Design Rationale

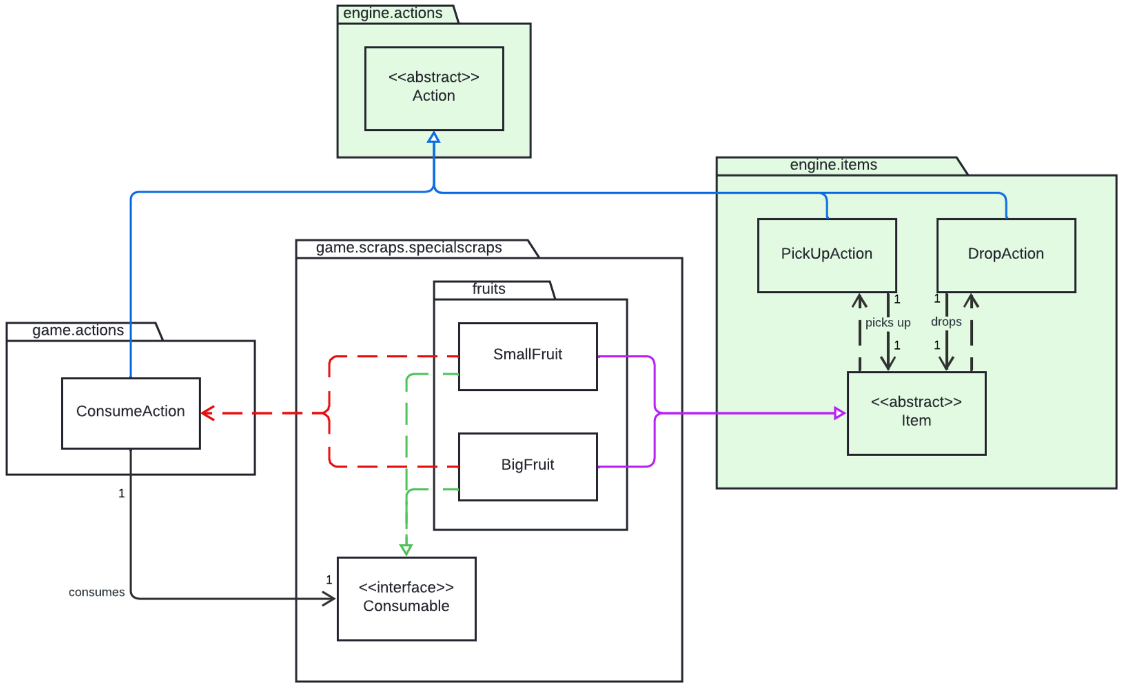
Design Goal

The design aims to break down the system into smaller, more manageable classes to promote modularity and encapsulation. In this system, each class will have a clear and distinct responsibility so it is easier to understand and maintain. By adhering to the SOLID principle and DRY principle, the system should allow for easy extension and promote reusability. This involves identifying common functionalities and abstracting them into reusable components as well as anticipating potential changes in order to enhance code flexibility and adaptability to changing requirements and future extensions.

Assignment 1 Modification

Classes Modified / Created	Roles and Responsibilities	Rationale
Inheritree (extends Ground)	An abstract class representing inheritree. Relationships: 1. It extends the Ground class to achieve the fruit-producing featu by overriding the tick() method.	**Alternate Previous Solution:** Introduce an Inheritree abstract class extending Ground class and create concrete classes for all stages of inheritree that inherit from it. On top of this, implement a method growToNextStage() which returns null to grow the tree after specified ticks. Sapling class overrides this method to return a Mature tree. **Finalised Solution:** Remove the implementation of method growToNextStage() from the classes. Then, in the Sapling class, overrides the tick() method to grow to next stage (ie. Mature tree). **Reasons for Decision:** 1. The previous design violates the Interface Segregation Principle which emphasises that a class should not be forced to implement methods it doesn't need. This can be observed in the Mature tree class where it was needed to implement the growToNextStage() method even though it cannot grow.
Sapling (extends Inheritree)	Class representing the sapling state of the inheritree. Relationships: 1. It extends the Inheritree class to promise to produce a fruit at each tick. 2. It depends on SmallFruit to produce SmallFruit. 3. It grows to the next state (ie. the mature state) by overriding	

	the tick() method.	2. In the latest design, the Sapling class call the tick() method in its super class to avoid duplicated code (ie. the implementation of feature to drop fruits) while overriding the tick() method.
MatureTree (extends Inheritree)	Class representing the mature state of the inheritree. Relationships: 1. It extends the Inheritree class to promise to produce a fruit at each tick. 2. It depends on BigFruit to produce BigFruit.	



New Class Diagram after removal of ConsumeItem and Fruits

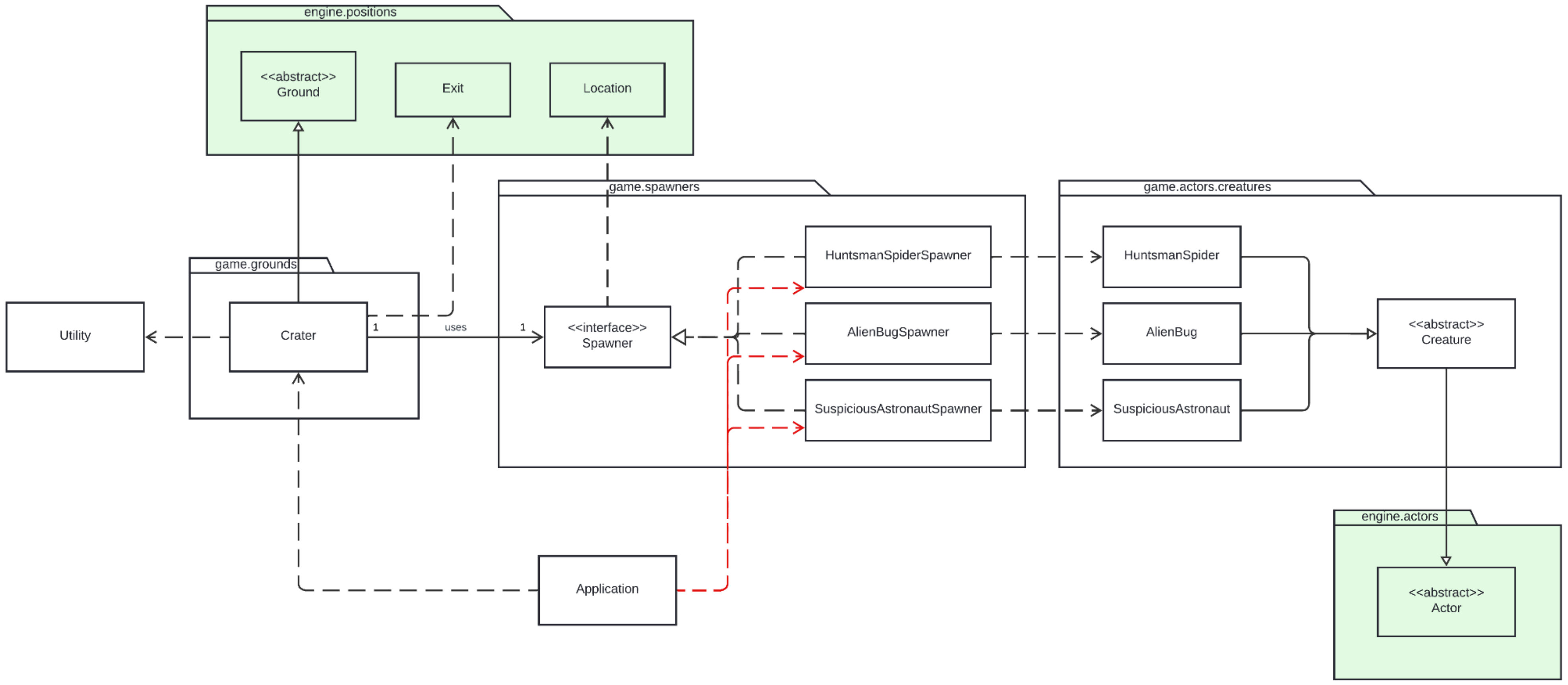
The reasoning behind this decision are as follows:

Classes Modified / Created	Roles and Responsibilities	Rationale
SmallFruit (extends Item, implements Consumable)	Class representing the small fruit produced by the inheritree at sapling state. Relationships: 1. It implements the Consumable interface and extends the Item class	**Alternate Previous Solution:** Introduce a Fruit abstract class that inherits ConsumableItem abstract class and create concrete classes for all types of fruits which inherit from it. ConsumableItem class that inherits the Item class is created for later requirements. For the abstract class Fruit, the SmallFruit class and BigFruit class that inherit it promise to have the characteristic dropping with a specific probability (ie. drop() method). **Finalised Solution:** Refactor BigFruit and SmallFruit to extend from Item and implement the Consumable interface. This allows the fruit to be picked up and dropped and still be consumed with implementing the consumedby method in the Consumable interface **Reasons for Decision:** 1. Although the original follows the SOLID principles, there is an issue of excessively deep abstract class hierarchy when Big Fruit and SmallFruit both inherit Fruit, which inherits ConsumeItem, which in turn, inherits Item. This can potentially cause complexity and potential maintenance issues as it would be harder to debug and maintain the code. Moreover, it is possible that it obscures the intended purpose of the affected classes, making it harder to understand the code. 2. Additionally, this reduces the redundancy which is the ConsumeItem class, which was ultimately not required as ConsumeAction takes in a class that implements Consumable interface, which means that the creation of ConsumeItem did not manage to reduce redundancy or dependency for the SmallFruit and LargeFruit classes and the Fruit abstract class. 3. As for the Fruit, the abstract class did not manage to reduce redundant code in any way as it only ultimately managed to reduce one line code in
BigFruit (extends Item, implements Consumable)	Class representing the small fruit produced by the inheritree at mature stage. 1. It implements the Consumable interface and extends the Item class	

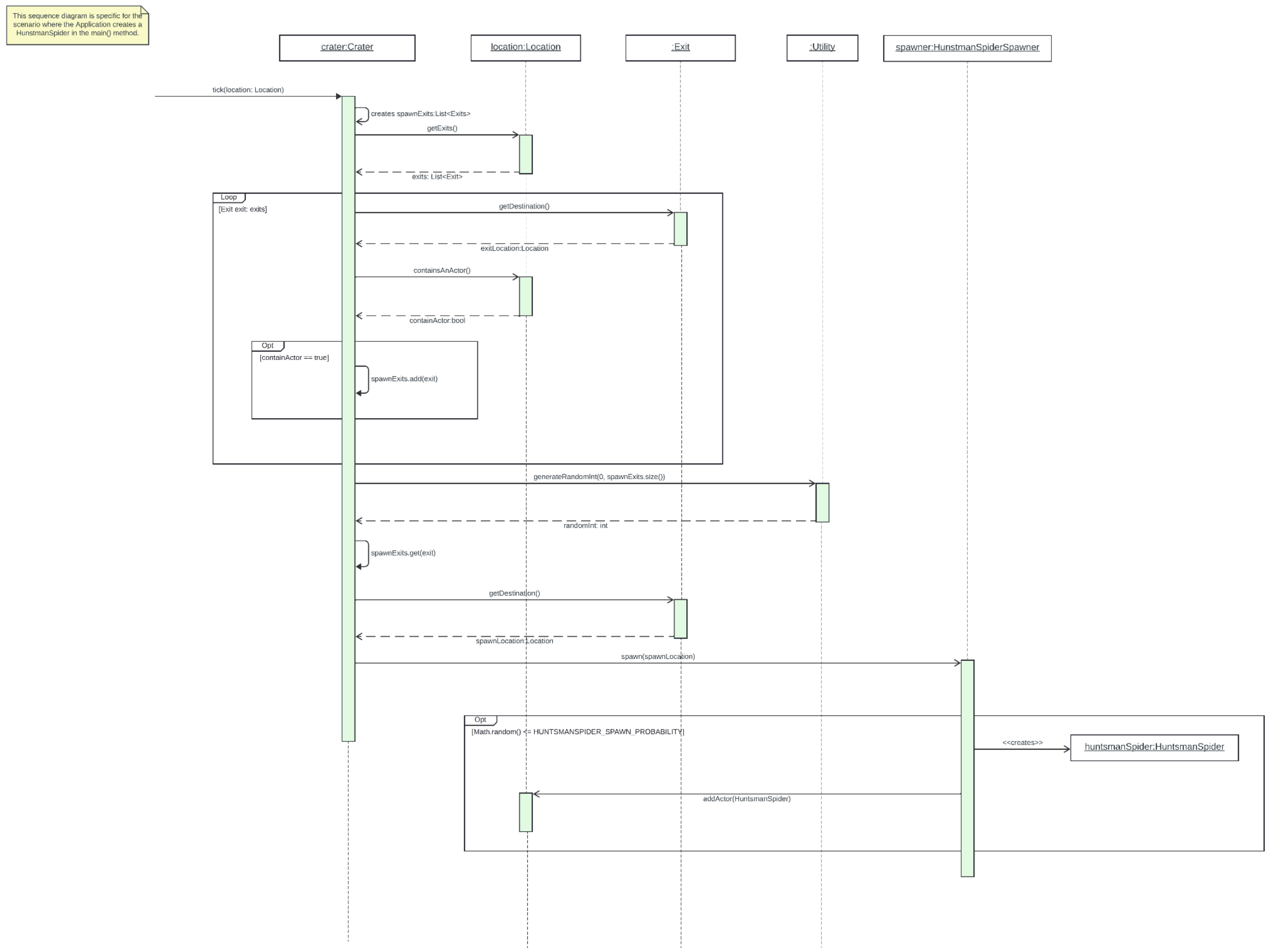
		the SmallFruit and LargeFruit classes and as the requirements in Assignment 2 was not looking to expand on Fruit Items, there was the decision to delete said class entirely.

Assignment 2

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



Requirement 1 Class Diagram

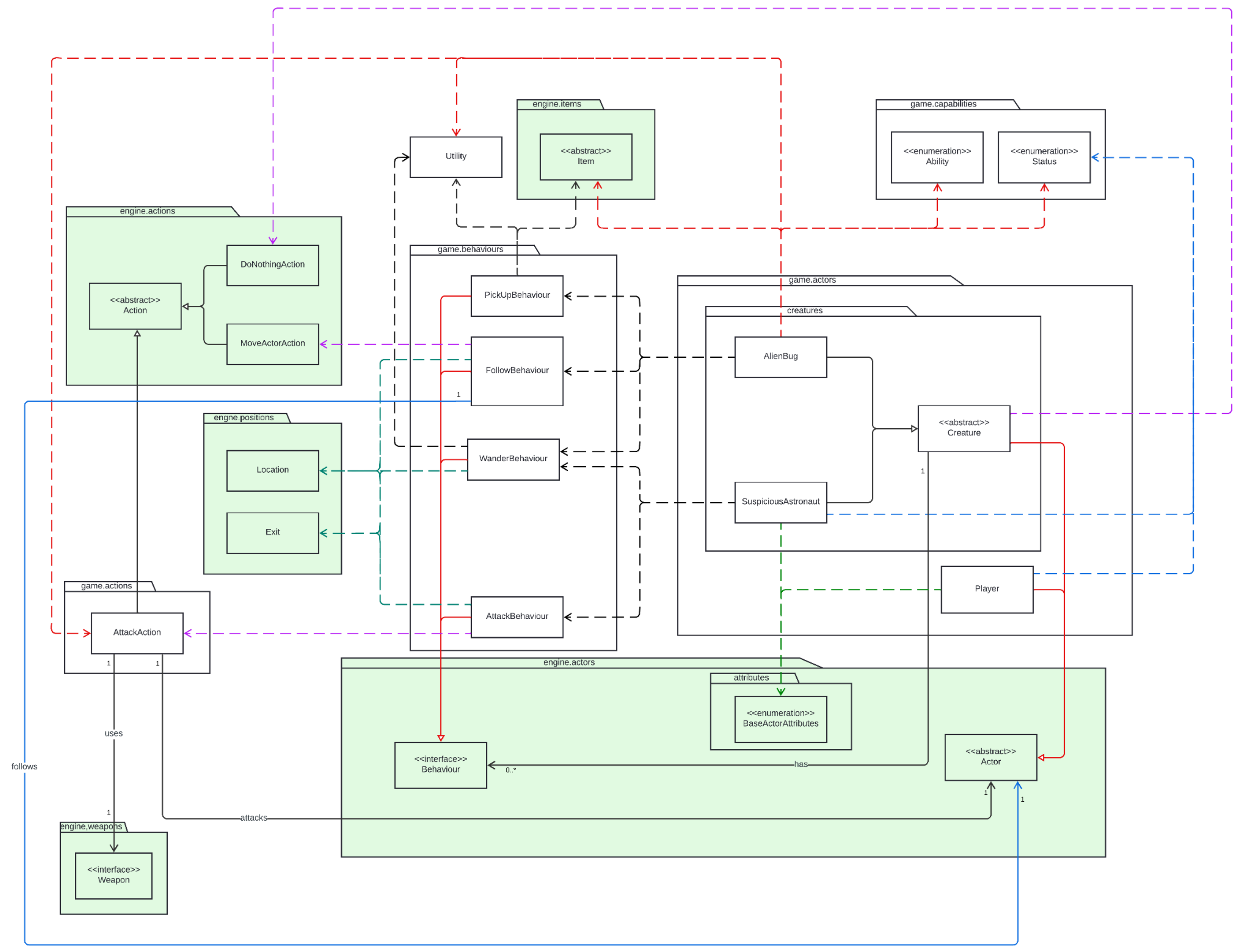


Requirement 1 Sequence Diagram: This sequence diagram is specific for the scenario where the Application creates a HunstmanSpider in the main() method.

