# FIT2099 Assignment 2 Design Documentation

Applied Session 33 (Group 4)

Group members:

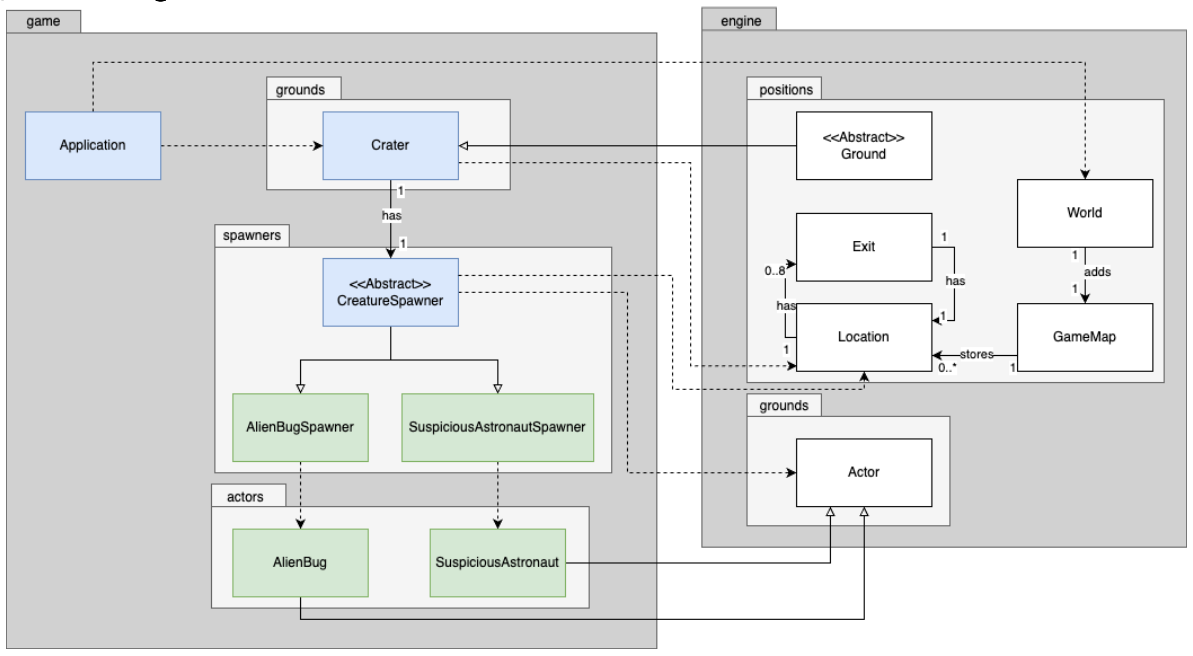
1. Claire Zhang
2. Jin Ruo Yew
3. Anh Nguyen
4. Ke Er Ang (Chloe)

---

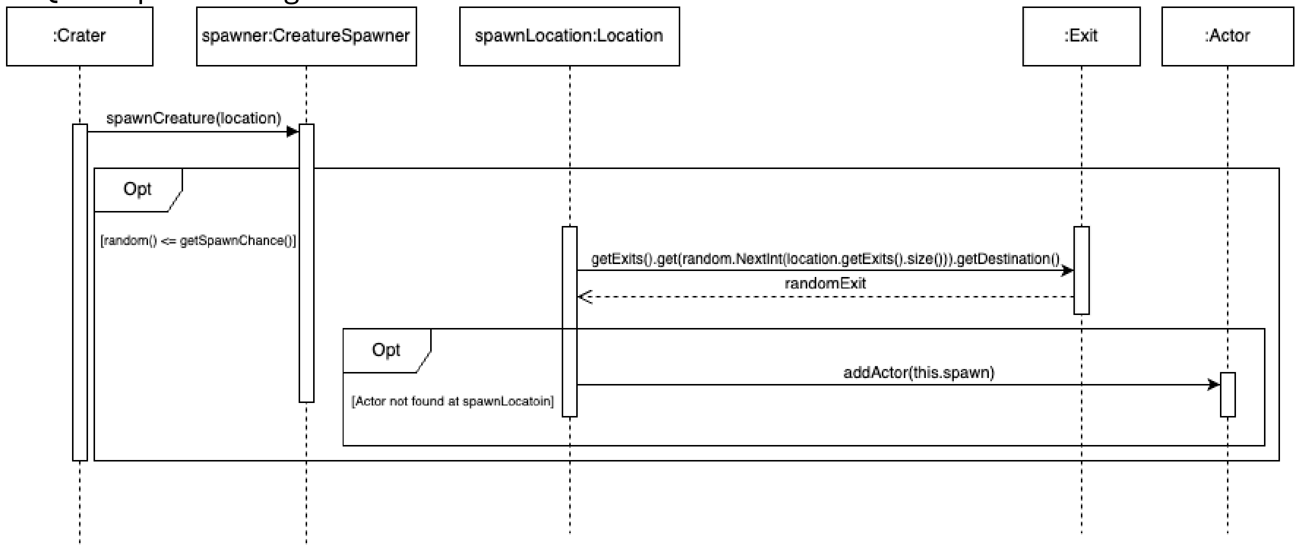
## Design Diagrams

*Note: For all UML diagrams, <u>newly created</u> classes are coloured in <u>green</u>, <u>updated/modified</u> classes from base code are coloured in <u>blue</u>.

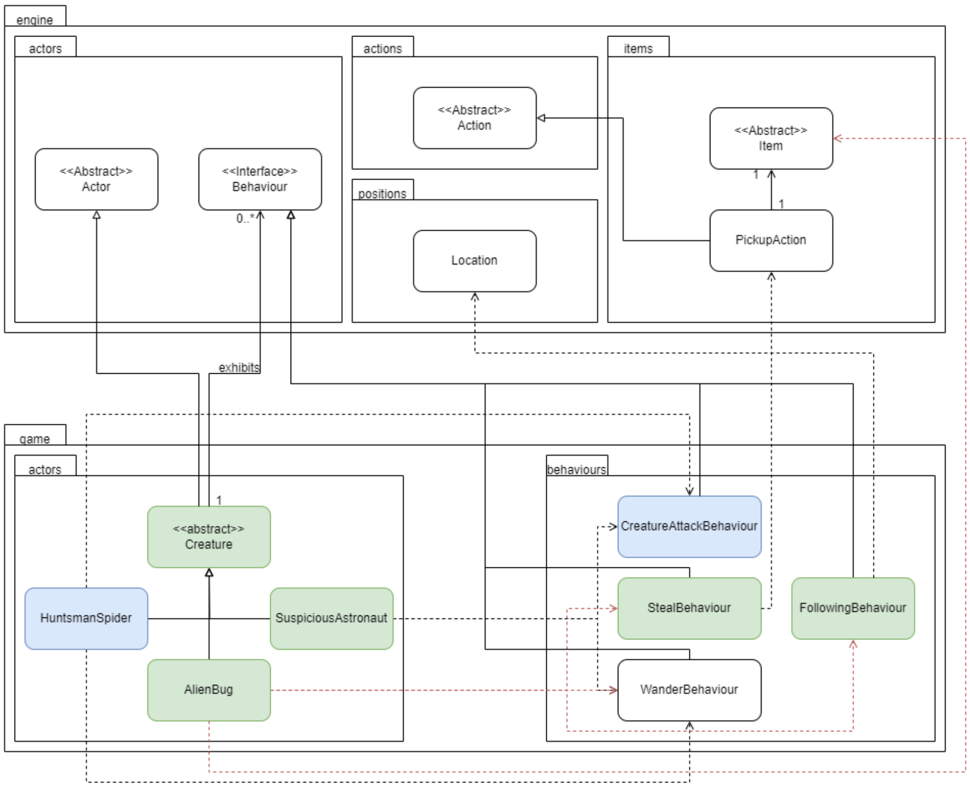
REQ 1 UML Diagram:



