FIT2099 Assignment 3 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 A3 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 to the game in future 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 with a red background are new
- Classes with a green background have been modified from their previous implementation.
- Classes with a white background are unchanged from the last assignment

Notable Changes from Assignment 2

In Assignment 3, several notable changes were introduced to improve code quality, maintainability and extensibility. Firstly, the SuspiciousAstronaut class was refactored to remove its dependency on the Player class. Instead, the maximum integer value was used as damage to ensure that SuspiciousAstronaut can instantly kill the player. This eliminated the need to access the player's maximum health value, promoting loose coupling.

The enemy spawning logic was also restructured by introducing the Spawner interface and concrete implementations like HuntsmanSpiderSpawner. This change follows the Open/Closed Principle, allowing for easy addition of new enemy spawners without modifying existing code.

Additionally, the use of int for spawn chance calculations was standardised across the codebase, promoting consistency.

REQ1: The Ship of Theseus

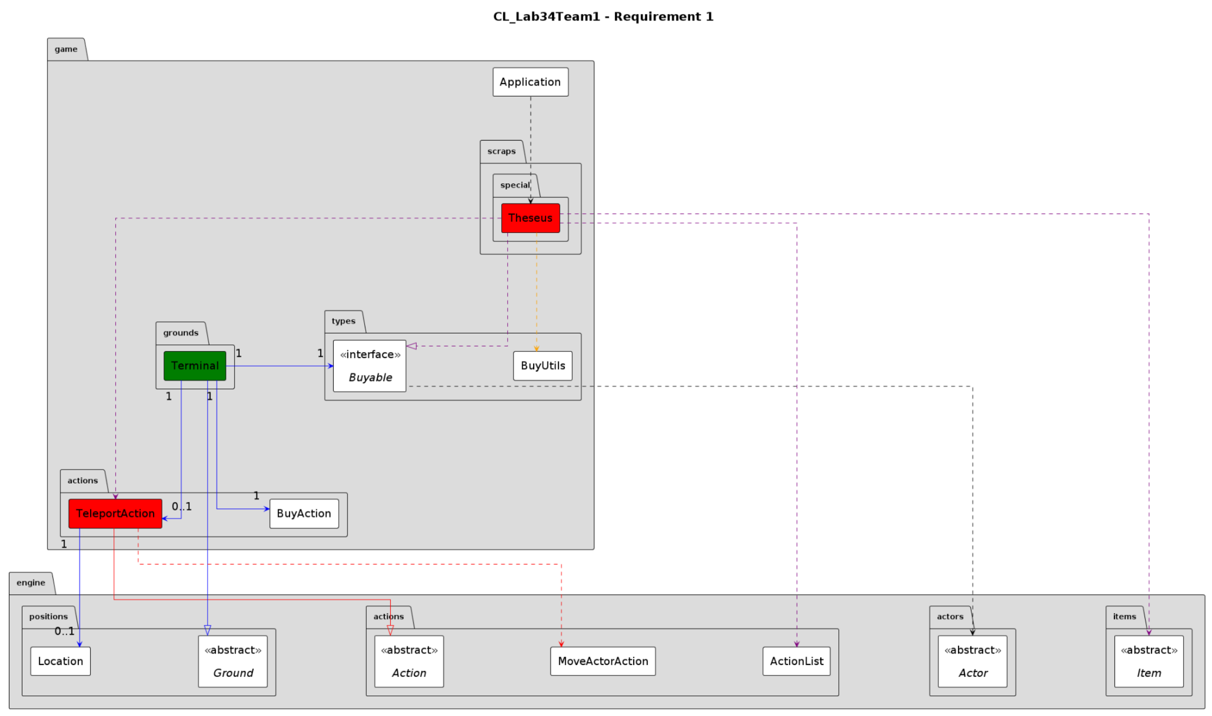
The intern can now interact with computer terminals (`=`) to travel between moons or to the factory's spaceship parking lot. If the intern is near any computer terminal, they can access travel options to locations they are not currently in. For example, if they are in the factory, the terminal will only display options to travel to Polymorphia or the new moon.

The computer terminal now also sells THESEUS (`^`), a portable teleporter that allows the intern to instantly move to a random location within a single map for free. THESEUS costs 100 credits and is available from the start of the game, along with the items from Assignment 2 REQ4.