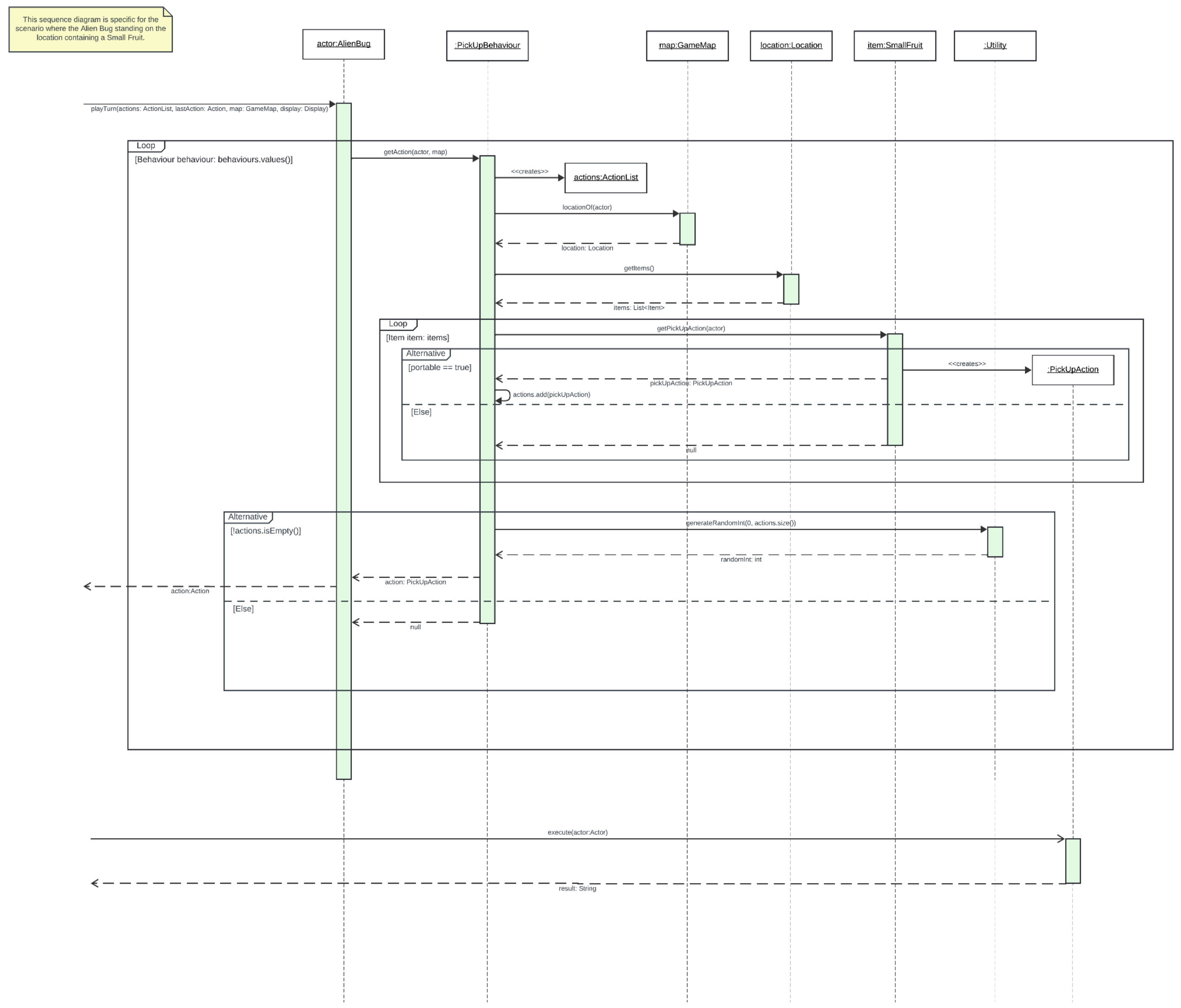
Classes Modified / Created	Roles and Responsibilities	Rationale
Spawner	An interface of a creature factory for spawning creatures. Relationships: 1. It depends on the Location class to know the actual location for new creature to spawn.	**Alternative Solution:** In the tick() method for Crater, the Crater class itself spawns a Creature (HunstmanSpider, AlienBug or SuspiciousAstronaut) if there isn't an actor already at the location. **Finalised Solution:** The 'factory method' design pattern was used to implement the spawning of Creatures. Rather than spawn the creature itself, the Crater class has an attribute to store a spawner used to spawn the corresponding Creature. This spawner is determined using constructor injection. Note that it only creates a Spawner which only generates one type of Creature since we are told "*Each crater instance can only spawn one type of creature*". The Spawner interface was created as a 'blueprint' for the Creature-specific spawners that were created later on. There is only one method in Spawner interface, being spawn(), such that the sole responsibility of any classes that implement Spawner would be to spawn their associated Creature with a specific probability. **Reasons for Decision:** 1. Each implementation of the Spawner would have the sole responsibility of spawning the (one and only one) Creature relevant to that Spawner. This satisfies SRP unlike our old implementation, where Crater was responsible for spawning different Creatures. In the new implementation, the Creature is now only responsible for behaving as the specified creature, and no longer has to deal with spawning itself. *(Single Responsibility Principle)* 2. Different creatures may have different spawning conditions and constraints. Hence, Spawner interface is implemented instead of an abstract class which is more expensive to use as we need to maintain the common attribute and logic. By using the Spawning interface, each creature implements its own code with specific conditions. Hence, when a new creature is introduced with a different
HuntsmanSpiderSpawner (implements Spawner)	Class representing a spawner to spawn Huntsman Spider. Relationships: 1. It implements Spawner interface and depends on HuntsmanSpider class as it can spawn a creature which is Huntsman Spider.	
AlienBugSpawner (implements Spawner)	Class representing a spawner to spawn Alien Bug. Relationships: 1. It implements Spawner interface and depends on AlienBug class as it can spawn a creature which is Alien Bug.	
SuspiciousAstronautSpawner (implements Spawner)	Class representing a spawner to spawn Suspicious Astronaut.	

	Relationships: 1. It implements Spawner interface and depends on SuspiciousAstronaut class as it can spawn a creature which is Suspicious Astronaut.	condition, it only needs to create a new class by implementing Spawner interface and invoke the spawn() method, without any modification to the Crater class. *(Open-closed Principle)* 3. By implementing the Spawner interface, each creature's spawner is forced to implement all non-default methods in the Spawner interface with their own code. Hence, those creatures' spawner can be substituted with the Spawner type in the attributes of the Crater class without breaking their expected behaviour as well as the other's. *(Liskov substitution Principle)* 4. The Spawner interface is small enough and contains only the methods needed by each creature's spawner. Hence, subclasses that implement it are not forced to implement methods they are not required to. *(Interface Segregation Principle)* 5. The probability of the creature being spawned from a crater is stored as a constant attribute in each creature class. These attributes are given meaningful names to avoid using 'magic numbers' and enhance code maintainability. Hence, developers understand the meaning of these values and can easily change them in one place without affecting other parts of the codebase. *(Connascence of Meaning)* **Limitations and Tradeoffs:** 1. A potential limitation of this implementation is that as the application scales, if more creatures are added, the corresponding number of spawners would need to be created corresponding to those creatures, which could become tedious to maintain and trace the flow of the implementation logic. However, this can be improved by having a sequence diagram without compromising with the pros.

Req 2 - The imposter among us



Requirement 2 Class Diagram



Requirement 2 Sequence Diagram: Scenario where the Alien Bug is standing on the location containing a Small Fruit.

