FIT2099 Assignment 2 Design Rationale

CL_AppliedSession34_Group1

Bryan Chow Wei Qian - 33564280
Lachlan MacPhee - 33109176
Ethan Watts - 32985495
Michael Di Giantomasso - 32472498

Table of Contents

Design Goals

The following section contains the design goals of the system in a clear and concise manner. These provide a high-level overview of how the system is engineered in such a way that it meets the A2 requirements, but does so in a manner that conforms to professional software development standards. [1]

Extensibility

Ensuring the system's extensibility was a key goal. We wanted to make it easy to add new features, fauna, and scraps to the game without having to make large modifications to the existing codebase. By prioritising extensibility, the game is better able to evolve and grow over time without becoming bogged down by technical limitations. [1]

Maintainability

Maintaining the codebase over time was another crucial goal. We focused on writing clean, well-structured code and documenting the system comprehensively to ensure that it remains manageable and comprehensible. By prioritising maintainability, we aimed to prevent technical debt from accumulating and to facilitate efficient bug fixes, optimisations, and enhancements. [1]

Design Choices

This section is broken down by each key requirement and is accompanied by the UML diagrams (class and sequence) for these requirements so that there is a visual aid for the explanations. [1]

It describes the roles and responsibilities of new or significantly modified classes and how they satisfy the design goals. [1]

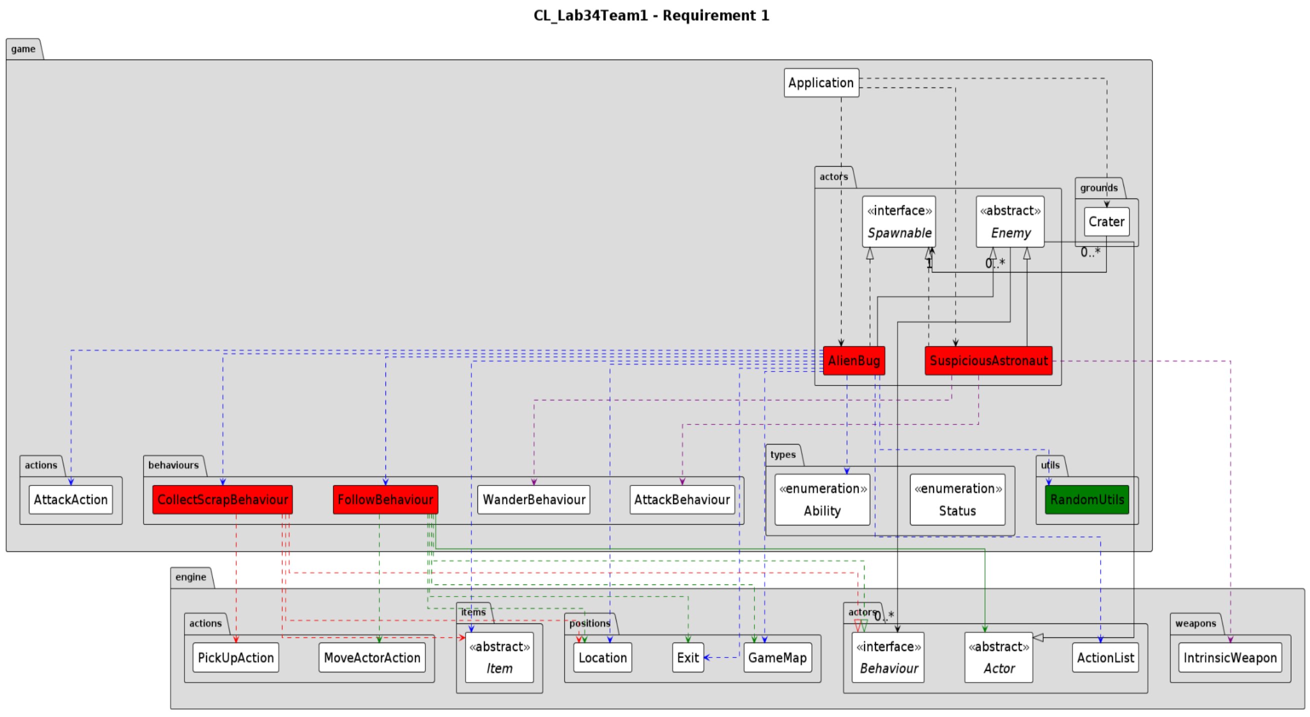
Note on UMLs

- Classes coloured red are new
- classes coloured green have been modified from their previous implementation.

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

In REQ1, the moon’s craters now can also spawn other creatures like Alien Bug and Suspicious Astronaut in addition to Huntsman Spiders.

UML



The diagram above represents an object-oriented system for REQ1.

Key Classes Introduced

- Crater - a class that spawns a given actor (currently just enemies) with a specific spawn chance
- Spawnable - an interface for spawnable actors that allows them to create new instances of themselves (for extensibility purposes)
- Spawner - a static utility class with code to spawn actors and items
- RandomUtils - another static utility class with code to get random exits or numbers of a specific type within a bound

Alternative Design Considered

My alternative design approach involved introducing an abstraction layer for the enemy spawning logic. A Spawner interface would be defined with a single method spawnEnemy(). Concrete classes would then implement this interface for each specific enemy type, such as HuntsmanSpiderSpawner, AlienBugSpawner and SuspiciousAstronautSpawner.
The Crater class would then have a reference to an instance of a Spawner implementation. Instead of directly handling the spawning logic within the Crater class, it would delegate the responsibility to the associated Spawner instance by invoking its spawnEnemy() method.

Pros:

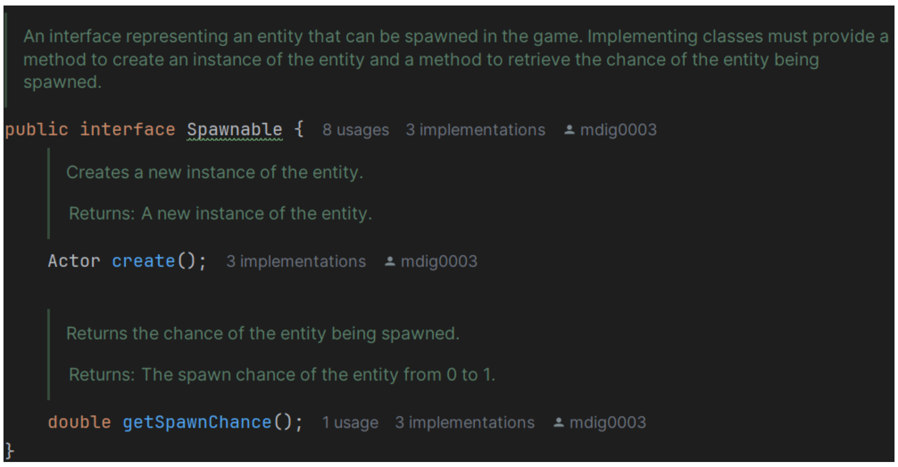
- Adheres to the Single Responsibility Principle (SRP) by separating the concerns of representing a crater and spawning enemies into different classes.
- New enemy types could be added by creating a new Spawner implementation without modifying existing code (OCP).

Cons:

- Increased complexity by introducing an additional abstraction layer and multiple classes.
- It may also incur some performance overhead due to the additional method dispatches and object creations.