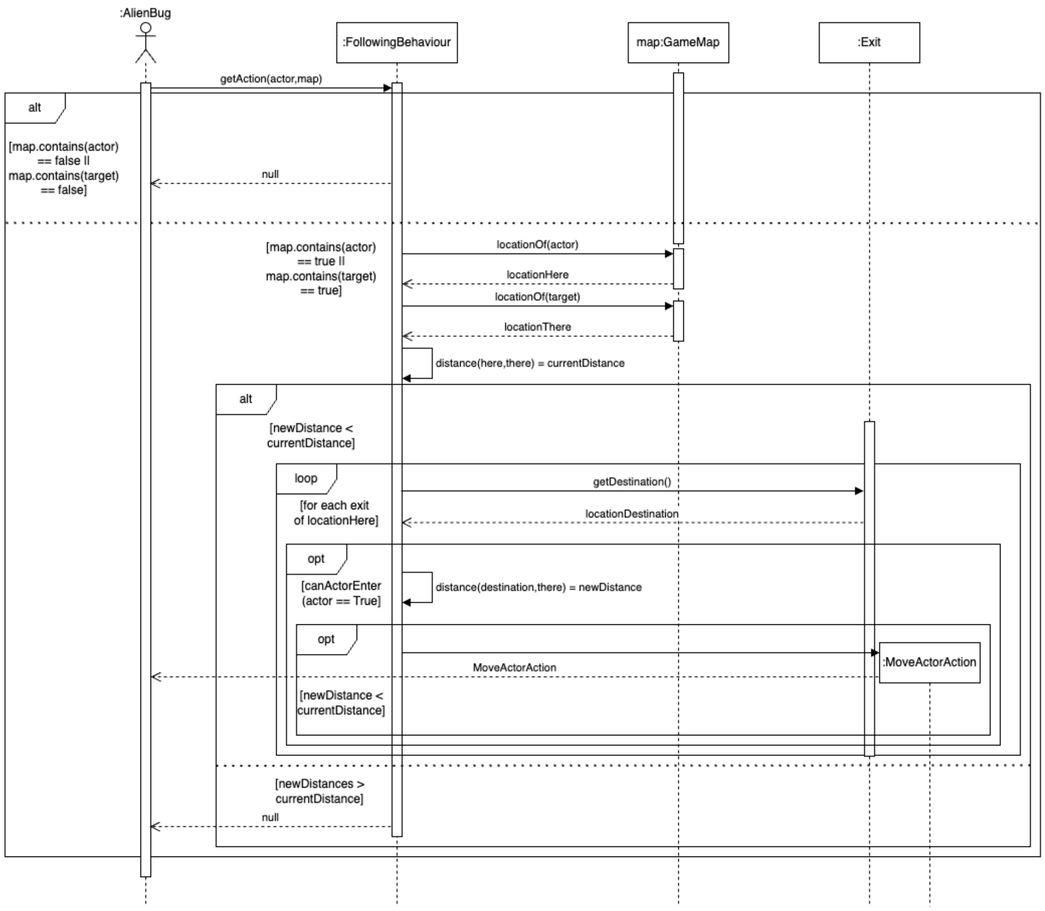
REQ 1 Sequence Diagram:


:Crater
spawner:CreatureSpawner
spawnLocation:Location
:Exit
:Actor
spawnCreature(location)
Opt
[random() <= getSpawnChance()]
getExits().get(random.NextInt(location.getExits().size())).getDestination()
randomExit
Opt
[Actor not found at spawnLocatoin]
addActor(this.spawn)

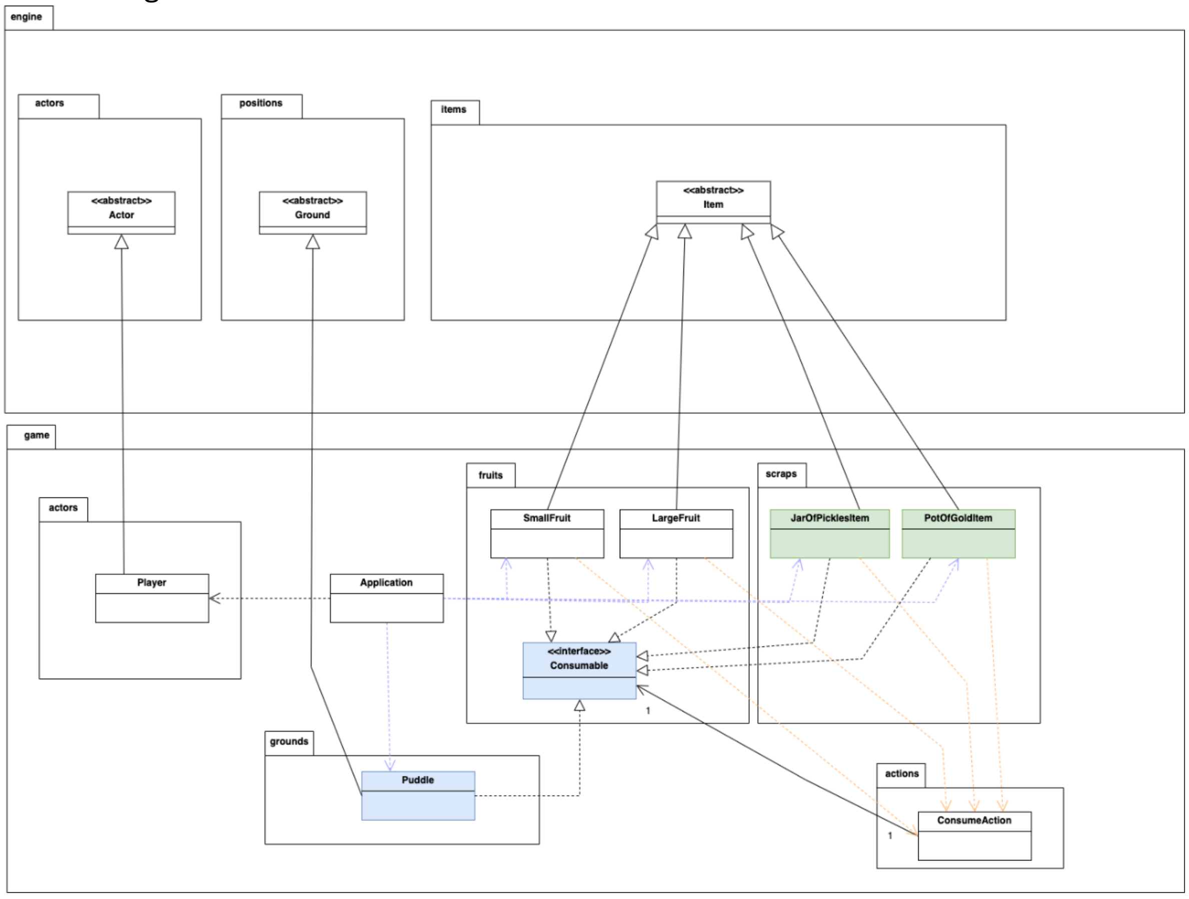
REQ 2 UML Diagram:



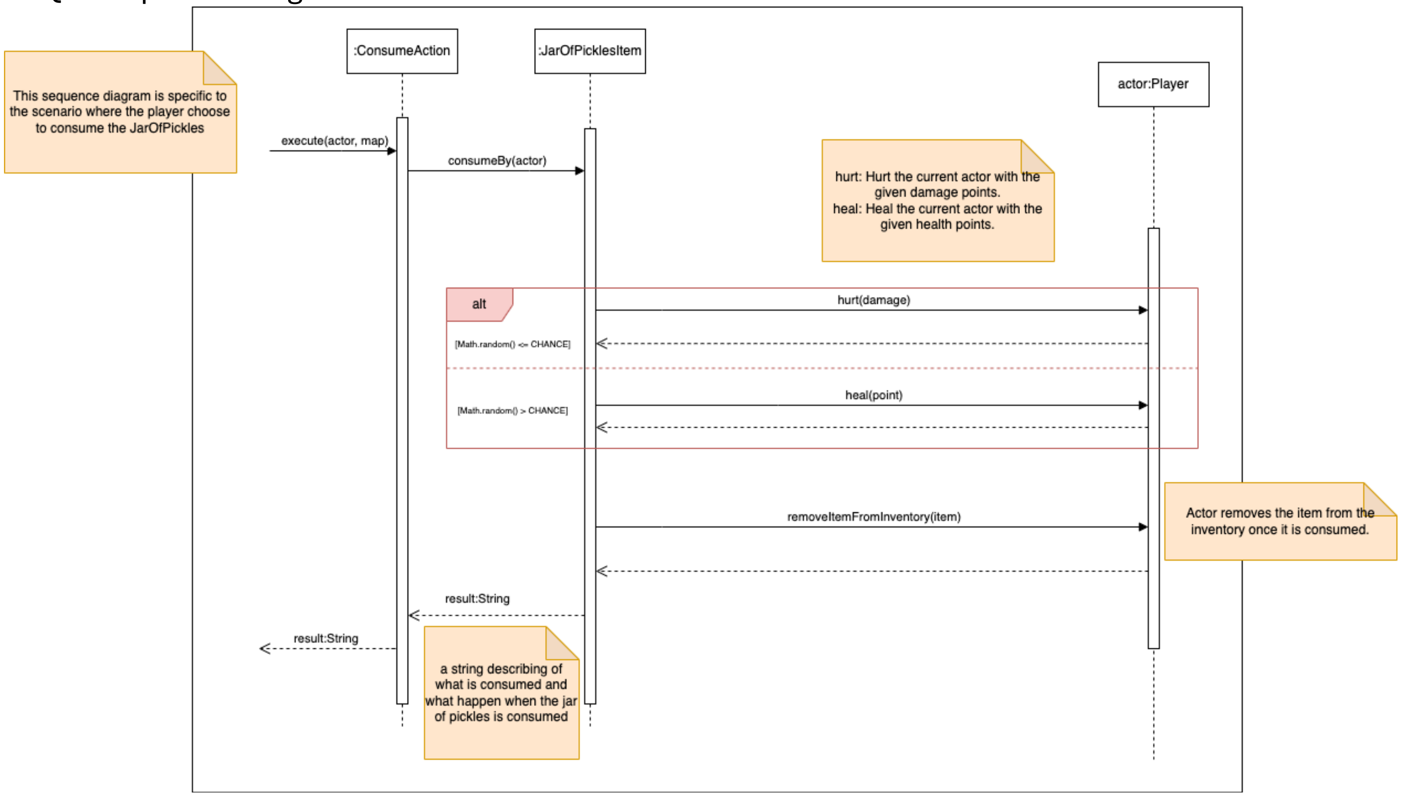
REQ 2 Sequence Diagram:


:AlienBug
:FollowingBehaviour
map:GameMap
:Exit
getAction(actor,map)
alt
[map.contains(actor) == false ll map.contains(target) == false]
null
[map.contains(actor) == true ll map.contains(target) == true]
locationOf(actor)
locationHere
locationOf(target)
locationThere
distance(here,there) = currentDistance
alt
[newDistance < currentDistance]
loop
[for each exit of locationHere]
getDestination()
locationDestination
opt
[canActorEnter (actor == True]
distance(destination,there) = newDistance
opt
[newDistance < currentDistance]
:MoveActorAction
MoveActorAction
[newDistances > currentDistance]
null

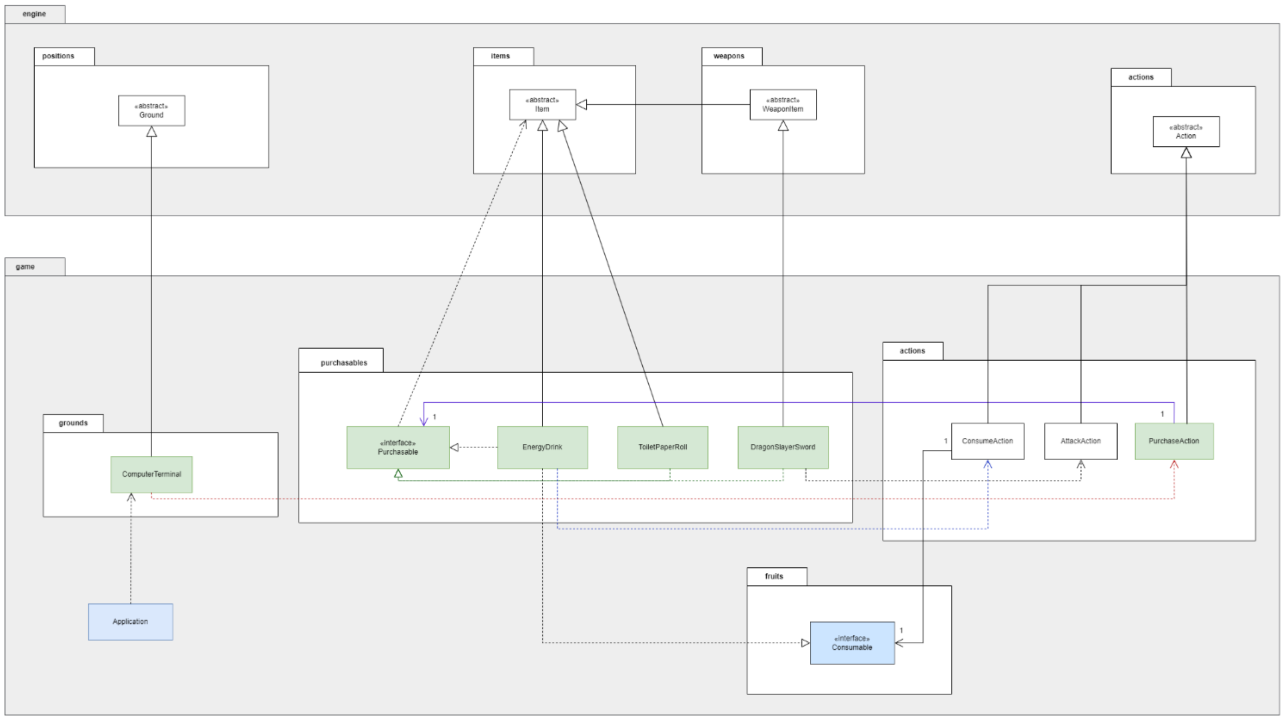
## REQ 3 UML Diagram:



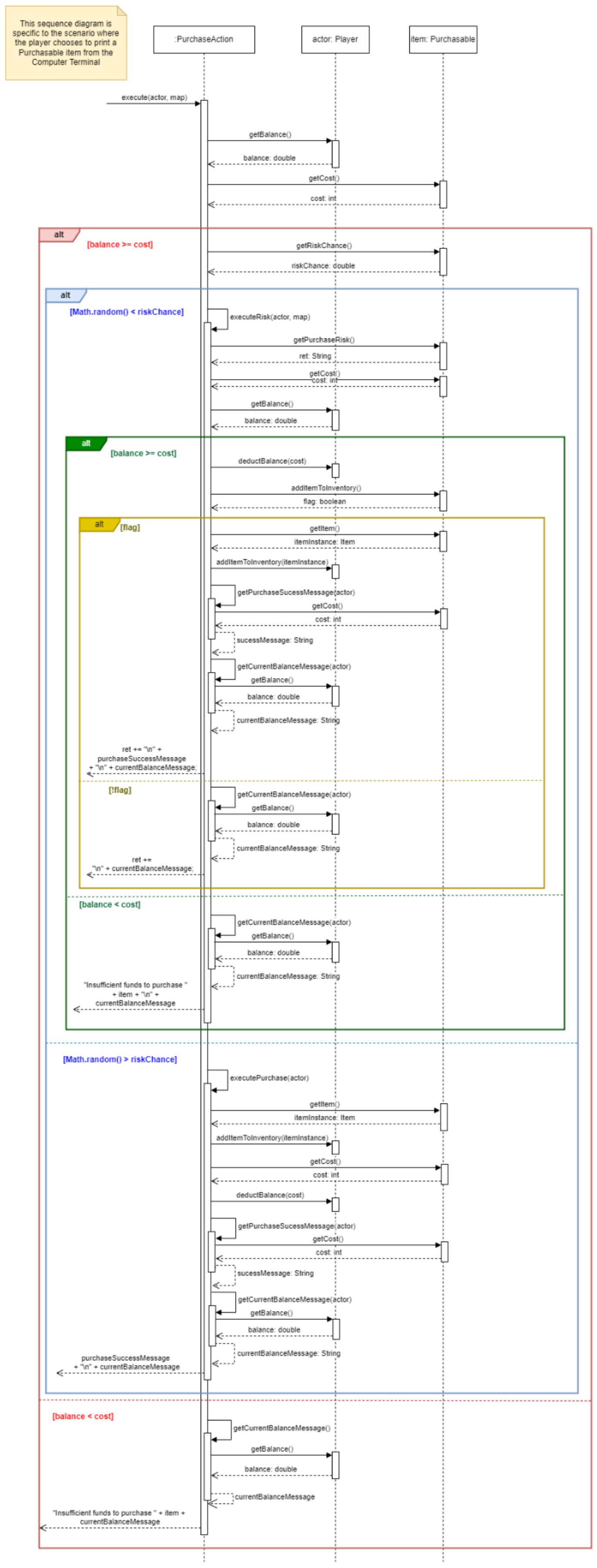
## REQ 3 Sequence Diagram:


This sequence diagram is specific to the scenario where the player choose to consume the JarOfPickles
:ConsumeAction
:JarOfPicklesItem
actor:Player
execute(actor, map)
consumeBy(actor)
hurt: Hurt the current actor with the given damage points.
heal: Heal the current actor with the given health points.
alt
hurt(damage)
[Math.random() <= CHANCE]
heal(point)
[Math.random() > CHANCE]
removeItemFromInventory(item)
Actor removes the item from the inventory once it is consumed.
result:String
result:String
a string describing of what is consumed and what happen when the jar of pickles is consumed

REQ 4 UML Diagram:

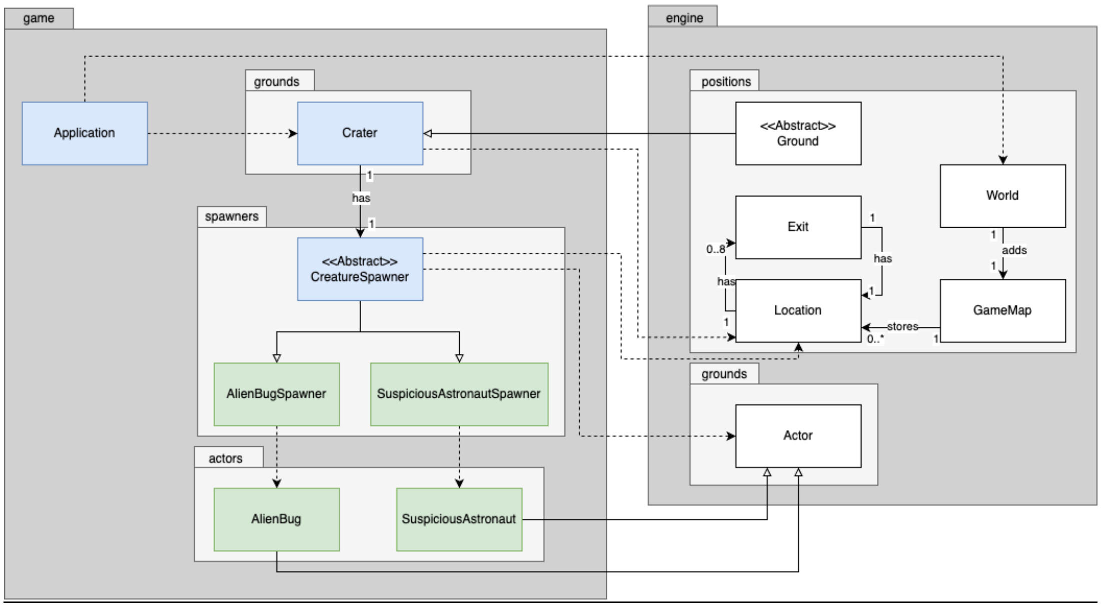


## REQ 4 Sequence Diagram:


This sequence diagram is specific to the scenario where the player chooses to print a Purchasable item from the Computer Terminal
:PurchaseAction
actor: Player
item: Purchasable
execute(actor, map)
getBalance()
balance: double
getCost()
cost: int
alt
[balance >= cost]
getRiskChance()
riskChance: double
alt
[Math.random() < riskChance]
executeRisk(actor, map)
getPurchaseRisk()
ret: String
getCost()
cost: int
getBalance()
balance: double
alt
[balance >= cost]
deductBalance(cost)
addItemToInventory()
flag: boolean
alt
[flag]
getItem()
itemInstance: Item
addItemToInventory(itemInstance)
getPurchaseSucessMessage(actor)
getCost()
cost: int
sucessMessage: String
getCurrentBalanceMessage(actor)
getBalance()
balance: double
currentBalanceMessage: String
ret += "\n" + purchaseSuccessMessage + "\n" + currentBalanceMessage;
[!flag]
getCurrentBalanceMessage(actor)
getBalance()
balance: double
currentBalanceMessage: String
ret += "\n" + currentBalanceMessage;
[balance < cost]
getCurrentBalanceMessage(actor)
getBalance()
balance: double
currentBalanceMessage: String
"Insufficient funds to purchase " + item + "\n" + currentBalanceMessage
[Math.random() > riskChance]
executePurchase(actor)
getItem()
itemInstance: Item
addItemToInventory(itemInstance)
getCost()
cost: int
deductBalance(cost)
getPurchaseSucessMessage(actor)
getCost()
cost: int
sucessMessage: String
getCurrentBalanceMessage(actor)
getBalance()
balance: double
currentBalanceMessage: String
purchaseSuccessMessage + "\n" + currentBalanceMessage
[balance < cost]
getCurrentBalanceMessage()
getBalance()
balance: double
currentBalanceMessage
"Insufficient funds to purchase " + item + currentBalanceMessage

**Design Rationale**

**REQ 1**



This diagram represents an aspect of the rogue-like game where craters across the game map can not only spawn huntsman spiders, but also other creatures, including alien bugs and suspicious astronauts.

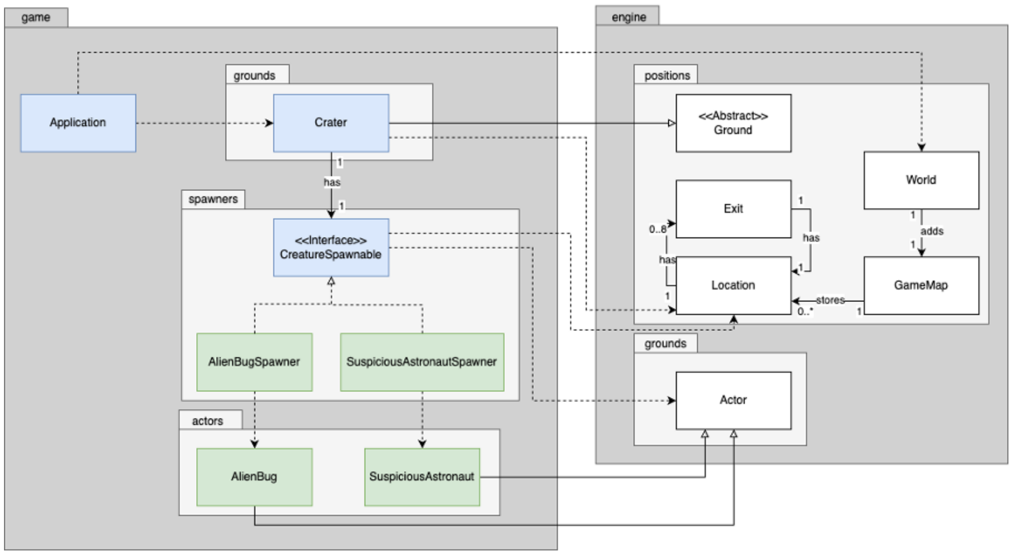
**Proposed design**

The proposed design has ground Crater injected with an abstract CreatureSpawner, which carries out the duty of spawning a new instance of the creature and placing it on the map. AlienBugSpawner, SuspiciousAstronautSpawner and any other future creature spawners can extend from the CreatureSpawner and share the same spawnCreature(location) code.

| Pros | Cons |
|---|---|
| • The dependency injection of CreatureSpawner into **Crater** keeps Crater class focused on representing a ground object (SRP).<br><br>• Connascence of Identity is avoided, as the CreatureSpawner instantiates a new creature to spawn from the crater, rather than pointing to one creature stored as an attribute.<br><br>• Avoids repeated code that spawns creatures, by extending from CreatureSpawner (DRY). | • **CreatureSpawner** has multiple responsibilities, it both instantiates the creature, and places the creature on the map (Violates SRP). |

| | |
|---|---|
| • Future creature spawners can be instantiated anywhere CreatureSpawner is expected, without modifications to the code. (LSP/Polymorphism). | |