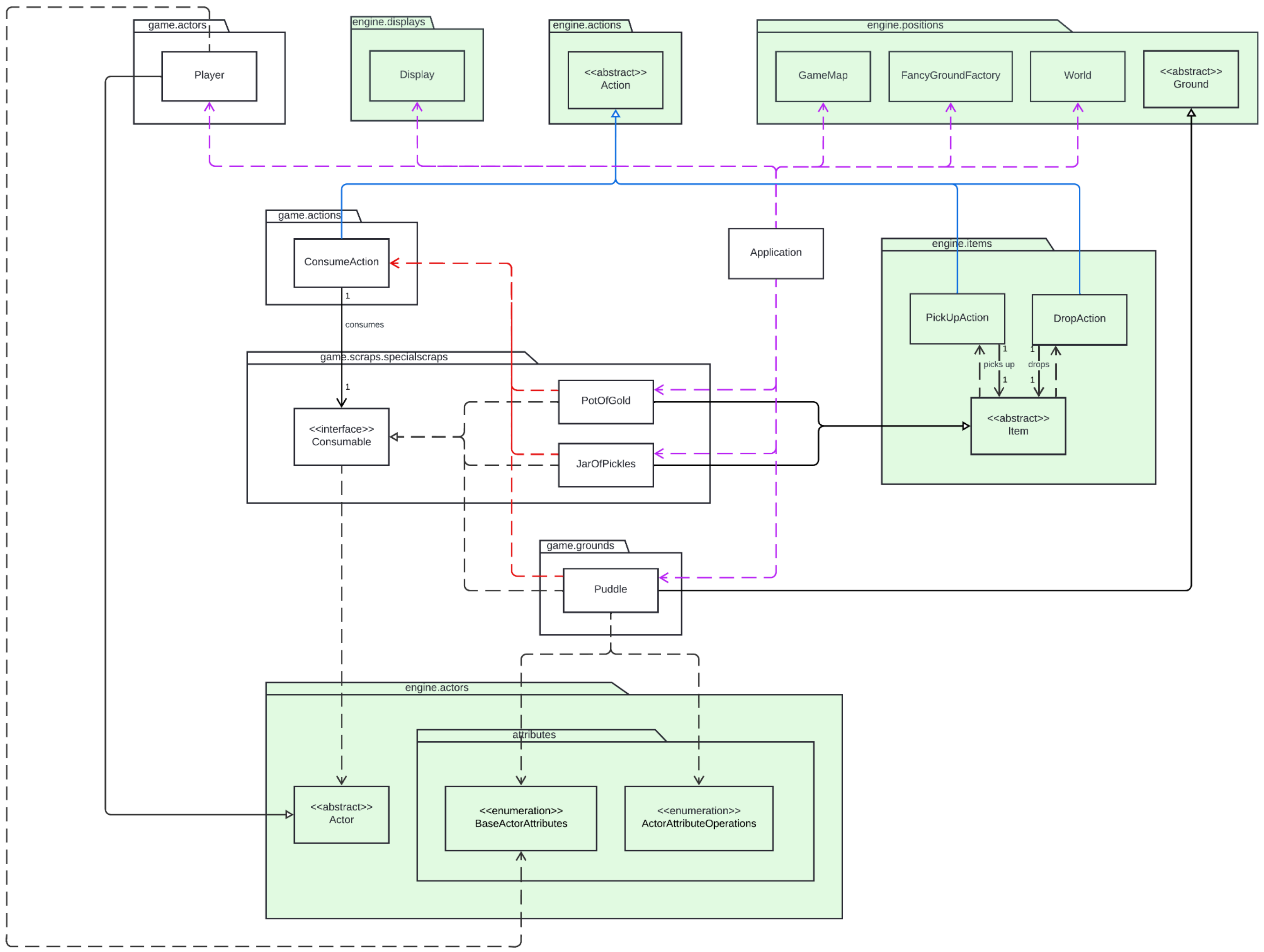
Classes Modified / Created	Roles and Responsibilities	Rationale
Alien Bugs (extends Creature)	Class representing the Alien Bugs creature that can be spawned by the Crater. Relationships: 1. It extends Creature abstract class to be spawned from the crater by its spawner. 2. It depends on PickUpBehaviour class, FollowBehaviour class and WanderBehaviour class to pick up scraps, follow player and wander around. 3. It depends on Item abstract class as it can store items in its item inventory. 4. It depends on the AttackAction class to generate an attack action for the player to attack it. 5. It depends on Status enum class to indicate itself is hostile to player. 6. It depends on Ability enum class to indicate itself can enter the spaceship.	**Alternative Solution:** When the bug is unconscious, it drops all the items from its inventory by directly creating a DropAction for each item and executing it. Implement the details of the killing ability of the suspicious astronaut in the Suspicious Astronaut class. **Finalised Solution:** Using a single if statement to check the status of the nearby actor in the overriding method allowableActions(), the Alien Bug and Suspicious Astronaut can be attacked by the actor by generating the AttackAction. When the Alien Bug is unconscious, it drops all the items from its inventory by executing the item.getDropAction(). For Suspicious Astronaut, implement its killing ability by adding the AttackBehaviour to the behaviours TreeMap of Suspicious Astronaut. To deal damage of value maximum health of the nearby actor, update the damage value of its intrinsic weapon by overriding the allowableActions() method in the Actor class. Although the Suspicious Astronaut can be attacked by the player in general, its capability to instantly kill the player makes it impossible to be attacked. Thus, the implementation of attackable by the player can be omitted. **Reasons for Decision:** 1. Since the maximum health of the Player might change in the future, we update the damage value whenever there is an actor nearby by overriding allowableActions() method instead of directly assigning the current maximum health of Player. This prevents the damage value and the maximum health from being connascent as changing the maximum health of the actor does not necessitate changing the code for the damage value of the intrinsic weapon of Suspicious Astronaut in order to preserve overall correctness. *(Connascence of Value)*
SuspiciousAstronaut (extends Creature)	Class representing the Suspicious Astronaut creature that can be spawned by the Crater. Relationships: 1. It extends Creature abstract class to be spawned from the	

	crater by its spawner. 2. It depends on FollowBehaviour class and WanderBehaviour class to follow player and wander around. 3. It depends on Status enum class to indicate itself is hostile to player. 4. It depends on BaseActorAttributes enum class to get the maximum health of base actor.	2. Instead of directly generating a new DropAction for each item, invoking the getDropAction() method in the Item class helps reduce the dependency of the Alien Bug class on DropAction. *(Dependency Inversion Principle)* **Limitations and Tradeoffs:** 1. The order of executing allowableActions() in the Player class and getIntrinsicWeapon() in SuspiciousAstronaut class to let the Suspicious Astronaut kill the intern is important since the damage of its intrinsic weapon will only be updated once the allowableActions() is called. It might not behave correctly if the order is wrong. *(Connesance of Execution)*
PickUpBehaviour (extends Behaviour)	Class representing the pickup behaviour of an actor which creates PickUpAction when the actor standing on the ground containing portable items. Relationships: 1. It extends Behaviour class to determine the conditions needed to perform a pick-up action.	**Alternative Solution:** Combine all the behaviours of the Alien Bug (ie. pickup item, follow intern and wander) in a single class by implementing the details of the pickup and follow behaviours in the same class with the existing wander behaviour. To implement the pickup behaviour, create and add a PickUpAction directly to the ActionList. To implement the follow behaviour, use the if statement and instanceOf() method to check whether the actor within its surroundings is the player. **Finalised Solution:** Create a new class for PickUpBehaviour which inherits from the abstract class Behaviour and all the details of the pickup behaviour are implemented in this class. In the PickUpBehaviour class, loop through the items found at its current location and invoke its getPickUpAction() to generate PickUpAction for each item. In the FollowBehaviour class, use a single if statement with Status.FOLLOWABLE to determine whether the nearby actor is followable by the Alien Bug. **Reasons for Decision:** 1. Separating each behaviour into its own class keeps each behaviour class focused on a single responsibility which allows easier extension and modification of without affecting other behaviours. *(Single Responsibility Principle)*
FollowBehaviour (extends Behaviour)	Class representing the behaviour of an actor which starts following the other actor within its surroundings. Relationships: 1. It extends Behaviour class to determine the conditions needed to perform a move-actor action.	