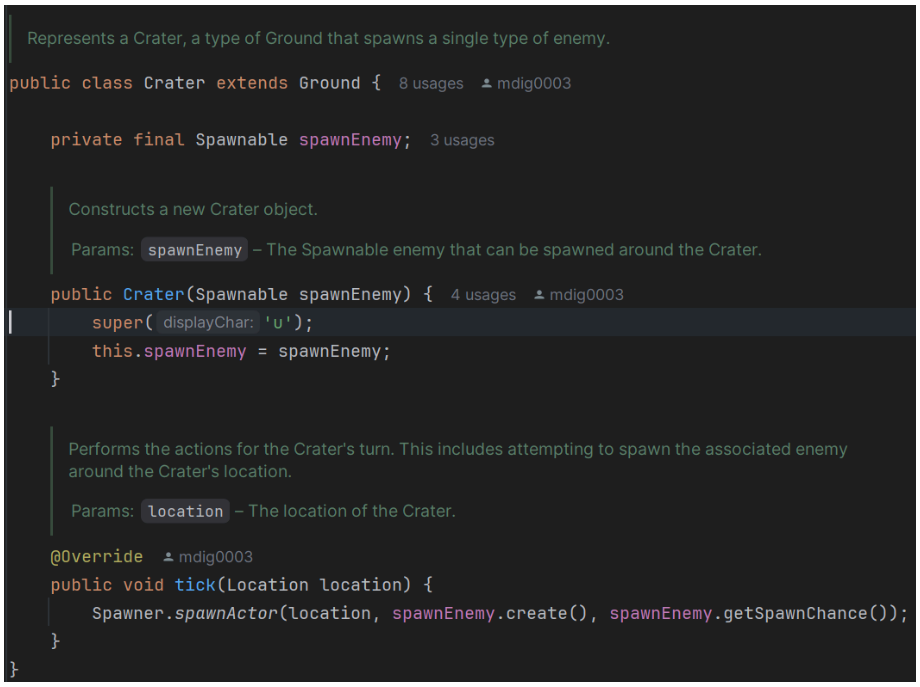
Current Design

The current approach builds upon the design used in Assignment 1 to handle the spawning of different types of enemies in the game.



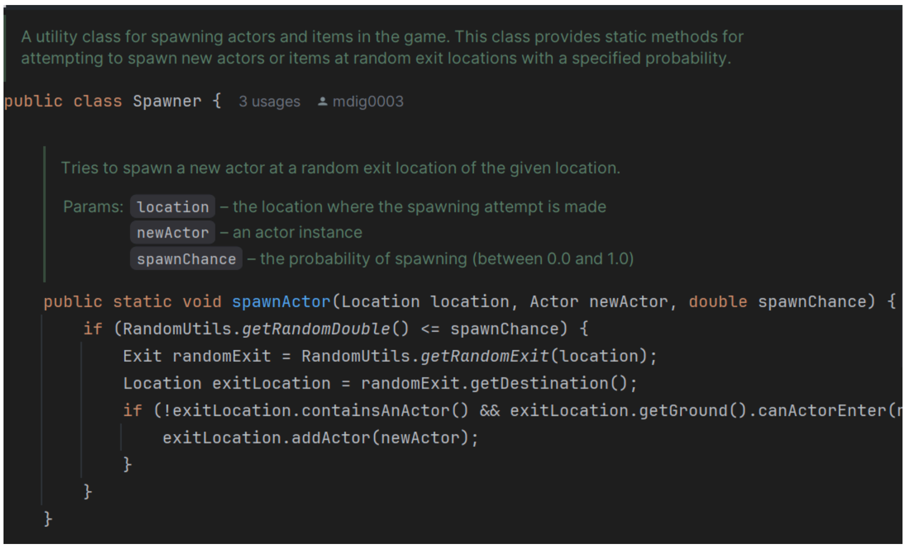
```
An interface representing an entity that can be spawned in the game. Implementing classes must provide a
method to create an instance of the entity and a method to retrieve the chance of the entity being
spawned.
public interface Spawnable {   8 usages   3 implementations   mdig0003
    Creates a new instance of the entity.
    Returns: A new instance of the entity.
    Actor create();   3 implementations   mdig0003

    Returns the chance of the entity being spawned.
    Returns: The spawn chance of the entity from 0 to 1.
    double getSpawnChance();   1 usage   3 implementations   mdig0003
}
```

Each spawnable enemy, such as HuntsmanSpider or AlienBug, implements the Spawnable interface, which encapsulates the ability of an actor to create new instances of itself and provide its spawn chance.



```
Represents a Crater, a type of Ground that spawns a single type of enemy.
public class Crater extends Ground {  8 usages  mdig0003

    private final Spawnable spawnEnemy;  3 usages

    Constructs a new Crater object.
    Params: spawnEnemy – The Spawnable enemy that can be spawned around the Crater.
    public Crater(Spawnable spawnEnemy) {  4 usages  mdig0003
        super( displayChar: 'u');
        this.spawnEnemy = spawnEnemy;
    }

    Performs the actions for the Crater's turn. This includes attempting to spawn the associated enemy
    around the Crater's location.
    Params: location – The location of the Crater.
    @Override  mdig0003
    public void tick(Location location) {
        Spawner.spawnActor(location, spawnEnemy.create(), spawnEnemy.getSpawnChance());
    }
}
```

The Crater class acts as a spawning ground for these enemies. It depends on the Spawnable interface, adhering to the Dependency Inversion Principle (DIP) and promoting loose coupling between the crater and the specific enemy types.



```
A utility class for spawning actors and items in the game. This class provides static methods for
attempting to spawn new actors or items at random exit locations with a specified probability.
public class Spawner {  3 usages  mdig0003

    Tries to spawn a new actor at a random exit location of the given location.
    Params: location – the location where the spawning attempt is made
            newActor – an actor instance
            spawnChance – the probability of spawning (between 0.0 and 1.0)
    public static void spawnActor(Location location, Actor newActor, double spawnChance) {
        if (RandomUtils.getRandomDouble() <= spawnChance) {
            Exit randomExit = RandomUtils.getRandomExit(location);
            Location exitLocation = randomExit.getDestination();
            if (!exitLocation.containsAnActor() && exitLocation.getGround().canActorEnter(n
                exitLocation.addActor(newActor);
            }
        }
    }
```

The Crater class delegates the spawning logic to the Spawner utility class, which provides a centralised method spawnActor for spawning actors based on the provided location, actor instance, and spawn chance.

This approach enables the system to easily incorporate new spawnable enemies by creating new classes that implement the Spawnable interface, without modifying the existing code (OCP). It also ensures a separation of concerns, where the Crater class focuses on representing the crater itself, while the spawning logic is encapsulated within the Spawner utility class and the Spawnable interface implementations (SRP).

Pros:

- The current design is simpler and requires fewer classes compared to the alternate design which involves creating multiple classes to achieve the same functionality.
- New spawnable actors can be added just by implementing the Spawnable interface without modifying existing code (OCP).
- The Crater class depends on the Spawnable interface instead of concrete classes, promoting loose coupling (DIP).
- The Spawner utility class can be reused in other parts of the application where spawning actors is required, reducing code duplication (DRY).