To use THESEUS, the intern must put it down on the ground and select the option to teleport (i.e., "Teleport with THESEUS"). THESEUS will then randomly teleport the intern to a new location within the current map. However, the factory is not responsible if the intern is teleported on top of a tree or a wall.

A new moon is now accessible through the computer terminal, allowing the intern to travel there and collect more scraps.

UML



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

Key Classes Introduced

1. `Theseus`: Represents a special item in the game that can be bought by actors. It implements the `Buyable` interface and provides functionality for teleporting to a random location on the current map when used.

2. `Terminal`: Represents a ground object in the game where actors can purchase `Buyable` items. It maintains a list of available `Buyable` items and provides an `allowableActions` method that returns a list of `BuyAction`s for each item. It also allows hostile actors to travel to different game maps.

Alternative Design Considered

Description

Terminal
An alternative design for adding teleportActions in the Terminal could involve using a HashMap, where the key represents the gameMap and the value represents the teleportAction required to travel to a different gameMap. When adding the possible travelActions to the Terminal, this approach would allow for all the valid travelActions based on the player's current gameMap, as we would iterate through keys to check if there is a player present in a specific gameMap.

Pros:
- This design would reduce the number of lines of code needed to implement the teleportation functionality.
- It adheres to the SOLID principles, promoting a more modular and maintainable codebase.
- Would adhere to our goals of extensibility and allow for a far simpler scalable approach.

Cons:
- Introducing a HashMap for this purpose might add unnecessary complexity to the overall design.
- If the game were to be extended to support multiple players, this approach could potentially lead to additional issues or complexities in managing the teleportActions for each player.

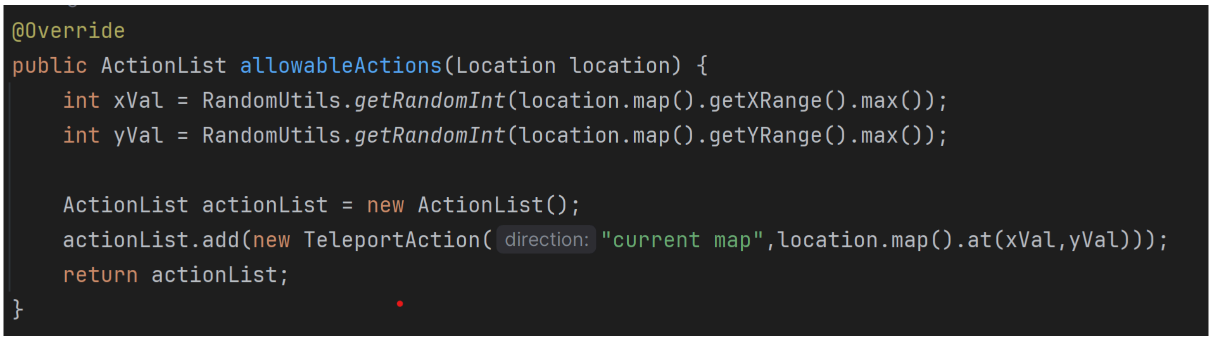
Ultimately, the choice between using a HashMap or a simpler approach should be based on the specific requirements and future scalability needs of the game. If the current implementation is straightforward and meets the requirements without adding excessive complexity, it may be preferable to stick with the simpler approach. However, if the game is expected to grow in complexity and require more flexibility in managing teleportActions, using a HashMap could provide a more extensible solution.

Final Design

Description

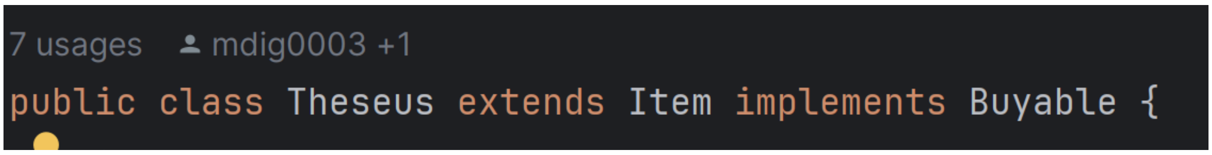
The final design chosen is to make `Theseus` a subclass of `Item` and implement the `Buyable` interface. This design adheres to the SOLID principles and promotes a clear separation of concerns.