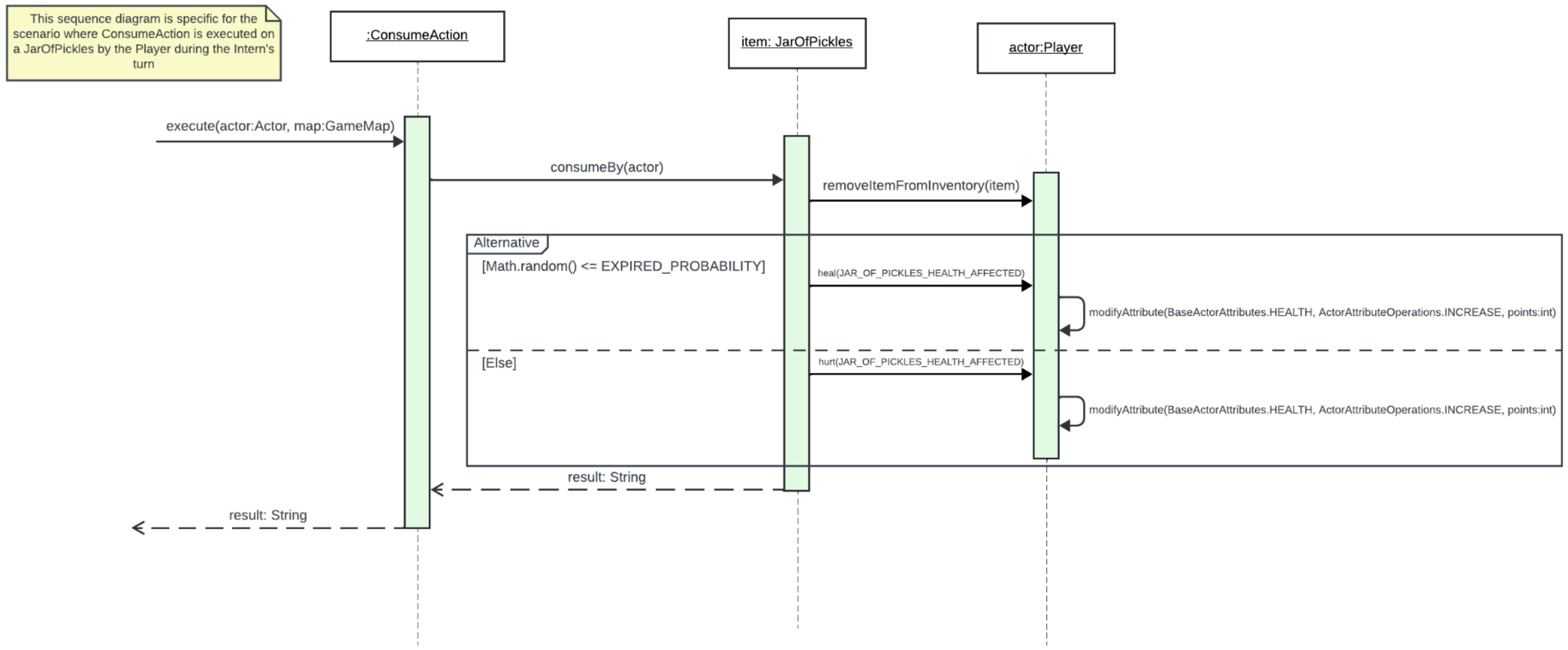
	2. It depends on MoveActorAction class to keep following some followable actors.	2. Since there may be different creatures sharing the same behaviours, separately implementing the behaviour classes can significantly reduce duplicate codes as developers only have to add relevant behaviours for the new creature by putting them into its behaviours TreeMap. *(DRY Principle)* 3. Since the Alien Bug may have additional behaviours in the future with different priorities, extending the Behaviour abstract class allows creating only a single TreeMap of type Behaviour to store all the behaviours the creature has in a sorted manner based on their priorities. Thus, the Alien Bug concrete class will not be associated with each and every behaviour class which helps in code maintenance. *(Dependency Inversion Principle)* 4. The addition of new behaviour can be implemented through inheritance and polymorphism without altering the existing code which promotes code maintainability and reduces the risk of introducing bugs when extending the new behaviour. *(Open-closed Principle)* 5. The subclasses of the Behaviour class can be substituted for the base class without affecting the expected behaviours of the creature. This flexibility enables different types of behaviour to be performed interchangeably within the Creature class. *(Liskov Substitution Principle)* 6. The Behaviour class is split from the Action class to ensure the behaviour classes only implement the details of performing the behaviours rather than the details of the actual actions. *(Interface Segregation Principle)* 7. To maintain the flexibility to follow any creature within its surroundings, the FollowBehaviour class has an attribute to set the target to follow using constructor injection to reduce the dependency of the FollowBehaviour class on different actor classes. Once it finds a target to follow, it will follow the same target until it dies or game over so constructor injection is used instead of setter

		injection to prevent further change on the target. *(Dependency Injection Principle)* 8. The priorities of the behaviours are stored as attributes in the Alien Bug class with some meaningful name to avoid magic numbers *(Connascence of Meaning)* and enhance code maintainability since developers are only required to change the priority value at this single attribute and other places depending on this priority will then utilise the new value. *(Connascence of Value)* 9. Instead of directly generating a new PickUp Action for each item, invoking the getDropAction() method in the Item class helps reduce the dependency of the PickUpBehaviour class on PickUpAction. *(Dependency Inversion Principle)* Additionally, due to encapsulation, PickUpBehaviour has no direct access to the item's portable attribute, thus this check can be done in the item.getPickUpAction() before creating a PickUpAction for that item. **Limitations and Tradeoffs:** 1. Having various behaviour classes might increase dependency management overhead which might lead to challenges in tracking and managing dependencies and result in a fragile codebase.

Req 3 - More scraps



Requirement 3 Class Diagram


This sequence diagram is specific for the scenario where ConsumeAction is executed on a JarOfPickles by the Player during the Intern's turn
:ConsumeAction
item: JarOfPickles
actor:Player
execute(actor:Actor, map:GameMap)
consumeBy(actor)
removeItemFromInventory(item)
Alternative
[Math.random() <= EXPIRED_PROBABILITY]
heal(JAR_OF_PICKLES_HEALTH_AFFECTED)
modifyAttribute(BaseActorAttributes.HEALTH, ActorAttributeOperations.INCREASE, points:int)
[Else]
hurt(JAR_OF_PICKLES_HEALTH_AFFECTED)
modifyAttribute(BaseActorAttributes.HEALTH, ActorAttributeOperations.INCREASE, points:int)
result: String
result: String

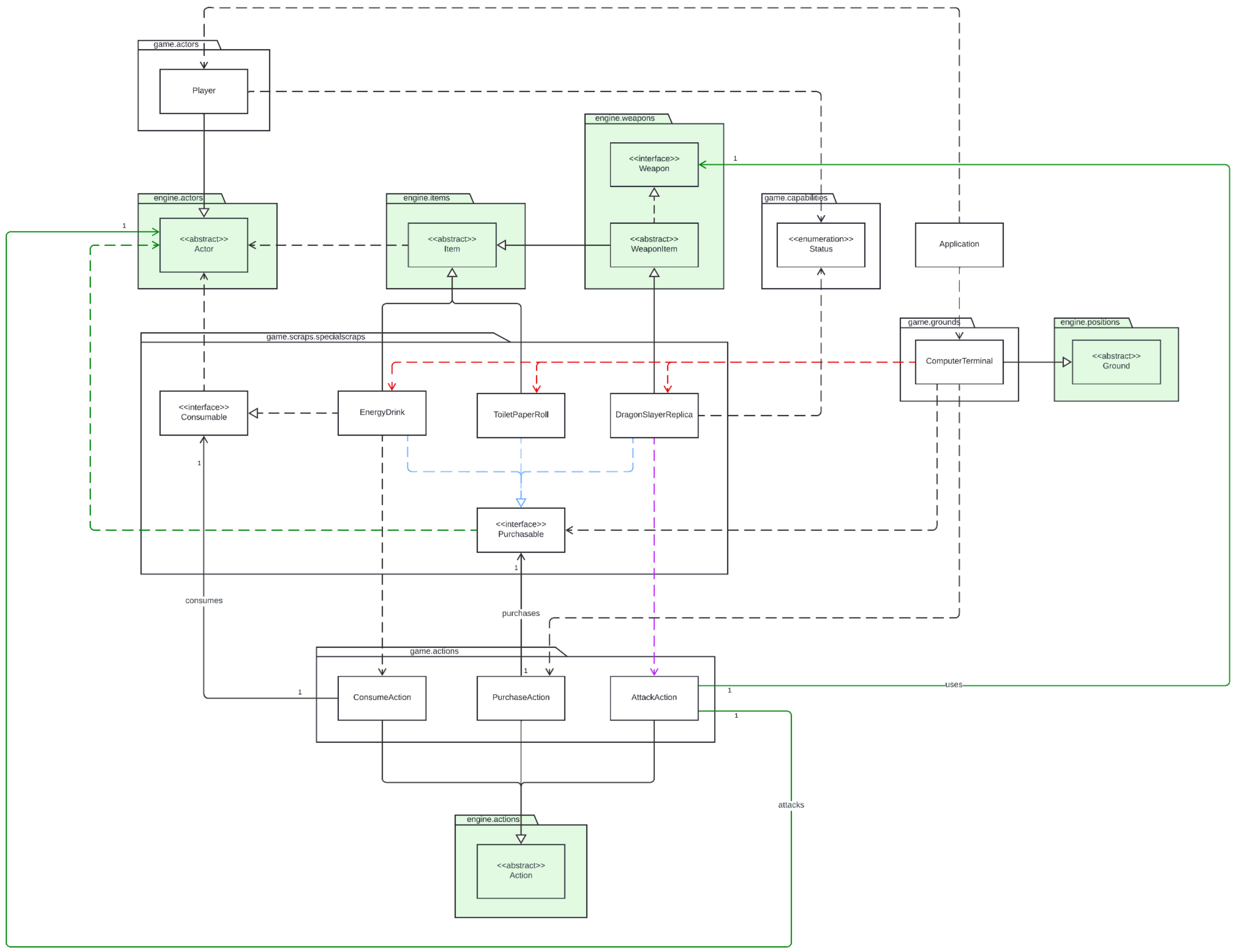
Requirement 3 Sequence Diagram: Jar of Pickles Sequence Diagram when Consume Action is executed