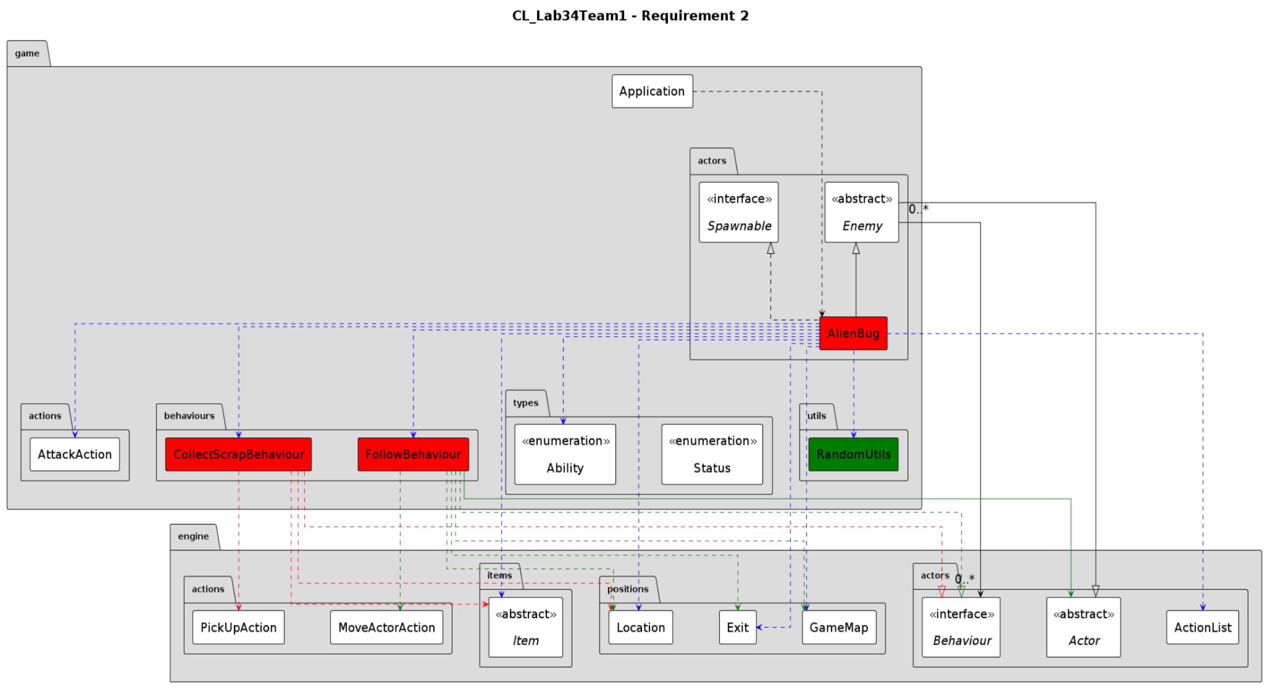
Cons:

- The introduction of the Spawnable interface and related classes adds some complexity to the codebase, potentially violating the KISS (Keep It Simple, Stupid) principle to some extent.
- The spawning logic is encapsulated within the Spawner utility class and the Spawnable interface instead of just a single class, which isn’t ideal.

REQ2: The imposter among us

In REQ2, two new hostile creatures, AlienBug and Suspicious Astronaut are introduced on the moon. The Alien Bug can follow the Intern around, steal valuable scraps from the Intern's spaceship, and must be defeated by the Intern to retrieve the stolen scraps. The Suspicious Astronaut will wander if the Intern is not nearby but instantly kill the Intern if within one exit away, regardless of the Intern's health.

<u>UML</u>



The diagram above represents an object-oriented system for REQ2.

<u>Key Classes Introduced</u>

- The Enemy abstract class defines the common behaviour and properties for all types of enemies in the game, representing hostile actors that can be attacked by other actors with the HOSTILE_TO_ENEMY capability. The Enemy class maintains a map of Behaviour instances that it will execute based on priority.

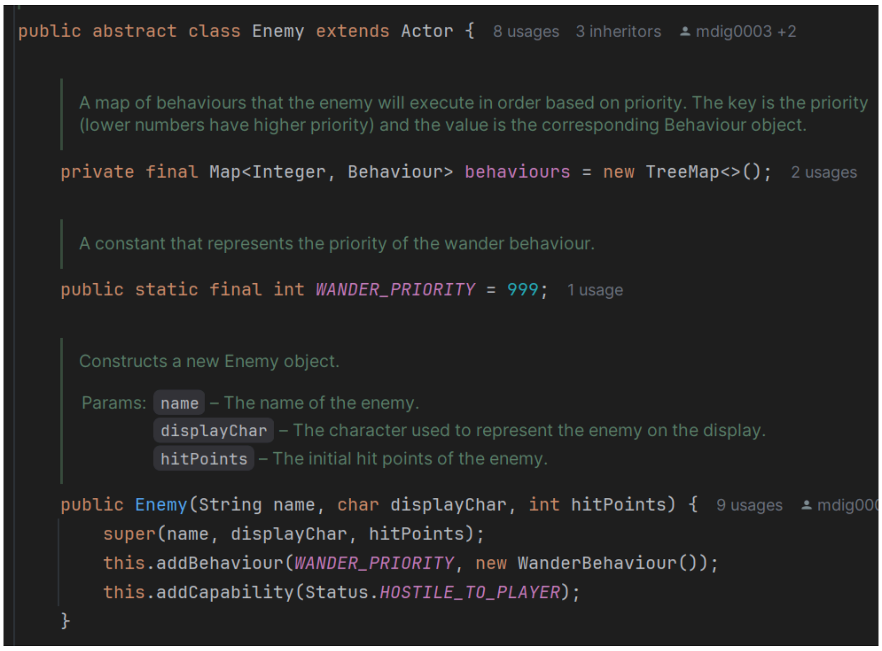
- The AlienBug class is a concrete implementation that extends the Enemy abstract class, representing an AlienBug.

- The CollectScrapBehaviour implements the Behaviour interface from the game engine and it allows an actor (AlienBug) to collect scraps from the ground. It will randomly select an item at the current location and return a PickUpAction for it.

- The FollowBehaviour also implements the Behaviour interface from the game engine and it figures out a MoveAction that will move the actor one step closer to a target Actor.

- The SuspiciousAstronaut class is a concrete implementation that extends the Enemy abstract class, representing a SuspiciousAstronaut.

Notable Changes from Assignment 1



```
public abstract class Enemy extends Actor {   8 usages   3 inheritors   mdig0003 +2

    A map of behaviours that the enemy will execute in order based on priority. The key is the priority
    (lower numbers have higher priority) and the value is the corresponding Behaviour object.
    private final Map<Integer, Behaviour> behaviours = new TreeMap<>();   2 usages

    A constant that represents the priority of the wander behaviour.
    public static final int WANDER_PRIORITY = 999;   1 usage

    Constructs a new Enemy object.
    Params: name – The name of the enemy.
            displayChar – The character used to represent the enemy on the display.
            hitPoints – The initial hit points of the enemy.
    public Enemy(String name, char displayChar, int hitPoints) {   9 usages   mdig000
        super(name, displayChar, hitPoints);
        this.addBehaviour(WANDER_PRIORITY, new WanderBehaviour());
        this.addCapability(Status.HOSTILE_TO_PLAYER);
    }
```

The behaviours for the Enemy class are now stored in a TreeMap instead of ArrayList, with the keys representing behaviour priorities and the value containing the specific Behaviour. This is because TreeMap allows efficient retrieval of behaviours based on their priorities.

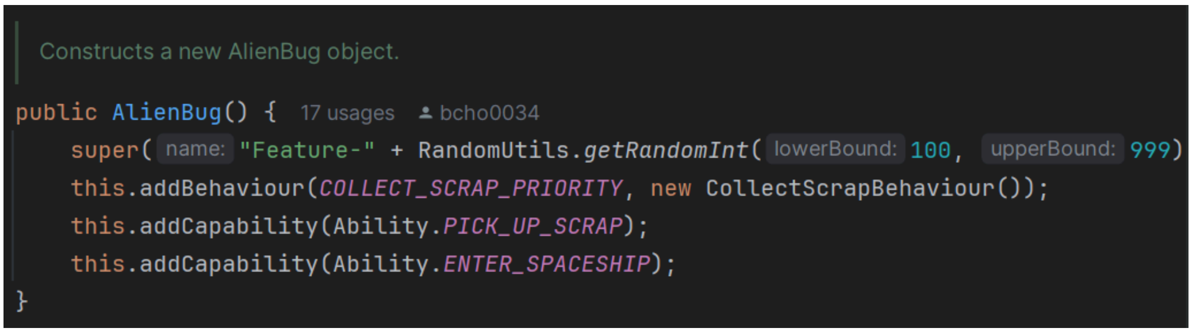
Furthermore, the priority value for the behaviours now also uses named constants instead of hard-coded integers to improve the connascence of meaning. The introduction of the constants making the code more self-documenting and easier to understand.