**Alternative design**

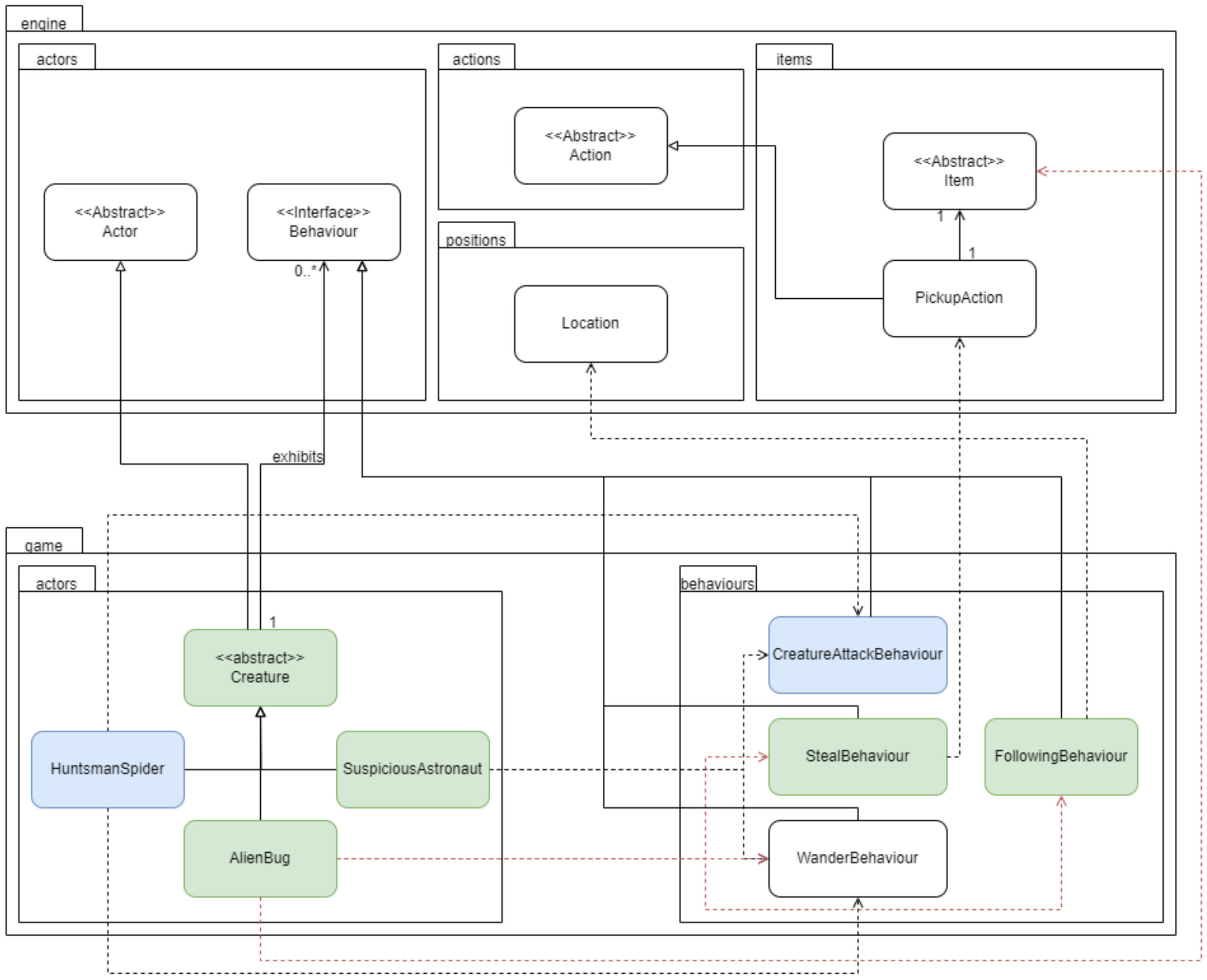


In this alternative design, instead of the abstract CreatureSpawner, Crater has an interface CreatureSpawnable that has a method that should create an instance of a creature, and another method that decides whether to spawn a creature. Different concrete creature spawners can implement this interface.

| Pros | Cons |
|---|---|
| • The dependency injection of CreatureSpawnable into Crater keeps Crater class focused on representing a ground object (SRP).<br><br>• Connascence of Identity is avoided, as the CreatureSpawnable implementers instantiate a new creature to spawn from the crater, rather than pointing to one creature stored as an attribute.<br><br>• CreatureSpawnable interface provides flexibility in implementation.<br><br>• Creature spawners can implement multiple interfaces aside from | • Apart from the different spawn rates and creature instances, the spawn implementation is identical for SuspiciousAstronautSpawner and AlienBugSpawner, so code is repeated (Violates DRY).<br><br>• If one implementation of CreatureSpawnable spawns a random creature, while another only spawns one creature, substituting the latter for the former could lead to unexpected behaviour (violates LSP) |

| | |
|---|---|
| CreatureSpawnable for additional functionality (OCP). | |

**REQ 2**



**Design Goal:**
The design goal for this assignment is to incorporate new Actors into the existing game while adhering closely to relevant design principles and best practices.

**Design Decision:**
In implementing SuspiciousAstronaut and AlienBug, the decision was made to create an abstract Creature class with playTurn and allowableActions methods reused from HuntsmanSpider. This approach was chosen to promote code reusability and maintainability, ensuring that all creature types benefit from unified methods.

**Alternative Design:**
One alternative approach could involve each new class (SuspiciousAstronaut and AlienBug) directly extending the Actor class and implementing their own versions of playTurn and allowableActions methods independently. However, this design is less preferred because it

leads to significant code duplication, which can increase maintenance challenges and the risk of bugs.

**Analysis of Alternative Design:**
The alternative design is not ideal because it violates various Design and SOLID principles:

1. **DRY:**
   - Both SuspiciousAstronaut and AlienBug would duplicate code from Huntsmanspider (playTurn and allowableActions method) violating DRY
2. **High Connascence of Convention:**
   - Having near identical methods in multiple classes increases connascence of convention, meaning that changes in those methods' logic would require changes across all classes with those methods

**Final Design:**
In our design, we closely adhere to:

1. **SRP:**
   - **Application:** The new abstract Creature class centralises common code, which simplifies the modification of behaviour across all creatures without altering individual subclass
   - **Why:** Adhering to the SRP is crucial as it ensures that each class has only one reason to change, reducing the complexity of updates and the potential for errors during maintenance. It facilitates easier debugging and testing since changes are localised.
   - **Pros:** The abstract Creature class enhances maintainability by centralising behaviour modification in one location, such as handling playTurn and allowableActions methods for all derived classes. This reduces the need for changes in multiple classes when updating general behaviours.
   - **Cons:** However, there is a risk that the Creature class may become a bottleneck if too many responsibilities are loaded onto it, potentially making the class cumbersome and harder to manage if the behaviours of derived creatures diverge significantly.

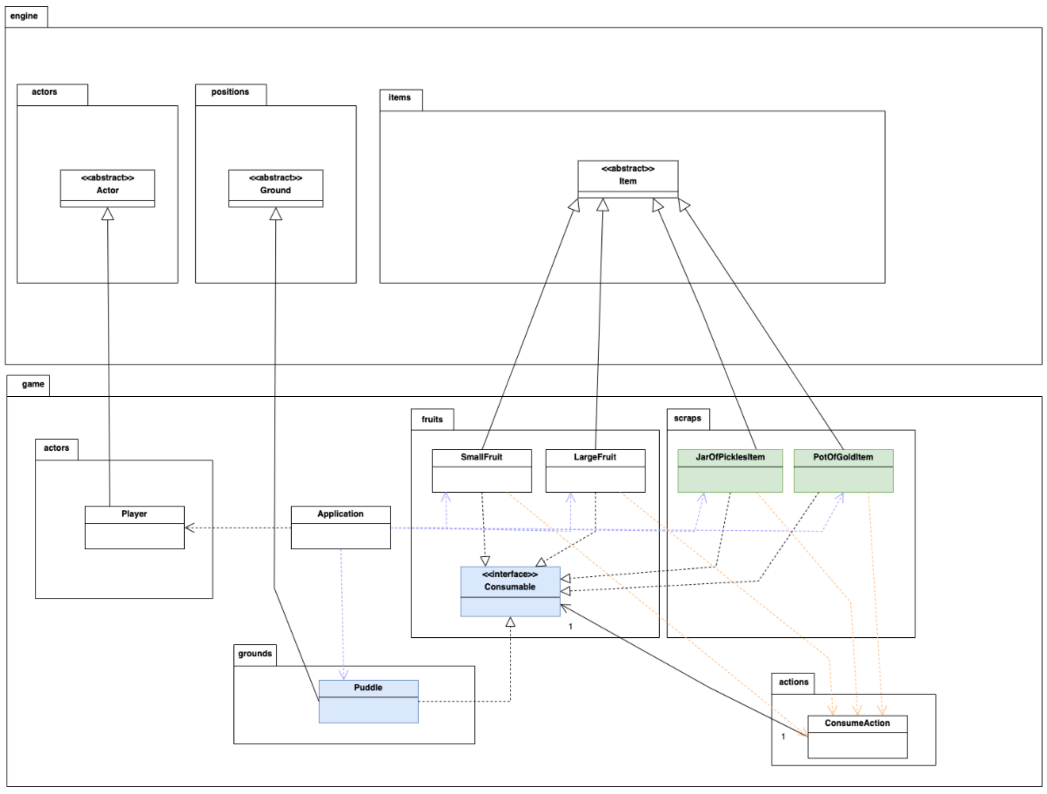
2. **OCP:**
   - **Application:** The design is open for extension but closed for modification. New types of creatures can be added without modifying existing code, just the extending Creature class. Overriding methods in subclasses (like in AlienBug) allows for extending functionality without modifying the abstract class
   - **Why:** Adherence to OCP is vital as it allows the software module to be extensible without being modified, which reduces the risk of introducing bugs into the existing code during enhancements.
   - **Pros:** New creature types like AlienBug can be introduced by simply extending the Creature class and overriding specific methods without needing to alter the existing code structure, promoting easier scalability and extensibility.