Classes Modified / Created	Roles and Responsibilities	Rationale
JarOfPickles (extends Item implements Consumable)	Class representing Relationships: 1. They extend Item as both can be picked up and dropped by the Intern and implement Consumable as both can be consumed by the intern and modify their attributes 2. They depend on the ConsumeAction class to generate a consume action for an actor to consume it.	**Alternate Solution:** 1. Creating a single Item class and using if else statements for the methods to determine the effects based on the item display character. 2. Adding onto any of the pre-existing scraps such as Fruits or Metal Sheets and using If Else to manage the items and features **Finalised Solution:** 1. Create Jar of Pickles and Pot of Gold from ConsumableItem, which already implements the Consumable implement to ensure the consumedby method is to be implemented by both classes **Reasons for Decision:** 1. The alternate solution creates a tight coupling between the modules and the item display character since it would require changes to all the modules that rely on it if the format or structure of display character were to change (e.g. from character to numeric id). *(Connascence of Identity)* 2. There should not be a single class that manages both the Jar of Pickles and the Pot of Gold as it will ensure that not relevant methods and attributes will be implemented by either class. *(Single Responsibility Principle)* 3. By creating new classes, it ensures that the code is scalable for future extensions as there is not a requirement to modify previous parts of the code when adding new special scraps. *(Open-closed Principle)* 4. When looping through the items in the Player's inventory, the instances accepted are Objects of instance Items, as the Picking Up and Dropping of Jar of Pickles and Pot of Gold items did not cause the application to crash, Liskov's Substitution principle have been adhered to whereby the children classes were successfully upcasted into items of the parent class *(Liskov substitution Principle)*
PotOfGold (extends Item implements Consumable)		

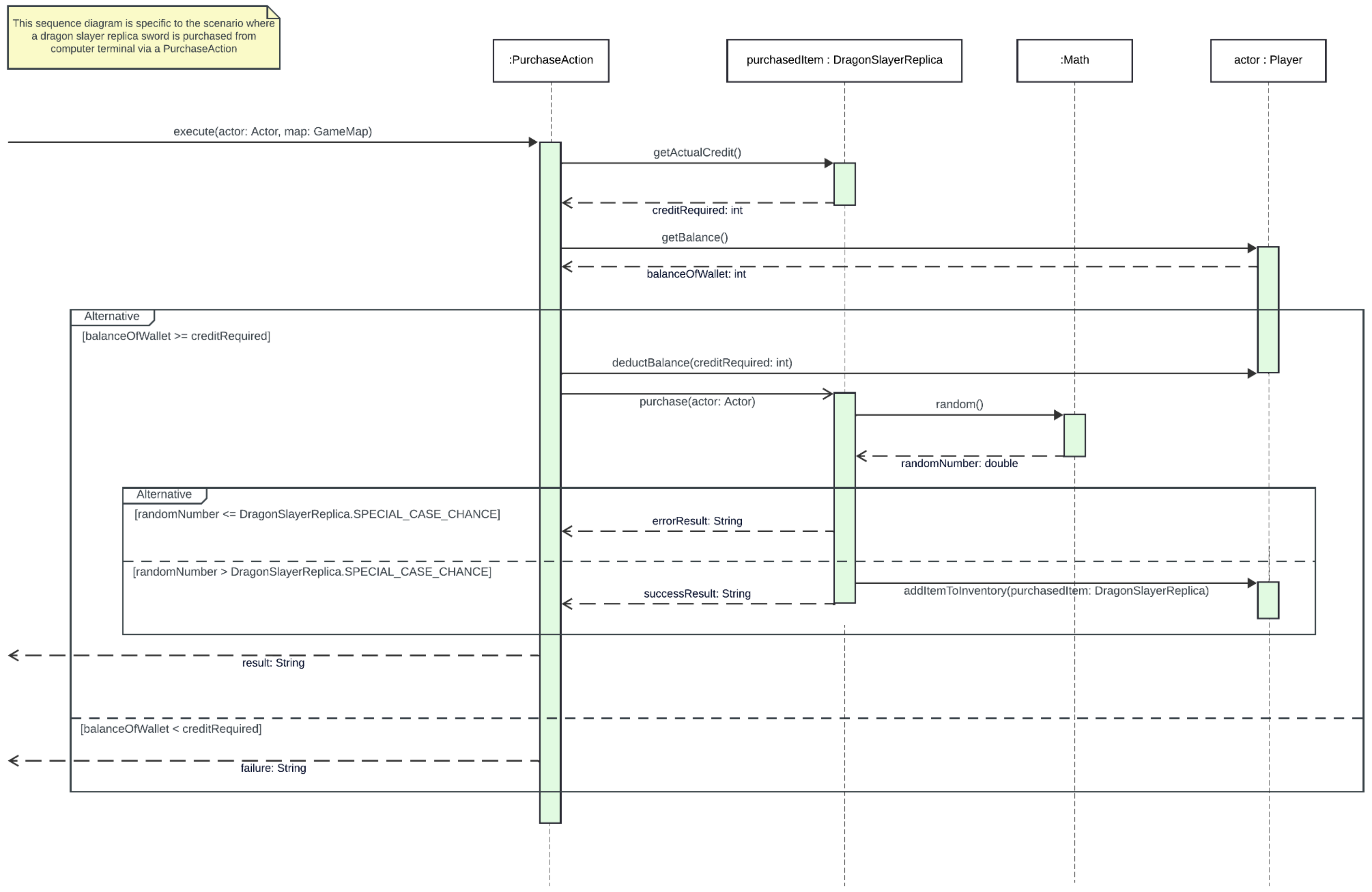
		5. The two classes implemented were only made to implement required interfaces and methods and neither depended on classes that had no relation to Items in any way. *(Interface Segregation Principle)* 6. By depending on the Consumable interface rather than concrete subclasses like JarOfPickles and PotOfGold, the ConsumeAction class is decoupled from the specific effects of consuming different types of consumable objects. This promotes flexibility, as ConsumeAction can be used with any object that implements the Consumable interface. *(Dependency Inversion Principle)* 7. To ensure that magic numbers are not present in the code when possible, the numbers are all assigned to variables with meaningful names in each of the relevant classes. *(Connascence of Meaning)* **Limitations and Tradeoffs:** 1. The abundance of classes that are required to be created as more items that implement Consumable are implemented might make the code hard to read. Hence, adequate and concise Javadoc and documentation is required to negate the potential issues that are unfortunately unavoidable to ensure that the SOLID principles have not been violated. 2. Using an interface for Consumable instead of an abstract class might introduce duplication of code across multiple subclasses which violates DRY principle as there might exist similar implementations of consuming effect for several consumable substances.
Puddle (extends Ground implements Consumable)	Class representing Relationships: 1. It extends the Ground class	**Alternate Solution:** 1. Create an item that cannot be picked up nor dropped but can be consumed on the Ground even without looping through player inventory or refactor the ground class to be a non-portable item. 2. Have the GameMap constantly check if the player is standing on a puddle and return the corresponding allowable action

		Finalised Solution: 1. Puddle implements Consumable, which allows the player to consume Ground whilst standing on it. The allowable action of ground is overridden to ensure that only one Puddle, the Puddle the intern is standing on can return the Consume Action. **Reasons for Decision:** 1. Puddle, and the newly added code, will only affect Puddle whilst ensuring that Puddle does not implement method unrelated to enabling the Intern to drink from a Puddle *(Single Responsibility Principle)* 2. Moreover, the creation of Puddle did not require refactoring in the higher level classes such as Ground and Consumable, enforcing the principle that it is important to ensure that parent classes are open to extension but closed to modification *(Open-closed Principle)* and the Dependency Inversion Principle when higher level modules are not dependant on future changes that might occur in Puddle. *(Dependency Inversion Principle)* 3. Puddle was easily accepted as an item that implements Consumable in the Consume Action, which proves that it was successfully upcasted into the class it implemented*(Liskov substitution Principle)* 4. The Consumable interface that was implemented by Puddle was a small compact interface with only one method and hence did not consist of methods irrelevant to Objects that should be consumable *(Interface Segregation Principle)* **Limitations and Tradeoffs:** 1. Many crucial methods of Puddle as a child of Ground have been overwritten, which might confuse when testing puddle without checking the code nor the Javadoc. Regardless, it is even worse of a practice to create two classes to manage the many functions of Puddle as that would ultimately cause even more confusion.