Current Design

Both AlienBug and SuspiciousAstronaut extends the abstract Enemy class which defines common properties and behaviours of all enemies in the game thus reducing code repetition (DRY). This will continue to make it easy if more enemies are to be created in future.

The AlienBug is given the ability to pick up scraps and enter the Intern's spaceship using the Ability enumeration. The specific behaviours of the Alien Bug, such as collecting scraps and following the Intern, are encapsulated in separate classes that implement the Behaviour interface.

We have implemented the Spawnable interface, which is needed for the future requirements when the crater can only spawn one type of enemy. This also follows the interface segregation principle as all methods from the interface are being implemented and the single responsibility principle. This also forces the person implementing it to add the chance for the enemy to be spawned. For the damage that the suspicious astronaut does on the player, I have had to pass in the player when creating a new instance of this class, to get their max health and make sure that the attack performed is on the max health in case it ever changes.



```
Constructs a new AlienBug object.

public AlienBug() {  17 usages  bcho0034
    super( name: "Feature-" + RandomUtils.getRandomInt( lowerBound: 100,  upperBound: 999)
    this.addBehaviour(COLLECT_SCRAP_PRIORITY, new CollectScrapBehaviour());
    this.addCapability(Ability.PICK_UP_SCRAP);
    this.addCapability(Ability.ENTER_SPACESHIP);
}
```

This promotes code reusability, as the same behaviour can be used by other actors that need to collect scraps from the ground by adding it (DRY). Additionally, this design also adheres to SRP, as the responsibility of different behaviour are separated into different classes. For example CollectScrapBehaviour class is solely responsible for the scrap collection behaviour. The AlienBug class can be substituted for the Enemy abstract class without violating correctness, as it adheres to the contract defined by the base class (LSP).

Pros:

- The CollectScrapBehaviour and FollowBehaviour classes can be reused by other actors that require similar behaviours, promoting code reuse and reducing duplication (DRY).

- The responsibilities of the AlienBug class and its behaviours are separated, adhering to the Single Responsibility Principle (SRP) and promoting better code organisation and maintainability.

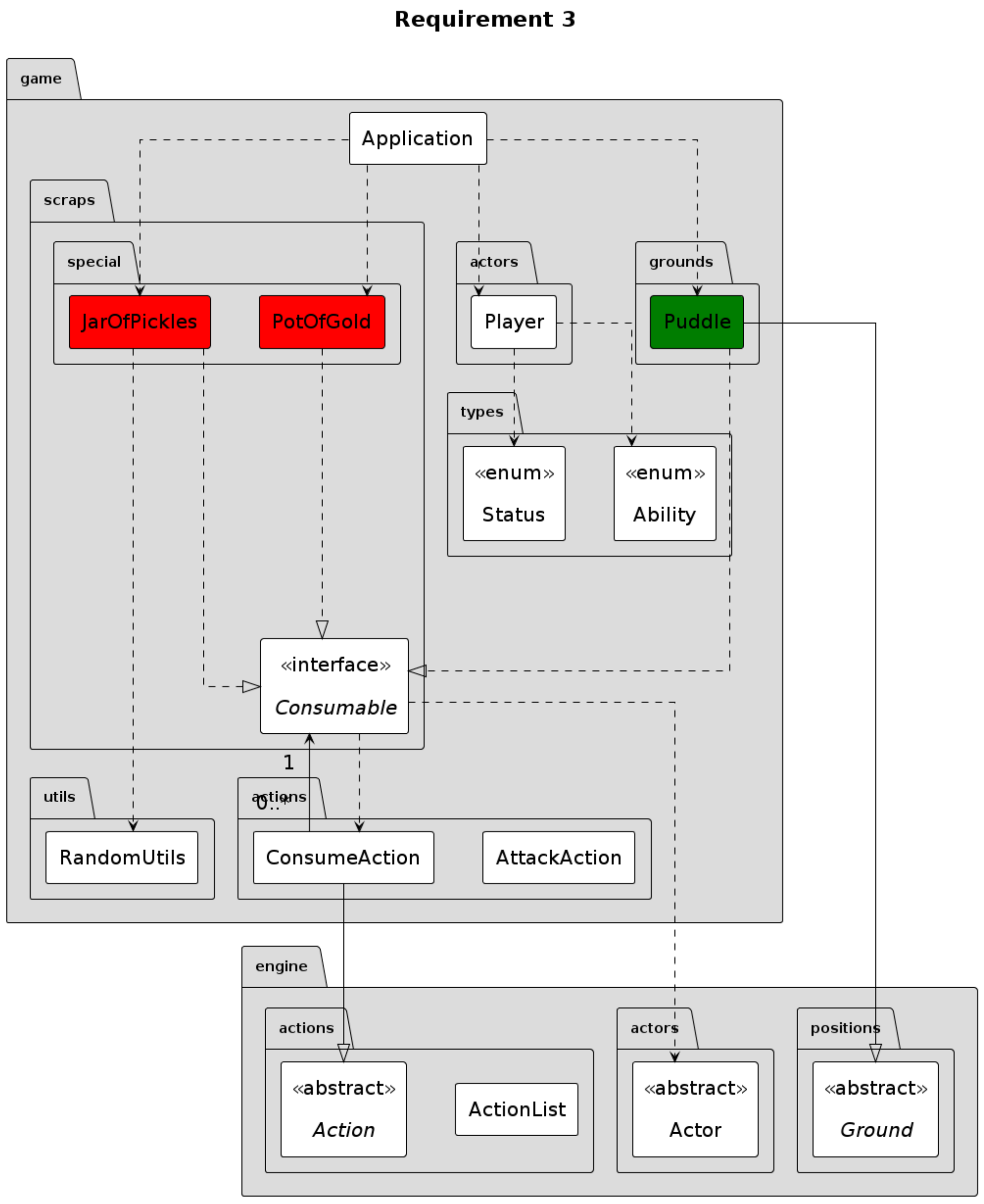
Cons:

- The Suspicious Astronaut requires the player to be passed in to be able to get the max HP of that player. An alternative could have been to create a static class to get that max HP.

REQ3: More scraps

In Requirement three, two new special items, ***PotOfGold*** and ***JarOfPickles***, are introduced on the Moon. Players can consume the ***PotOfGold*** to gain an increase of ten credits to their balance. Consumption of the ***JarOfPickles*** will either increase or decrease the player's HP by 1, depending on the game mechanics. Additionally, this requirement introduces a feature where the intern character can consume a puddle of water if standing on it, which will result in an increase of their maximum HP by 1.

UML



The diagram above represents an object-oriented system for REQ3.

