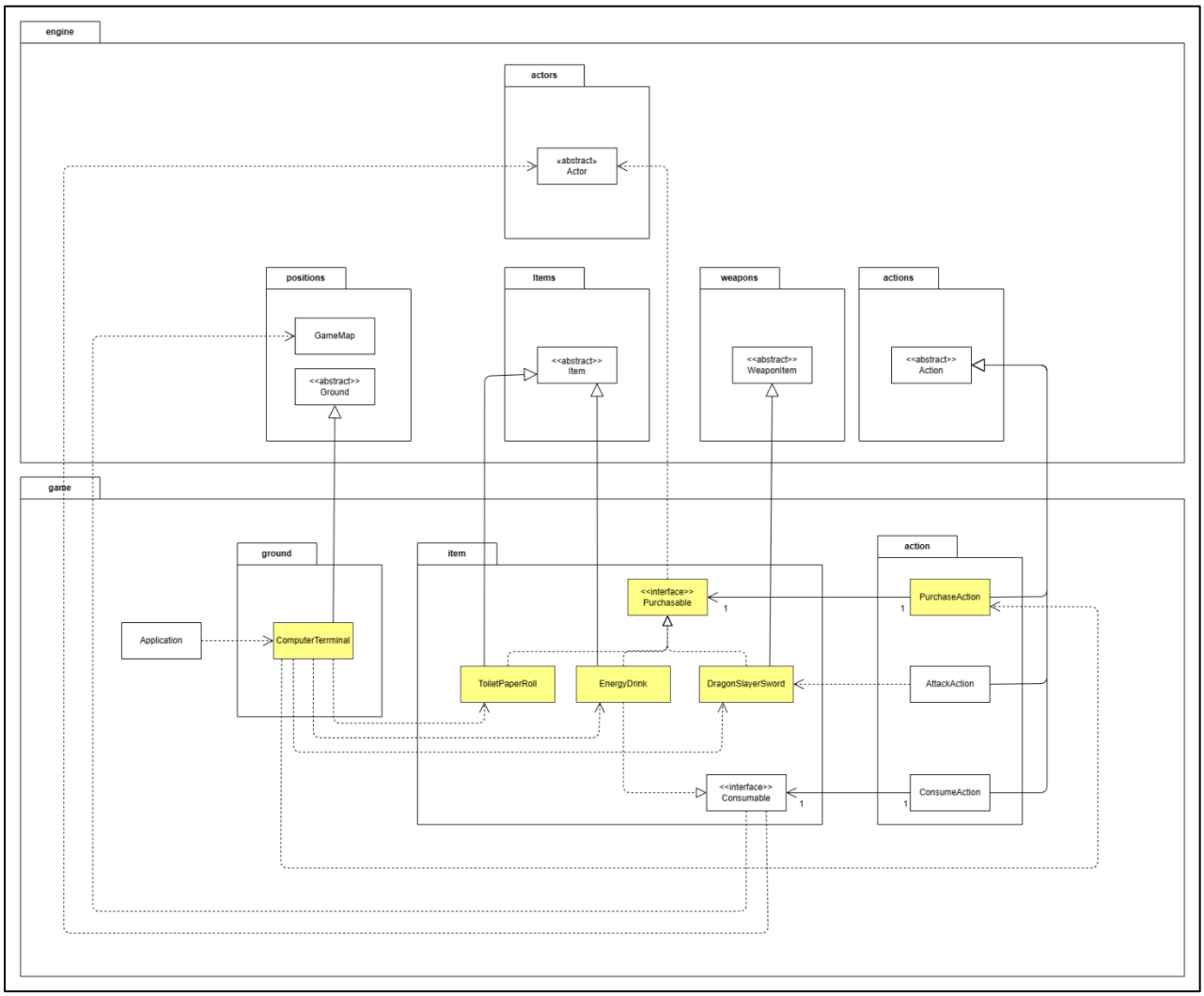


Figure 7 Class Diagram of REQ 4.