- **Cons:** Less flexibility, all types of Creatures would have to adhere to the abstract class.

3. **LSP:**
   - **Application:** Subclasses (HuntsmanSpider, SuspiciousAstronaut, and AlienBug) can be used interchangeably without affecting the functioning of the system.
   - **Why:** Following LSP is important as it ensures that a subclass can stand in for its parent class without disrupting the behaviour of the program. This predictability is essential for system reliability and interface consistency.
   - **Pros:** Ensures that instances of SuspiciousAstronaut, HuntsmanSpider, and other subclasses can be replaced with one another without affecting the integrity of the system, making the software more robust and reliable.
   - **Cons:** However, Strict adherence to this principle might limit how different behaviours can be implemented in subclasses. For instance, if AlienBug needs to significantly alter a method that is central to the Creature class's contract, it could violate LSP by not fully adhering to the expected behaviour.

**Conclusion:**
Overall, our chosen design provides a robust framework for integrating new creature types into the existing system. By carefully considering relevant factors, such as design principles, requirements, and constraints, we have developed a solution that is efficient, scalable, and maintainable, paving the way for future enhancements, extensions, and optimizations.

## REQ 3



In this design, the Item class serves as a superclass which branches into specific items such as small fruit, large fruit, jar of pickles and pot of gold. Besides, the Consumable class which was initially an abstract class in assignment 1, has been changed to an interface which is implemented by consumable items. This approach of implementation makes sure that only objects meant to be consumable must have the methods in the Consumable interface. Lastly, the ConsumeAction class is associated with consumable objects and by leveraging the methods specified by the Consumable interface, it defines the mechanics of how items are consumed within the game.

| Pros | Cons |
|---|---|
| Having a Consumable class as interface is more extensible, as it is not limited to be extended by subclasses of a specific type. Other new objects of different types with different consuming behaviour can be easily added without changing the code in consumable class. (Open Closed Principle) Hence, it can be easily maintained and extended. | Abstract classes and interfaces, even though they offer a clear and structured framework, can come with additional overhead related to performance and code base comprehension. Sometimes the development process might be slowed down by using too many abstractions due to the complexity of the code. |
| As the ConsumeAction class is associated with a consumable object, it does not depend on the concrete implementations of the item. The implementation details instead depend on the abstraction class which eases the maintenance and improves the code's modularity. | ConsumeAction associated on the Consumable interface by name. If the interface name changes, ConsumeAction would also need to change. (Connascence of name) |
| As the class is loosely coupled, ConsumeAction can operate with any Consumable without knowing its concrete implementation. | |
| As high-level modules do not depend on low-level modules and both depend on abstractions, this design adheres to the dependency inversion principle. | |

**Alternative design:**

The alternative design is to have the BalanceIncreasable Interface which is implemented by PotOfGold item and CreditToWallet class which extends the action class and have an association relationship with the BalanceIncreasable class. The CreditToWallet action will implement the execution logic when the PotOfGold is consumed and the actual logic of how the balance is increased when a PotOfGold is consumed will be defined in PotOfGold class.

| Pros | Cons |
|---|---|
| The BalanceIncreasable item only defines the methods  that can improve a player's balance and the CreditToWalletAction only focuses on implementing the action of crediting the wallet when an item is consumed. These extra classes as well as the Consumable and ConsumeAction classes now only have one job to handle which is either increase/decrease hitpoints of the actor or increase/decrease balance of the player, it adheres to Single Responsibility Principle. | BalanceIncreasable interface might be underutilized as not much item is allowed to increase balance of the player, making this additional abstraction layer somewhat redundant. |
| Other items that can increase the player’s balance can just implement the BalanceIncreasable interface without modifying other classes, hence the class is easily extended and maintained. | As the BalanceInterface class and Consumable classes as well as the ConsumeAction and CreditToBalance classes have a lot of similar properties, there might be repetition of code and hence violates DRY principle. |

**REQ 4**

A. Implementation of Computer Terminal (=) that can print items
   Computer Terminal is implemented as a concrete class that extends Ground from the engine, since the Player can stand on top of it. The allowableActions() method is overridden in this class to add actions to purchase items.

| Pros | Cons |
|---|---|
| Code reusability, ComputerTerminal class inherits methods from its parent, Ground, allowing us to reuse code which was already written in the parent class, reducing repetitive code blocks. | Increased level of coupling between classes, any changes to the purpose of Ground should be changed in its child classes as well (connascence of meaning), which may lead to unexpected behaviours. |
| Easy to extend and maintain, new purchasable items can be added by addition of an action that stores the item. | Limits the flexibility of how the ComputerTerminal class can be implemented or used in the game. |
| Encapsulates all purchase actions within the ComputerTerminal class improves code organization, with only the terminal handling all purchasing actions. | |

B. Implementation of Items that can be printed from the Computer Terminal
Create the 3 new items listed below:
(1) Energy Drink extends abstract Item class from engine and implements Consumable.
(2) Dragon Slayer Sword extends abstract WeaponItem class from engine.
(3) Toilet Paper Roll extends abstract Item class from engine.
Create a new Purchasable interface, where all the item classes listed implements the created interface.