Key Introduced Classes

Here's a brief description of the highlighted classes and their roles in the UML diagram:

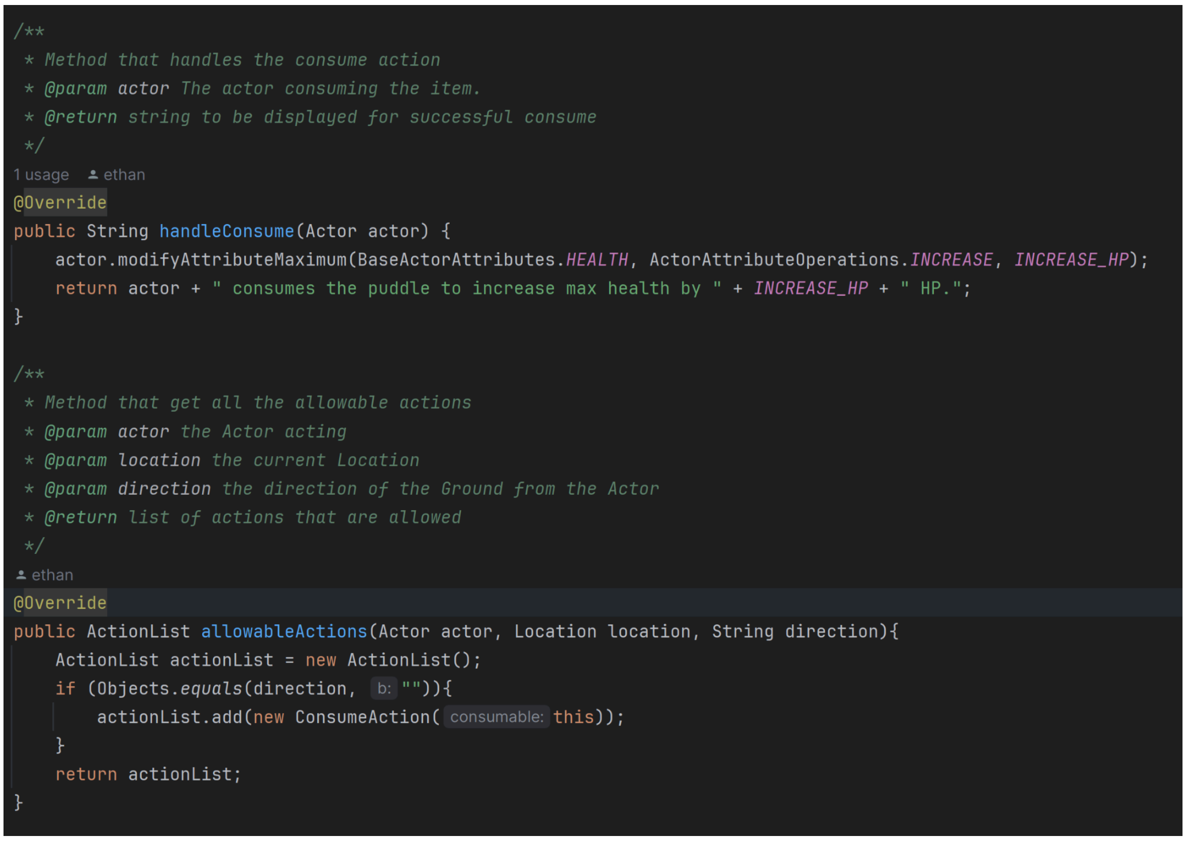
- The ***PotOfGold*** class represents a consumable item in the game that, when used by a player, increases their balance by ten credits. This class extends the basic ***Item*** structure and adheres to the Consumable interface, allowing for integration into the game's item usage system.

- The ***JarOfPickles*** class encapsulates a unique consumable item which, upon consumption, can either increase or decrease the player's HP by 1. As with ***PotOfGold***, it extends from the ***Item*** class and implements the ***Consumable*** interface, fitting seamlessly into the game's broader system of items and effects.

- The ***Puddle*** class is designed as a ground type on the game map that offers a health increase in player HP. Specifically, when an intern character interacts with the puddle by standing on it, they receive an increase in maximum HP by 1.

Notable Changes from Assignment 1

In the game, the Puddle class is categorised as a ground type element on the map that provides a health benefit to players. Specifically, when an intern character steps onto a puddle, their maximum HP increases by 1.



```java
/**
 * Method that handles the consume action
 * @param actor The actor consuming the item.
 * @return string to be displayed for successful consume
 */
1 usage   ethan
@Override
public String handleConsume(Actor actor) {
    actor.modifyAttributeMaximum(BaseActorAttributes.HEALTH, ActorAttributeOperations.INCREASE, INCREASE_HP);
    return actor + " consumes the puddle to increase max health by " + INCREASE_HP + " HP.";
}

/**
 * Method that get all the allowable actions
 * @param actor the Actor acting
 * @param location the current Location
 * @param direction the direction of the Ground from the Actor
 * @return list of actions that are allowed
 */
ethan
@Override
public ActionList allowableActions(Actor actor, Location location, String direction){
    ActionList actionList = new ActionList();
    if (Objects.equals(direction, b: "")){
        actionList.add(new ConsumeAction( consumable: this));
    }
    return actionList;
}
```

Furthermore, the group chose to remove the ***Scrap*** abstraction class for special scraps, considering it was redundant. Instead, we adopted the ***Item*** abstract class, which improved code organisation and adhered more effectively to the DRY (Don't Repeat Yourself) principle.

<u>Alternative Design Considered</u>

An alternative design involves the introduction of an "***AddBalanceAction***" class in the actions. This class would encapsulate the action of adding balance to a player's wallet as an allowable action. This abstraction would not only apply to the ***PotOfGold*** item but also extend to any future special scrap items designed to adjust a player's balance. The ***AddBalanceAction*** would be integrated into the game by including it in the ***allowableActions*** list for applicable items.

Pros:

- Adherence to Single Responsibility Principle (SRP), by delegating the balance addition functionality to a separate action class, the ***PotOfGold*** and similar items are not burdened with multiple responsibilities. This separation enhances the modularity and maintainability of the code.

- Extensibility (Open/Closed Principle - OCP), new types of items that involve balance changes can easily integrate the ***AddBalanceAction*** without modifying the existing code. This makes the system more flexible and scalable, accommodating future expansions with minimal changes.

Cons:

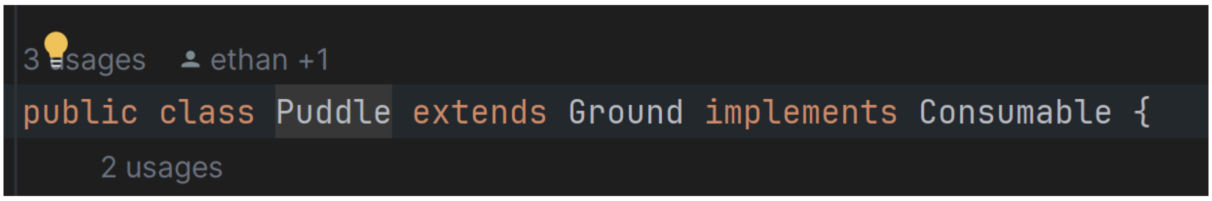
- Increased Complexity, introducing a new class for handling balance additions adds an additional layer of abstraction to the system. This may complicate the architecture, requiring developers to understand more about the interaction between different classes.

- Performance Considerations, the additional abstraction might introduce minor performance overhead due to more complex method dispatches and the creation of additional objects in scenarios where many items are interacting with the player's balance.

- This alternative design strategy aligns with principles of good object-oriented design by enhancing the system's organisation and flexibility while carefully considering the trade-offs in complexity and performance.