Req 4 - Static factory's staff benefits



Requirement 4 Class Diagram


This sequence diagram is specific to the scenario where a dragon slayer replica sword is purchased from computer terminal via a PurchaseAction
:PurchaseAction
purchasedItem : DragonSlayerReplica
:Math
actor : Player
execute(actor: Actor, map: GameMap)
getActualCredit()
creditRequired: int
getBalance()
balanceOfWallet: int
Alternative
[balanceOfWallet >= creditRequired]
deductBalance(creditRequired: int)
purchase(actor: Actor)
random()
randomNumber: double
Alternative
[randomNumber <= DragonSlayerReplica.SPECIAL_CASE_CHANCE]
errorResult: String
[randomNumber > DragonSlayerReplica.SPECIAL_CASE_CHANCE]
addItemToInventory(purchasedItem: DragonSlayerReplica)
successResult: String
result: String
[balanceOfWallet < creditRequired]
failure: String

Requirement 4 Sequence Diagram: Scenario where a dragon slayer replica sword is purchased from computer terminal via a PurchaseAction

Classes Modified / Created	Roles and Responsibilities	Rationale
EnergyDrink (extends Item, implements Consumable and Purchasable)	Class representing a portable energy drink which can be purchased from a computer terminal, and able to heal players if consumed. Relationships: 1. It extends the Item class to achieve the portable functionality. 2. It implements the Consumable interface as it can be consumed. 3. It implements the Purchasable interface as it can be purchased. 4. It depends on the ConsumeAction class to generate a consume action for the player to consume it.	**Alternate Solution:** 1. In each class of the item that is purchasable, override the allowableActions() in the Item class by adding the PurchaseAction into the ActionList. 2. Create a PurchasableItem abstract class and all the items that are purchasable inherit from it. a. Implement the method purchase() in the abstract class to avoid repeated code in each purchasable item class. b. Declare an abstract method purchase() in the abstract class for the details of the purchase action. **Finalised Solution:** Create an interface Purchasable which defines a specific method purchase() for the implementation of purchasing. Then, create a new class for each purchasable item which inherits from a specific class (such as WeaponItem) based on their usage and provides its own implementation of the purchase method by implementing the Purchasable interface. Lastly, create a concrete class PuchaseAction which inherits from the Action class, with an attribute to store the purchasable item. In the override execute() method, the purchase() method is invoked to add the item to the actor's inventory and the payment processing logic is implemented. **Reasons for Decision:** 1. By separating the purchase action details from the Purchasable class, we ensure that each class focuses on a single responsibility, making it easier to understand, extend, and modify each class independently. The PurchaseAction class is responsible for managing the payment processing logic, while each purchasable item is responsible for managing its purchase constraints. *(Single Responsibility Principle)* 2. By implementing the payment processing logic in PurchaseAction, we can avoid repeated code, as there's no need to duplicate the logic in each purchasable item class in the same way. *(DRY Principle)*
DragonSlayerReplica (extends WeaponItem, implements Purchasable)	Class representing a portable dragon slayer replica sword which can be purchased from a computer terminal, and able to attack hostile creatures. Relationships: 1. It extends the WeaponItem class to achieve the portable and weapon functionality. 2. It implements the Purchasable interface as it can be purchased. 3. It depends on the AttackAction	

	class to perform an attack action for player to attack hostile creatures.	3. Different purchasable items may have different purchase constraints. Hence, a Purchasable interface is created instead of an abstract class which requires expensive maintenance of common attributes and methods. By using the Purchasable interface, each purchasable item implements its own purchase logic along with its purchase constraint. Hence, when a new purchasable item is introduced with a different purchase logic, it only needs to create a new purchasable item class by implementing Purchasable interface with the purchase() method, without any modification to the PurchaseAction class. *(Open-closed Principle)* 4. By having the Purchasable interface, only item classes that can be purchased need to implement it. This forces purchasable items to implement all non-default methods in the Purchasable interface with their own code. Consequently, purchasable items can be substituted with the Purchasable type in attributes of the PurchaseAction class without breaking their expected behaviour as well as the whole system. *(Liskov Substitution Principle)* 5. The Purchasable interface is small enough and contains only the methods needed by the purchasable item which implements it, in other words, subclasses that implement it are not forced to implement methods they are not required to. This enhanced the clarity and maintainability of code compared to having a large interface for all purchasable items (eg. including purchasing, attacking and consuming methods). *(Interface Segregation Principle)* 6. By using the Purchasable interface , multiple checks for the type of purchasable item are not required, as all the purchase details are implemented by the subclasses. Consequently, the PurchaseAction class no longer needs to depend on every purchasable subclass, decoupling it from the specific details of purchasing implementations *(Dependency Inversion Principle)*. Additionally, by having an attribute purchasedItem in PurchaseAction class and setting it into the selected purchasable item using constructor injection, it reduces the dependency of the
Toilet Paper Roll (extends Item, implements Purchasable)	Class representing a portable toilet paper roll which can be purchased from a computer terminal. Relationships: 1. It extends the Item class to achieve the portable functionality. 2. It implements the Purchasable interface as it can be purchased.	
Purchasable	An interface for some item that can be purchased from the computer terminal. Relationships: 1. It depends on the Actor class to know which actor purchased the purchasable item.	
PurchaseAction (extends Action)	Class representing the purchase action. Relationships: 1. It extends the Action class to perform an action. 1. It is associated with the Purchasable interface to perform the purchase action on	

	the purchasable item.	PurchaseAction class on different purchasable item classes *(Dependency Injection Principle)*. 7. The amount of credits and the probability of each special case (some items have additional information, such as the amount of credits in a special case) are stored as attributes in each purchasable item class. These attributes are given meaningful names to avoid using 'magic numbers' and enhance code maintainability. Hence, developers understand the meaning of these values and can easily change them in one place without affecting other parts of the codebase. *(Connascence of Meaning)* **Limitations and Tradeoffs:** 1. Having the EnergyDrink class implement multiple interfaces, such as Purchasable and Consumable, raises concerns about the clarity of its responsibilities and the potential for design complexity which might require additional time and effort for developers to locate its specific functionality and navigate the codebase.
ComputerTerminal (extends Ground)	An abstract class representing a computer terminal which can purchase purchasable items. Relationships: 1. It extends Ground class to achieve performing purchase action on each purchasable item by overriding the allowableActions() method. 2. It depends on the Purchasable interface to store a list of purchasable items. 3. It depends on the PurchaseAction class to perform	**Alternate Solution:** Create a list of items that can be purchased from the computer terminal and store the list as an attribute in the generic Ground class, constantly check if the player is near the computer terminal to find all available purchase actions for each purchasable item in the allowableActions() method and return them. **Finalised Solution:** Create a ComputerTerminal concrete class that inherits from the Ground abstract class. Then, create a method to return a list of purchasable items from the computer terminal instead of storing the list as an attribute to reduce the association relationship to a dependency relationship towards the Purchasable interface. Lastly, override the allowableActions method in the Ground abstract class and return all purchase actions for each purchasable item. **Reasons for Decision:**

	purchase action on purchasable items.	1. By creating a separate ComputerTerminal class, the responsibility of managing computer terminal functionality is encapsulated within this class, while the generic ground class handles generic ground-related functionality. This encapsulation prevents the mixing of functionalities from other types of ground into the generic ground class. *(Single Responsibility Principle)* 2. A new class ComputerTerminal is created by extending the Ground class to handle specific functionality related to computer terminals. This is done by overriding methods in ComputerTerminal instead of altering the behaviour of the Ground abstract class to include specific functionality for computer terminals, such as adding all purchase actions into the allowableActions() method in the generic ground class. *(Open-closed Principle)* 3. The ComputerTerminal class inherits from the Ground abstract class and provides its own implementation of the allowableActions() method. This ensures that instances of ComputerTerminal can be substituted for instances of Ground without changing the expected behaviour of the system. *(Liskov substitution Principle)* 4. By extending the Ground abstract class, the ComputerTerminal class is not forced to depend on or implement the methods it is not required (eg. getPickUpAction, getDropAction()). *(Interface Segregation Principle)* 5. The ComputerTerminal class depends on the Purchasable interface, which represents a high-level abstraction. This allows the class to be decoupled from specific implementations of purchasable items, promoting flexibility and maintainability. *(Dependency Inversion Principle)* 6. The ComputerTerminal class has a method to return a list of purchasable items instead of storing the list as a class attribute. This leads to the dependency relationship being inverted from a concrete association to a dependency on an abstraction.