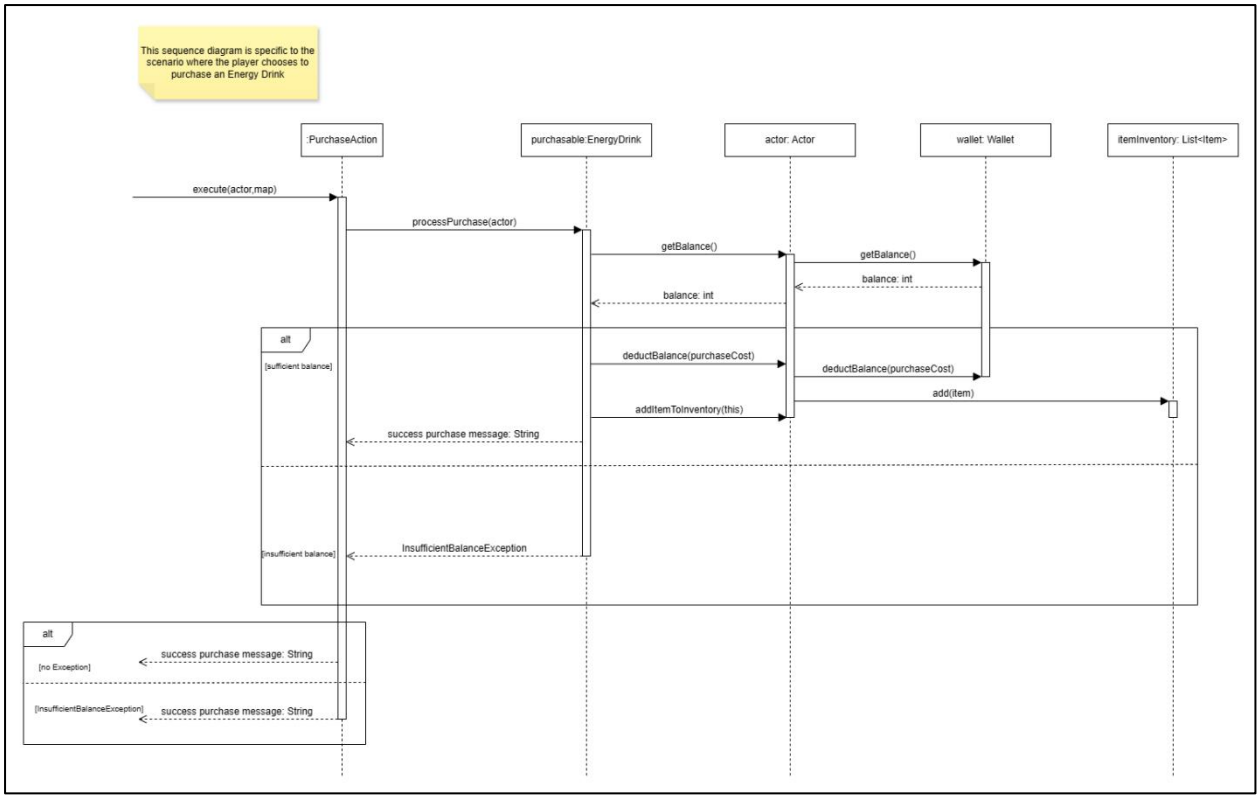


Figure 8 Sequence Diagram of REQ 4, intern purchases an Energy Drink.

Design Goal:

The design goal for this assignment is to meet the user requirements while adhering closely to relevant design principles and best practices of object-oriented software designs.

Design Decision:

In implementing REQ4: Static factory's staff benefits, the decision was made to create three new classes, EnergyDrink, DragonSlayerSword, and ToiletPaperRoll, where EnergyDrink and ToiletPaperRoll extend the Item class, and Dragon Slayer Sword extends the WeaponItem class. These classes implement the Purchasable interface to add the capability to be purchased. The Purchasable interface has two methods, processPurchase and getBaseCost, which are implemented by the classes that require them. The ComputerTerminal class extends the Ground class and provides the actions to purchase a Purchasable available in the ComputerTerminal. The PurchaseAction class extends the Action class, processes the purchase of a Purchasable and catching the potential InsufficientBalanceException when balance is insufficient.

With the adherence to the SOLID principles, the implementation of REQ4 is designed to be efficient, scalable, and maintainable. The use of abstraction and inheritance reduces

redundancy and facilitates reusability and extensibility of the code. This design decision allows the code is easy to understand, maintain and extend in the future.

Alternative Design:

One alternative design approach could involve using a ComputerTerminal class that directly processes the purchase of items without the need for the Purchasable interface. However, this violates SOLID principles.

Analysis of Alternative Design:

The alternative design is not ideal because it violates various Design and SOLID principles:

1. **Single Responsibility Principle (SRP):**
   - The alternative design would violate the SRP by combining the responsibility of processing purchases for different items into a single class. This would make the class less cohesive and harder to maintain.
2. **Open-closed Principle (OCP):**
   - The alternative design violates the OCP as it does not allow adding new items that can be purchased without modifying the existing ComputerTerminal class. This leads to a violation of the OCP as the class is not open for extension without modifying the existing ComputerTerminal class.
3. **Dependency Inversion Principle (DIP):**
   - The alternative design violates the DIP by depending on concrete classes directly for processing item purchases. This leads to a violation of the DIP as the class is not dependent on abstractions that can represent the different items that can be purchased.