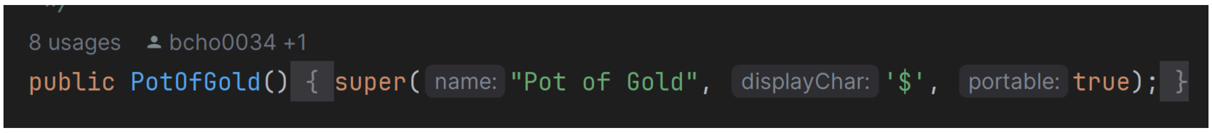
<u>Current Design</u>

We have chosen to leave puddles as an extension of ground as it previously already was. However I have chosen to implement the interface consumable which will handle the consumable and change the HP of the actor.



```
3 usages   ethan +1
public class Puddle extends Ground implements Consumable {
    2 usages
```

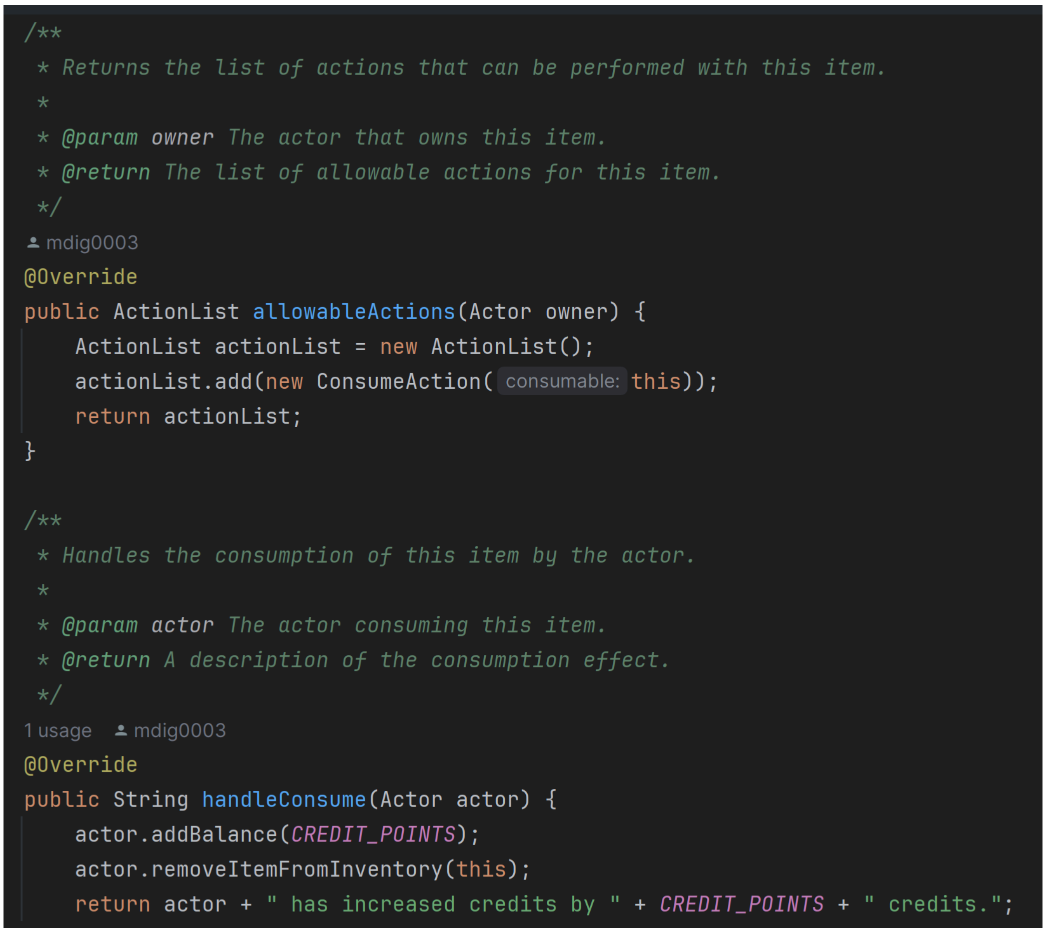
Both the PotOfGold and JarOfPickles classes extend the item class as well as utilise the consumable interface. This is similarly implemented to other special scraps such as largeFruit and smallFruit.



```
8 usages   bcho0034 +1
public PotOfGold() { super( name: "Pot of Gold",  displayChar: '$',  portable: true); }
```

These classes continue to follow the interface segregation principles as well as the single responsibility principle.

When Implementing our design, we tried to implement the KISS and DRY principles. This is by utilising other interfaces and abstraction classes to handle all of the main functionality without us repeating the code.



```
/**
 * Returns the list of actions that can be performed with this item.
 *
 * @param owner The actor that owns this item.
 * @return The list of allowable actions for this item.
 */
mdig0003
@Override
public ActionList allowableActions(Actor owner) {
    ActionList actionList = new ActionList();
    actionList.add(new ConsumeAction( consumable: this));
    return actionList;
}

/**
 * Handles the consumption of this item by the actor.
 *
 * @param actor The actor consuming this item.
 * @return A description of the consumption effect.
 */
1 usage   mdig0003
@Override
public String handleConsume(Actor actor) {
    actor.addBalance(CREDIT_POINTS);
    actor.removeItemFromInventory(this);
    return actor + " has increased credits by " + CREDIT_POINTS + " credits.";
```

Pros:

- Adhering to the Interface Segregation Principle enhances the maintainability and comprehensibility of the code, ensuring that changes in one part of the system minimally impact others.

- The design of all classes for this requirement strictly follows the SRP (Single Responsibility Principle), fostering improved code organisation and facilitating easier management and scalability.

Cons:

- The ***CREDIT_POINTS*** is a static final value. If the game requires different pots of gold to have different values, this implementation does not support such changes. Using a constructor parameter or a setter method to define the credit value could provide more flexibility.

- The ***handleConsume*** method handles both the modification of the actor's state (increasing balance) and the item's state (removal from inventory). Ideally, these concerns should be separated to adhere to SRP, possibly by handling the removal of the item in another class or method that specifically deals with inventory management.

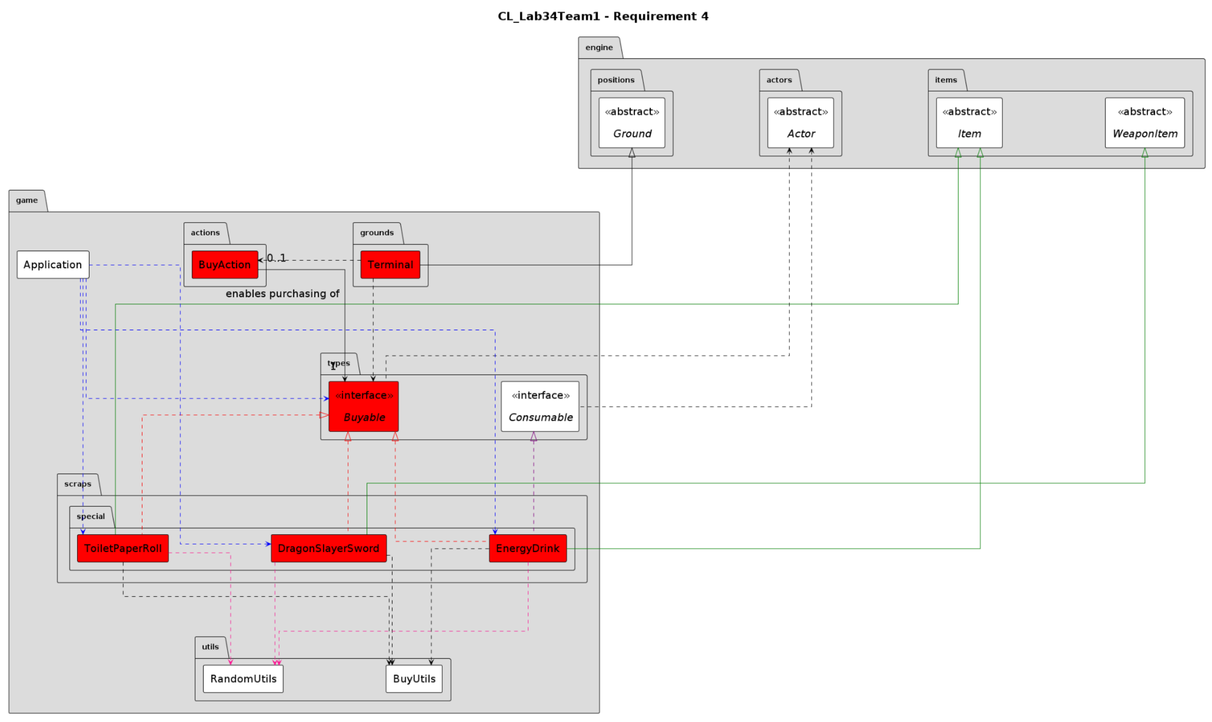
REQ4: Static factory's staff benefits

In Static Factory, the intern can access a Computer Terminal inside the spaceship on the abandoned moon. This Computer Terminal acts as a store, allowing the intern to purchase various items using credits. The items available for purchase include an Energy Drink, a replica of the legendary Dragon Slayer Sword, and a Toilet Paper Roll.