		Limitations and Tradeoffs: 1. The items that can be purchased from this computer terminal are now returned by a method in the form of a list, with items manually added to the list, reflecting a brute force implementation. As the number of computer terminals and purchasable items grows, the manual approach of maintaining lists of purchasable items may become unsustainable. Additionally, the current implementation may not be easily reusable for other computer terminals that offer different sets of purchasable items.

Assignment 2 - Contribution Log

FIT2099 - CL_Lab26Team2 - Assignments' Contribution Logs

Task/Contribution	Contribution type	Planning Date	Contributor	Status	Actual Completion Date	Extra notes
Meeting discussion						
Revised A1 UML diagram and discuss reflecting changes based on feec	Brainstorm	23/04/2024	EVERYONE	DONE	23/04/2024	
Distribute tasks related to the changes made in A1	Discussion	23/04/2024	EVERYONE	DONE	23/04/2024	
Distribute task after reviewing the assignment brief	Discussion	23/04/2024	EVERYONE	DONE	23/04/2024	
Briefly create a basic UML for each task: create new class needed	UML diagram	23/04/2024	EVERYONE	DONE	23/04/2024	
Discuss the type of classes, ie concrete, abstract, interface or enum	Brainstorm	23/04/2024	EVERYONE	DONE	23/04/2024	
Confirming 'factory' concept and the scope of sequence diagram with tu	Discussion	07/05/2024	EVERYONE	DONE	07/05/2024	
Refactoring of Assignment 1						
Add Spawner interface for Creatures	Implementation	04/05/2024	Jeevan Vetteel Tharun	DONE	30/04/2024	Put inside 'spawners' package
Add Huntsman Spider Spawner	Implementation	04/05/2024	Jeevan Vetteel Tharun	DONE	30/04/2024	Put inside 'spawners' package
Implementation of Inheritree Grow	Implementation	04/05/2024	Wei Xin Soh	DONE	04/05/2024	
Remove Consumable Item	Implementation	04/05/2024	Suet Yuen Chin	DONE	02/05/2024	Edited BigFruit and SmallFruit and corresponding Javadoc as well
Requirement 1						
Create Alien Bug	Implementation	04/05/2024	Jeevan Vetteel Tharun	DONE	03/05/2024	Merged to main on 07/05/2024
Create Suspicious Astronaut	Implementation	04/05/2024	Jeevan Vetteel Tharun	DONE	03/05/2024	Merged to main on 07/05/2024
Create Alien Bug Spawner	Implementation	04/05/2024	Jeevan Vetteel Tharun	DONE	04/05/2024	Merged to main on 07/05/2024
Create Suspicious Astronaut Spawner	Implementation	04/05/2024	Jeevan Vetteel Tharun	DONE	04/05/2024	Merged to main on 07/05/2024
Javadoc	Documenting	07/05/2024	Jeevan Vetteel Tharun	DONE	07/05/2024	...
Requirement 2						
Alien Bug picks up scraps	Implementation	04/05/2024	Wei Xin Soh	DONE	02/05/2024	
Alien Bug follows player after player nears Alien Bug for the first time	Implementation	04/05/2024	Wei Xin Soh	DONE	01/05/2024	
Alien Bug drops all scraps when defeated	Implementation	04/05/2024	Wei Xin Soh	DONE	02/05/2024	
Alien Bug can enter spaceship	Implementation	04/05/2024	Wei Xin Soh	DONE	02/05/2024	
Suspicious Astronaut instantly kills intern when intern is one exit away	Implementation	04/05/2024	Wei Xin Soh	DONE	04/05/2024	
Javadoc	Documenting	07/05/2024	Wei Xin Soh	DONE	07/05/2024	
Requirement 3						
Create Jar of Pickles	Implementation	04/05/2024	Suet Yuen Chin	DONE	02/05/2024	Changes to inheritance
Create Pot Of Gold	Implementation	04/05/2024	Suet Yuen Chin	DONE	02/05/2024	Changes to inheritance
Jar of Pickles hurt or heal players with a 50% Chance	Implementation	04/05/2024	Suet Yuen Chin	DONE	30/04/2024	
Pot Of Gold can be used to increase Intern's balance by 10	Implementation	04/05/2024	Suet Yuen Chin	DONE	30/04/2024	
Intern can drink water from the Puddle	Implementation	04/05/2024	Suet Yuen Chin	DONE	30/04/2024	
Drinking Puddle increases Intern's maximum health by 1	Implementation	04/05/2024	Suet Yuen Chin	DONE	30/04/2024	
Javadoc	Documenting	07/05/2024	Suet Yuen Chin	DONE	02/05/2024	
Requirement 4						
Create Computer Terminal	Implementation	04/05/2024	Ying Shan Khor	DONE	02/05/2024	
Create Purchasable Interface	Implementation	04/05/2024	Ying Shan Khor	DONE	24/04/2024	
Create PurchaseAction class	Implementation	04/05/2024	Ying Shan Khor	DONE	24/04/2024	
Create Energy Drink	Implementation	04/05/2024	Ying Shan Khor	DONE	24/04/2024	
Create Dragon Slayer Replica	Implementation	04/05/2024	Ying Shan Khor	DONE	24/04/2024	
Create Toilet Paper Roll	Implementation	04/05/2024	Ying Shan Khor	DONE	24/04/2024	
Process payment logic	Implementation	04/05/2024	Ying Shan Khor	DONE	30/04/2024	
Computer Terminal prints Energy Drink in normal case and special case	Implementation	04/05/2024	Ying Shan Khor	DONE	03/05/2024	
Computer Terminal prints Dragon Slayer Replica in normal case and spe	Implementation	04/05/2024	Ying Shan Khor	DONE	03/05/2024	
Computer Terminal prints Toilet Paper Roll in normal case and special c	Implementation	04/05/2024	Ying Shan Khor	DONE	03/05/2024	
Javadoc	Documenting	07/05/2024	Ying Shan Khor	DONE	07/05/2024	
Design Rationale Document						
Refactoring of Assignment 1						
Spawner interface changes to UML and Design Rationale	Documenting	04/05/2024	Jeevan Vetteel Tharun	DONE	30/04/2024	
Huntsman Spider Spawner changes to UML and Design Rationale	Documenting	04/05/2024	Jeevan Vetteel Tharun	DONE	30/04/2024	
Inheritree Grow changes to UML and Design Rationale	Documenting	07/05/2024	Wei Xin Soh	DONE	02/05/2024	
Removal Consumable Item changes to UML and Design Rationale	Documenting	07/05/2024	Suet Yuen Chin	DONE	01/05/2024	

59						
60	**Requirement 1**					
61	UML Diagram	Documenting	03/05/2024	Jeevan Vetteel Tharun	DONE	03/05/2024
62	Sequence Diagram	Documenting	03/05/2024	Jeevan Vetteel Tharun	DONE	03/05/2024
63	Design Rationale	Documenting	07/05/2024	Jeevan Vetteel Tharun	DONE	07/05/2024
64						
65	**Requirement 2**					
66	UML Diagram	Documenting	03/05/2024	Wei Xin Soh	DONE	03/05/2024
67	Sequence Diagram	Documenting	03/05/2024	Wei Xin Soh	DONE	03/05/2024
68	Design Rationale	Documenting	07/05/2024	Wei Xin Soh	DONE	07/05/2024
69						
70	**Requirement 3**					
71	UML Diagram	Documenting	03/05/2024	Suet Yuen Chin	DONE	01/05/2024
72	Sequence Diagram	Documenting	03/05/2024	Suet Yuen Chin	DONE	03/05/2024
73	Design Rationale	Documenting	07/05/2024	Suet Yuen Chin	DONE	07/05/2024
74						
75	**Requirement 4**					
76	UML Diagram	Documenting	03/05/2024	Ying Shan Khor	DONE	03/05/2024
77	Sequence Diagram	Documenting	03/05/2024	Ying Shan Khor	DONE	03/05/2024
78	Design Rationale	Documenting	07/05/2024	Ying Shan Khor	DONE	06/05/2024
79						
80	**Finalisation**					
81	Code Testing and Debugging	Code review	10/05/2024	EVERYONE	DONE	09/05/2024
82	Design Rationale Document Check	Code review	10/05/2024	EVERYONE	DONE	10/05/2024