Final Design:

In our design, we closely adhere to SOLID and DRY.

1. **Single Responsibility Principle (SRP):**
    - **Application:**
        - The new classes, EnergyDrink, DragonSlayerSword, and ToiletPaperRoll, are only responsible for a single part of the functionality, which they inherit from their super class.
        - EnergyDrink and ToiletPaperRoll represent a physical object and handle the responsibility of an item, such as the ability to be picked up/dropped.
        - DragonSlayerSword represents an item that can be used as a weapon.
        - The ComputerTerminal class is only responsible for providing the actions to purchase a Purchasable available in the ComputerTerminal.
        - The PurchaseAction class is only responsible for processing the purchase of a Purchasable, and catching the potential InsufficientBalanceException when balance is insufficient.
    - **Why:**
        - Adhering to the SRP makes the code easier to understand, maintain, and extend. Each class has a clear and distinct responsibility, only handles a single concern individually.
    - **Pros/Cons:**
        - The benefits of adhering to the SRP include improving the readability, maintainability and extensibility of the codebase.
        - The downside could be the need to create multiple classes for different responsibilities, which may increase the number of classes in the codebase, potentially leading to more complex interactions between classes.
        - In this case, the pros outweigh the cons as the increase in classes is manageable and provides clear separation of concerns.
2. **Open-closed Principle (OCP):**
    - **Application:**
        - ToiletPaperRoll and EnergyDrink extending Item, and DragonSlayerSword extending WeaponItem are following OCP.
        - We can extend the code functionality, adding different items to the game with different extra functionalities, without modifying the existing Item class.
        - With abstraction, using the portable attribute with the getPickUpAction and getDropAction in Item class, we can determine whether the Item can be picked up or dropped, without explicitly checking the class using instanceOf in the parent class.

    - ComputerTerminal class extends the Ground class and adds the functionality to allow intern to purchase a Purchasable from it, without modifying the existing Ground class.
    - PurchaseAction class extends the Action class and adds the functionality to process the purchase of a Purchasable, without modifying the existing Action class.
- **Why:**
    - Adhering to the OCP allows for the code to be extended without modifying existing classes by using abstraction. This makes the code easier to maintain and extend in the future.
- **Pros/Cons:**
    - The benefits of adhering to the OCP include preventing the need to modify existing classes when adding new functionality.
    - The downside could be the need to create more abstract classes or interfaces as an abstraction, which may add complexity to the codebase.
    - In this case, the pros outweigh the cons as using abstractions allows for easier extension and maintenance of the codebase, without having to modify existing classes and potentially introducing bugs.

3. **Liskov Substitution Principle (LSP):**
    - **Application:**
        - ToiletPaperRoll, EnergyDrink did not override any existing methods from Item, thus will not change any meaning of the methods.
        - DragonSlayerSword overrides the allowableActions method from Item, the method still has the same meaning as in the super class where it returns the allowable actions.
        - ComputerTerminal overrides the allowableActions method from Ground, the method still has the same meaning as in the super class where it returns the allowable actions.
        - PurchaseAction overrides the execute method from Action, the method still has the same meaning as in the super class where it executes the action and returns a string as a result.
        - The inherited methods remain consistent with the super class.
    - **Why:**
        - Adhering to the LSP ensures that the inherited methods maintain their meaning in the subclasses. This means having consistent behavior across different classes and following the contracts, facilitating future extensions and maintenance.
    - **Pros/Cons:**
        - The benefits of adhering to the LSP include ensuring that the inherited methods are consistent and maintain their meaning in the subclasses.

    - There are no significant downsides to adhering to the LSP, as it ensures the robustness and consistency of polymorphism.
    - In this case, the pros outweigh the cons as using abstractions allows for easier extension and maintenance of the codebase, without having to modify existing classes and potentially introducing bugs.