The Energy Drink serves as a healing item, replenishing the intern's health when consumed. The Dragon Slayer Sword replica is a powerful weapon that the intern can wield to attack hostile creatures encountered on the moon. However, there's a chance that the Computer Terminal may malfunction and take the intern's credits without delivering the weapon. The Toilet Paper Roll, on the other hand, serves no practical purpose.

The Computer Terminal may offer discounts or charge extra fees for certain items, adding an element of unpredictability to the purchasing experience. All purchased items can be picked up, carried, and dropped off by the intern, similar to other items in the game.

UML



Key Classes Introduced

The UML class diagram above illustrates the class structure and relationships for implementing the buying functionality and special items in the game. Specifically,

1. BuyUtils
   - This class contains utility methods related to the buying process, such as calculating prices, applying discounts, or handling errors.
   - It is used by the `DragonSlayerSword`, `EnergyDrink`, and `ToiletPaperRoll`, providing common functionality for purchasing these items.

2. DragonSlayerSword:
   - It implements the `Buyable` interface, meaning it can be purchased from the Terminal.
   - It uses the `RandomUtils` and `BuyUtils` classes for implementing the buying logic and handling probability-based events, such as the Terminal malfunctioning.

3. EnergyDrink:
   - It implements the `Buyable` interface, allowing it to be purchased from the Terminal.
   - It also implements the `Consumable` interface, as the Energy Drink can be consumed by the player to provide a healing effect.
   - Like the `DragonSlayerSword`, it uses the `RandomUtils` and `BuyUtils` classes for implementing buying logic and handling probability-based events, such as the Terminal offering a discount.

4. ToiletPaperRoll:
   - It implements the `Buyable` interface, allowing it to be purchased from the Terminal.
   - It uses the `RandomUtils` and `BuyUtils` classes for implementing buying logic and handling probability-based events, such as the Terminal offering a discount for this item.

5. Terminal:
   - Extends the `Ground` class from the `engine.positions` package, as it is a physical item that must be placed on a ground position in the game world.
   - It has a relationship with the `Buyable` interface as it can sell items that implement the `Buyable` interface, such as the `DragonSlayerSword`, `EnergyDrink`, and `ToiletPaperRoll`.

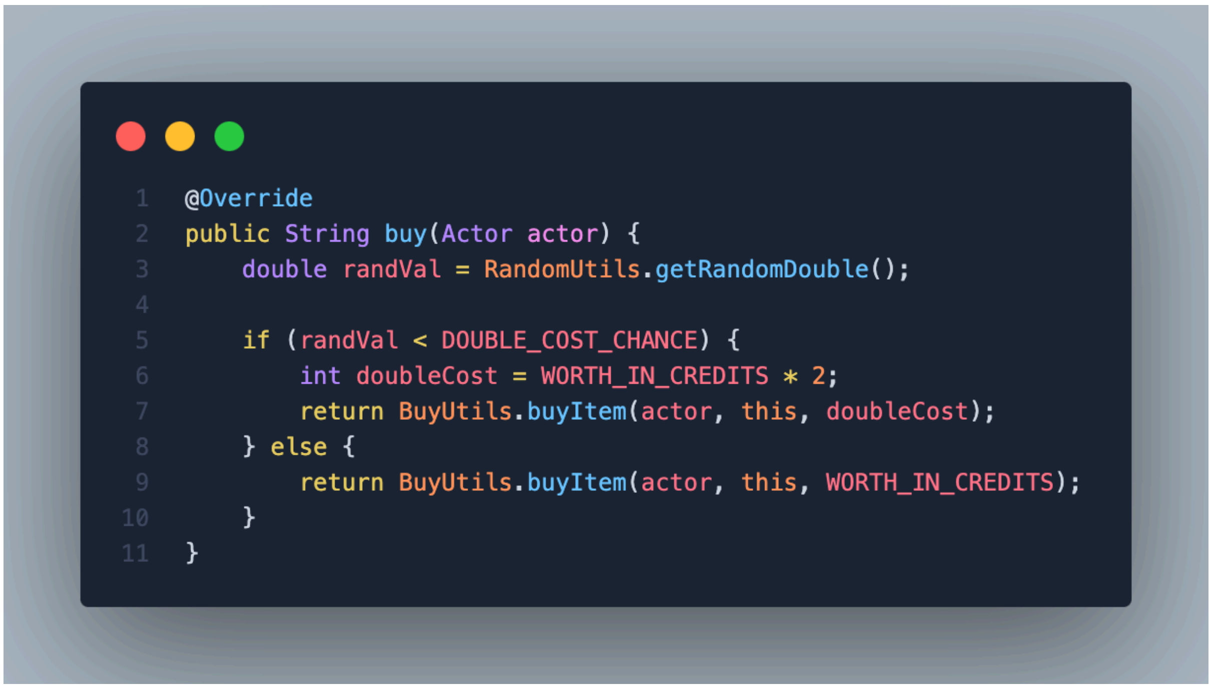
6. Buyable:
   - This interface defines the contract for items that can be purchased from the Computer Terminal.

7. BuyAction:
   - This class represents the action of buying an item (in this case from the Terminal).
   - It has an association relationship with the `Buyable` interface as the BuyAction requires a Buyable item parameter.

Current design

Pros

Single Responsibility Principle (SRP)
The design separates concerns by having dedicated classes for different responsibilities. For example, BuyUtils handles buying-related operations, and individual item classes like DragonSlayerSword, EnergyDrink, and ToiletPaperRoll manage their respective item and purchasing behaviours. An extract of the EnergyDrink buy method is given below to show how it manages its purchasing behaviour.



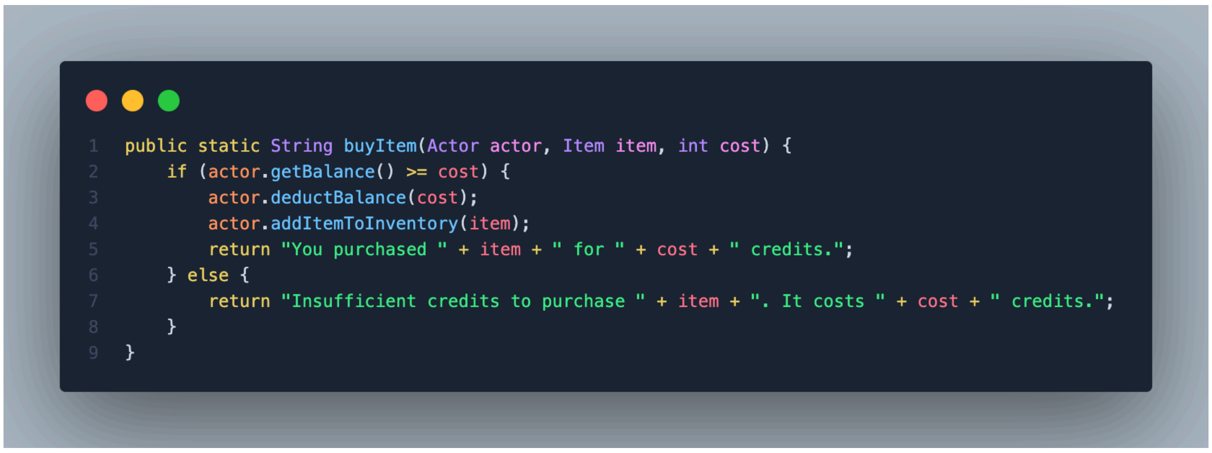
```java
@Override
public String buy(Actor actor) {
    double randVal = RandomUtils.getRandomDouble();

    if (randVal < DOUBLE_COST_CHANCE) {
        int doubleCost = WORTH_IN_CREDITS * 2;
        return BuyUtils.buyItem(actor, this, doubleCost);
    } else {
        return BuyUtils.buyItem(actor, this, WORTH_IN_CREDITS);
    }
}
```

Open/Closed Principle (OCP)
The use of the Buyable interface allows for future extensions without modifying existing code. New items can be added by implementing these interfaces, promoting code extensibility.