| Pros | Cons |
|---|---|
| Code reusability, extending different abstract classes given in the engine for each item based on their traits, we can reuse the base code to the maximum extent. | Tight coupling between classes and the interface, methods within the interface must be well considered to be relevant to all purchasable items. |
| Increased flexibility and extensibility, using an interface to group all purchasable items allows flexible implementation of adding the traits and behaviours of purchasable items to newly created items. | Purchasing logic assumes certain properties of purchasable items, changes to the interface may require modifications to the interface, implementing classes and the concrete purchasing action class itself. |
| Increased modularity, purchasable items are grouped by their common behaviour of being purchasable, which is easier to maintain. | |

C. Implementation of allowing Player to choose to pay with credits to print items from the Computer Terminal
Design 1: Create an interface PurchaseItemAction, which provides basic methods to be used when executing a purchase action, then implementing concrete classes which extend Action from the given engine and implement PurchaseItemAction interface for each item that can be printed from the Computer Terminal.

| Pros | Cons |
|---|---|
| Separating the purchase action logic into its own interface and concrete classes promotes separation of concerns. | Repetitive code blocks for methods that have the same logic across every concrete action class for each printable item, decreases code reusability. |
| Flexibility in how purchase actions are implemented for different items. Each concrete class implementing PurchaseItemAction can design logic | Increased complexity, adding an additional interface and multiple concrete classes for purchase actions adds complexity to the system when |

| | |
|---|---|
| specific to its associated item, easier to implement different purchase logic for different items. | there exist many purchasable items, making the code harder to maintain. |

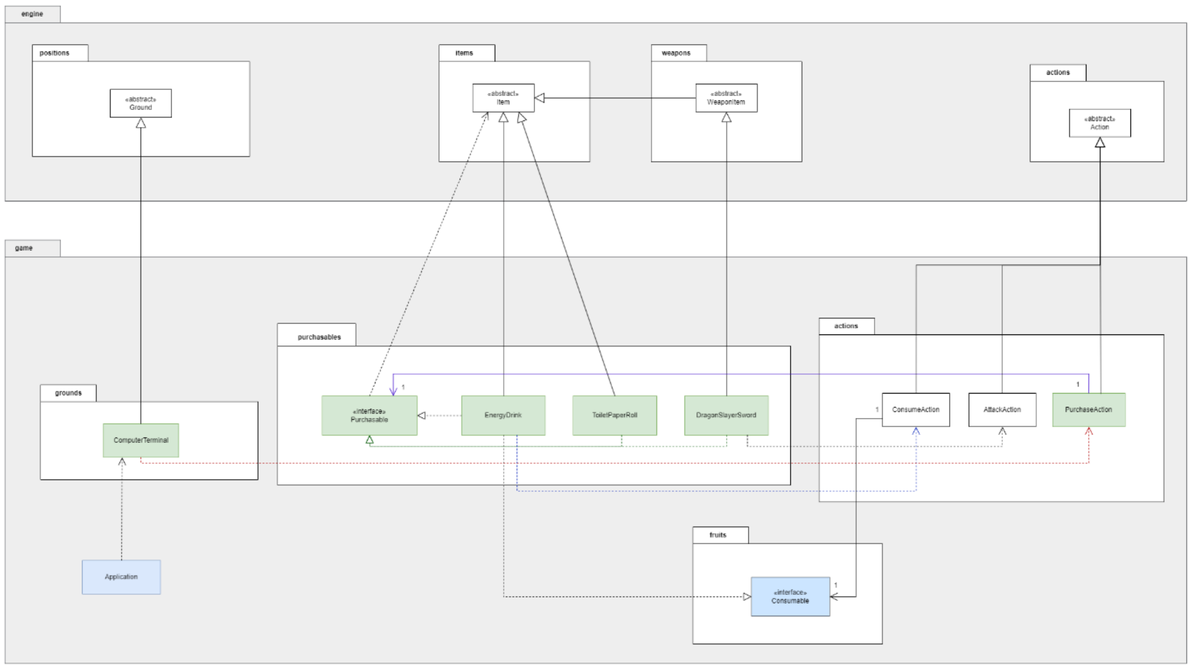
Design 2: Create an abstract class PurchaseItemAction which extends Action from the given engine and includes abstract methods for child classes to implement to execute the purchase action. Concrete action classes for each printable item then extends this created abstract class.

| Pros | Cons |
|---|---|
| Code reusability, base implementation of purchasing can be inherited from the created abstract class. | Multi-level inheritance, deeply nested inheritance may cause differences in the behaviour of child classes and their root parent. |
| Increased flexibility, child classes which inherit the abstract PurchaseItemAction class can implement abstract methods defined in the parent to implement different purchase logic for different items. | Dependencies on specific method names or variable names between the PurchaseItemAction abstract class and its concrete subclasses (Connascence of name). |

Design 3: Creating class PurchaseAction which extends Action that is used for any Purchasable item (stores the Purchasable).

| Pros | Cons |
|---|---|
| Higher code reusability, using a generic PurchaseAction class for every purchasable item avoids creating separate actions for each type. | Less flexibility, purchasing logic must be consistent across all purchasable items, harder to extend different variations of purchasing logic for different items from existing codebase. |
| Reduces dependencies with only 1 dependency from ComputerTerminal class to PurchaseAction compared to Design 1 & 2, making code easier to maintain without affecting too many classes. | PurchaseAction relies on the meaning of methods in purchasable items, changing the attribute names or method syntax would need modification in PurchaseAction class as well (Connascence of name & meaning). |
| PurchaseAction class encapsulates the logic for executing a purchase action, maintaining organized code. | Constant values defined in purchasable items are reused for checking and operations in PurchaseAction, changing predefined values may affect the |

| | |
|---|---|
| | execution of the action (Connascence of value). |



The UML diagram above shows the overall design of REQ 4, based on design decisions in parts A, B and Design 3 of part C. ComputerTerminal class which extends Ground from engine is responsible for adding new PurchaseActions that contain different types of items that implement the Purchasable interface. EnergyDrink class and ToiletPaperRoll class both extend Item, but EnergyDrink class implements an additional Consumable interface to heal the actor when consumed. DragonSlayerSword class extends abstract WeaponItem class to be used to attack targets, where WeaponItem also extends from abstract Item class.

The new abstract ComputerTerminal class for plants extends the abstract Ground class, new EnergyDrink class and ToiletPaperRoll class both extend Item, new DragonSlayerSword class extends abstract WeaponItem class. We want the Computer Terminal to also act as a type of Ground for actors to step on and purchasable items to implement the same logic (i.e. can be picked up/dropped off) as non-purchasable items in the game, it is logical to abstract the identities to avoid repetitions in code (DRY). Creating a new PurchaseAction class segregates the responsibility of handling purchase and process payment logic to a new class, avoiding existing classes having too many responsibilities (Single Responsibility Principle).

Using the new Purchasable interface allows us to remove dependencies of the PurchaseAction to different concrete purchasable items, by diverting the dependency on the Purchasable interface that abstracts the fundamental logic of all types of purchasable items (Dependency Inversion Principle). The new Purchasable interface also allows us to add new purchasable items by implementing the interface, with dependencies of classes that interact with purchasable items on the interface, we do not need to modify existing concrete purchasable item classes to extend the codebase (Open-Closed Principle).