4. **Interface Segregation Principle (ISP):**
    - **Application:**
        - Purchasable interface has only two methods, processPurchase and getBaseCost, which are implemented by the classes that require them.
        - These methods are only implemented by the classes that require them (item that can be purchased), and not by the classes that do not require them.
    - **Why:**
        - Adhering to the ISP ensures that classes only implement the methods they need, eliminating the need for classes to implement methods they do not care about or need.
    - **Pros/Cons:**
        - One of the benefits of adhering to the ISP is ensuring that classes only implement the methods they need.
        - The downside could be the need to create more interfaces. However, spliting larger interfaces into smaller ones lead to a more maintainable and cleaner code.
5. **Dependency Inversion Principle (DIP)**
    - **Application:**
        - PurchaseAction class depends on Purchasable interface, which is an abstraction, rather than concrete classes. This allows the PurchaseAction class to not be dependent on the concrete classes that implement the Purchasable interface.
        - ConsumeAction depends on Consumable interface, which is an abstraction, rather than concrete classes. This allows the ConsumeAction class to not be dependent on the concrete classes, like EnergyDrink, that implement the Consumable interface.
        - ComputerTerminal class have to depend on the concrete classes that implement the Purchasable interface to provide the actions to purchase each particular Purchasable available in the ComputerTerminal.
    - **Why:**
        - Adhering to the DIP ensures that high-level modules depend on abstractions rather than the details of low-level concrete implementations. The code can be extended and maintained more easily by depending on abstractions, which can be implemented by different concrete classes with different implementation details.

- **Pros/Cons:**
  - The benefits of adhering to the DIP include reducing dependencies on concrete classes by depending on abstractions, making the code more extensible and easier to maintain.
  - There are no significant downsides to adhering to the DIP, as it follows better design practices and leads to a more extensible and maintainable code.

6. **Don't Repeat Yourself (DRY)**
   - **Application:**
     - All of these newly added classes inherit methods from their super classes, and only overrides the methods that are specific to them, without repeating the code that is already implemented in the super classes, adhering to the DRY principle.
   - **Why:**
     - Adhering to the DRY principle reduces redundancy in the codebase. Reusing existing code and avoiding duplication results in a cleaner, more maintainable codebase.
   - **Pros/Cons:**
     - The benefits of adhering to the DRY principle include reducing redundancy and improving maintainability.
     - There are no significant downsides to adhering to the DRY principle when used properly.

What Has Been Done and Why:

Added ToiletPaperRoll, EnergyDrink classes, both extending Item.
Added DragonSlayerSword class, extending WeaponItem.
Three of these classes implement Purchasable interface.

These three concrete classes that inherit either abstract Item or WeaponItem class have the necessary methods to get the actions to be picked up (`getPickUpAction`) and dropped (`getDropAction`) similar to other items so far.

Purchasable is an interface to add a capability to be purchased to the classes. In this case, it is used to add functionality to be purchased to three of these Item subclasses, with a method signature to process a purchase.

Process purchase would generally check Actor's balance, deduct the balance of the Actor's wallet if there is sufficient balance, otherwise throw a custom InsufficientBalanceException.

Added ComputerTerminal class that extends Ground.

Ground class has the functionality of a terrain, particularly providing a collection of Actions that the Actor can perform on its instance. This is where ComputerTerminal provide a surrounding Actor the actions to purchase a Purchasable available in the ComputerTerminal.

Added PurchaseAction extending Action.

PurchaseAction takes a Purchasable as parameter. This action tries to process the purchase of the Purchasable, and catch the potential InsufficientBalanceException when balance is insufficient, implementing the execute method from Action.

The application creates the ComputerTerminal and adds them to the gameMap

The functionality to add items to the location on gameMap has been provided by the game engine. Application class acts as the starting point of the game, initialising the game entities before the game loop starts running.

Overall Pros and Cons of the Design

Pros

- The implementation adheres to SOLID principles, which makes it easier to debug and extend the code.
- With the uses of abstraction and inheritance, it reduces redundancy and facilitates reusability and extensibility of the code.

Cons

- The processing of the purchase of the items are somewhat similar, and implemented in each of the classes that implement the Purchasable interface, ToiletPaperRoll, EnergyDrink and DragonSlayerSword. It could possibly be refactored to a single class that processes the purchase of the Purchasable in the future if more items are added that can be purchased to reduce redundancy.

How it can be easily extended

- To add more items that can be purchased, only need to implement the Purchasable interface by implementing the required methods in the concrete class.
- To add more purchasable items to the ComputerTerminal, only need to update the allowableActions() of ComputerTerminal by including new PurchaseAction with the corresponding Purchasable item. High level modules depend on the abstraction, like Purchasable, and is independent of the implementation details of the concrete classes that extend Item.

Conclusion:

Overall, our chosen design provides a robust framework for implementing the Requirement 4: Static factory's staff benefits in the game. By carefully considering the requirement and adherence to design principles like SOLID and DRY, we have developed a solution that is efficient, scalable, maintainable, paving the way for future extensions and maintenance.