Liskov Substitution Principle (LSP)
The inheritance hierarchies follow the LSP principle. For instance, DragonSlayerSword inherits from WeaponItem, ensuring that instances of DragonSlayerSword can be substituted wherever WeaponItem instances are expected.

DRY Principle

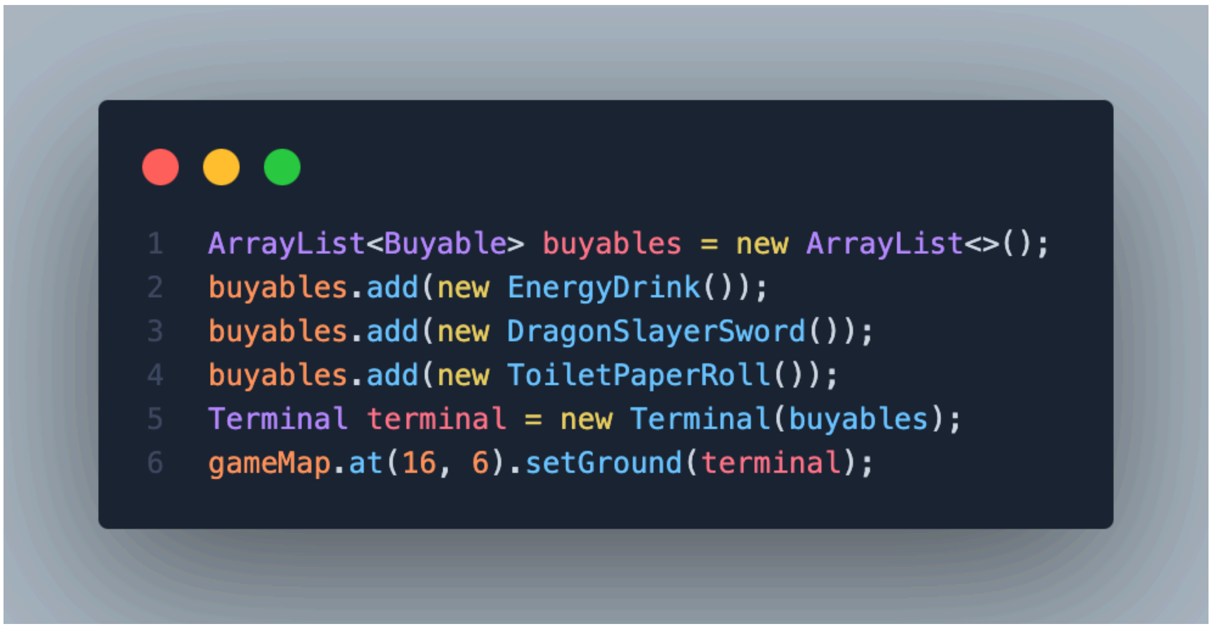
The use of utility classes like BuyUtils helps avoid duplication of code related to buying across different item classes. An extract of a BuyUtils method is given below.



```java
public static String buyItem(Actor actor, Item item, int cost) {
    if (actor.getBalance() >= cost) {
        actor.deductBalance(cost);
        actor.addItemToInventory(item);
        return "You purchased " + item + " for " + cost + " credits.";
    } else {
        return "Insufficient credits to purchase " + item + ". It costs " + cost + " credits.";
    }
}
```

Attribute Factory Pattern

The design utilises the Attribute Factory Pattern, where the Application class acts as the factory, creating instances of DragonSlayerSword, EnergyDrink, and ToiletPaperRoll. This pattern allows for the creation of objects with different attributes or behaviours without exposing the instantiation logic to the client code. It also means Terminal has no knowledge of what it should sell, but that it will just work so long as an arraylist of Buyable items are its input.



```java
ArrayList<Buyable> buyables = new ArrayList<>();
buyables.add(new EnergyDrink());
buyables.add(new DragonSlayerSword());
buyables.add(new ToiletPaperRoll());
Terminal terminal = new Terminal(buyables);
gameMap.at(16, 6).setGround(terminal);
```

Cons

Potential Violation of Interface Segregation Principle (ISP)
The Buyable interface may be too broad, as it’s implemented by items that have different buying behaviours (e.g., probability-based events, discounts). It might be better to segregate these behaviours into separate interfaces or use composition instead of inheritance.

KISS Principle
While the design aims for simplicity, the introduction of multiple interfaces and utility classes could potentially make the codebase more complex than necessary for a relatively straightforward buying mechanic.

Potential Leaky Abstractions
The use of probability-based events and discounts within the item classes themselves could be considered a leaky abstraction, as these concerns might be better handled by separate classes or components. It might be better to come up with a more generalised utility that accepts parameters to determine the buying behaviour.

Alternative Design Considered

An alternative design might separate the buying functionality into its own package. The core buying logic would be encapsulated in a BuyingService class, which would handle the actual purchase of Buyable items.

The Terminal, Buyable interface, and classes that implement that interface would remain the same as the current design, but to handle specific buying behaviours, such as discounts or errors, the design could employ the Decorator pattern [2]. The RandomDiscountDecorator and ErrorDecorator classes would implement the BuyingService interface and decorate the core buying logic with their respective behaviours, separating this logic away from the items themselves.

The RandomDiscountDecorator would apply a random discount to the purchase, while the ErrorDecorator simulates the probability-based error event where the Terminal takes the player's credits without delivering the item.

Pros:

- Single Responsibility Principle (SRP): The buying functionality is separated into its own package, with the BuyingService class responsible for handling the core buying logic. The decorators (RandomDiscountDecorator and ErrorDecorator) have a single responsibility of modifying the buying behaviour.

- Open/Closed Principle (OCP): By using the Decorator pattern, new buying behaviours can be added by creating new decorator classes without modifying the existing BuyingService or Buyable classes.

- Interface Segregation Principle (ISP): The Buyable interface is focused solely on the buying behaviour, avoiding potential violations of the ISP.

- Dependency Inversion Principle (DIP): The BuyingService and its decorators depend on abstractions (Buyable interface) rather than concrete implementations, making the code more modular and extensible.

- KISS Principle: The core buying logic is kept simple and separated from the specific buying behaviours, making the code easier to understand and maintain.

- DRY Principle: The decorators encapsulate the specific buying behaviours, avoiding duplication of code across different item classes.

Cons:

- Increased Complexity: The introduction of the Decorator pattern and the separation of concerns into multiple classes and interfaces may increase the overall complexity of the codebase, especially for a relatively simple buying mechanic.

- Potential Over Engineering:This design would be considered over engineered for a simple buying system, adding unnecessary abstractions and indirection. This is the main reason why we chose not to implement it, though in future we could pivot to this system if there were many more items to sell that also had unique purchasing behaviours.

[3] This alternative design partially included the use of generative AI for idea generation

References

[1] Lachlan MacPhee, Assignment 1, FIT2099 2024

[2] The Decorator Pattern in Java, https://www.baeldung.com/java-decorator-pattern

[3] Anthropic Claude, Generative AI, https://claude.ai