- `Theseus` is solely responsible for representing a buyable item with TeleportAction functionality.



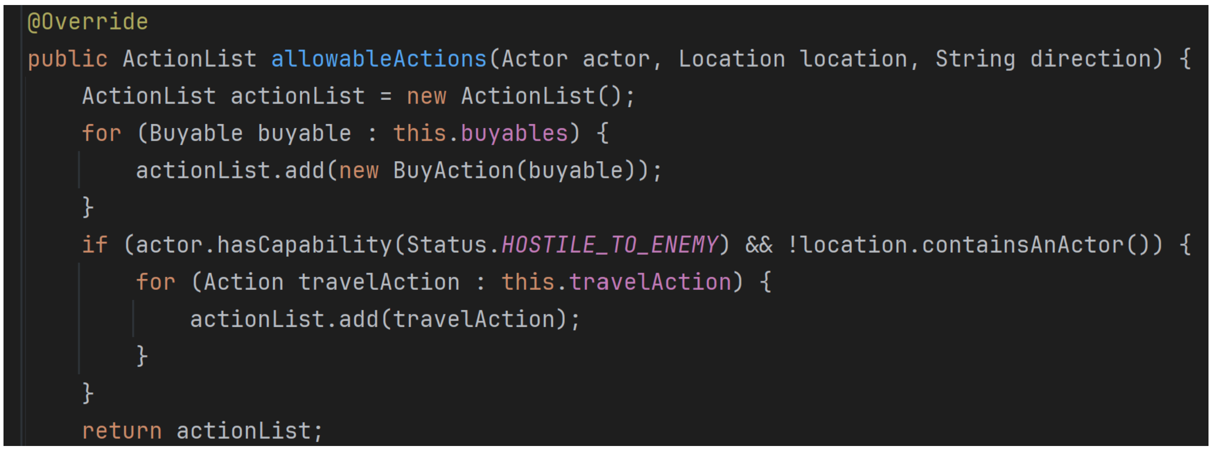
```
@Override
public ActionList allowableActions(Location location) {
    int xVal = RandomUtils.getRandomInt(location.map().getXRange().max());
    int yVal = RandomUtils.getRandomInt(location.map().getYRange().max());

    ActionList actionList = new ActionList();
    actionList.add(new TeleportAction( direction: "current map",location.map().at(xVal,yVal)));
    return actionList;
}
```

- The `Buyable` interface defines a contract for items that can be purchased by actors, promoting interface segregation.



```
7 usages  mdig0003 +1
public class Theseus extends Item implements Buyable {
```

- The `Terminal` class is responsible for managing the available `Buyable` items and providing the necessary actions for purchasing them.
- The `allowableActions` method in `Terminal` follows the Open-Closed Principle by allowing the addition of new `Buyable` items and `TeleportActions` without modifying the existing code.



```
@Override
public ActionList allowableActions(Actor actor, Location location, String direction) {
    ActionList actionList = new ActionList();
    for (Buyable buyable : this.buyables) {
        actionList.add(new BuyAction(buyable));
    }
    if (actor.hasCapability(Status.HOSTILE_TO_ENEMY) && !location.containsAnActor()) {
        for (Action travelAction : this.travelAction) {
            actionList.add(travelAction);
        }
    }
    return actionList;
```

- The use of the `Buyable` interface allows for the extension of the system with new buyable items, adhering to the Liskov Substitution Principle.

Pros

1. Clear separation of concerns: Each class has a well-defined responsibility, making the code more maintainable and easier to understand.
2. Adherence to SOLID principles: The design follows the Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle.
3. Extensibility: The use of the `Buyable` interface allows for easy addition of new buyable items without modifying existing code.

Cons

1. Limited functionality: The current design focuses on the buying functionality and teleportAction for `Theseus`. Additional features or interactions may require further design considerations.
2. Potential for code duplication: If there are multiple buyable items with similar functionality, there might be some code duplication in their implementations. A prime example of this would be when we implemented the different travel actions in application.

Overall, the final design chosen provides a clean and maintainable structure that adheres to key design principles and allows for extensibility in the future.

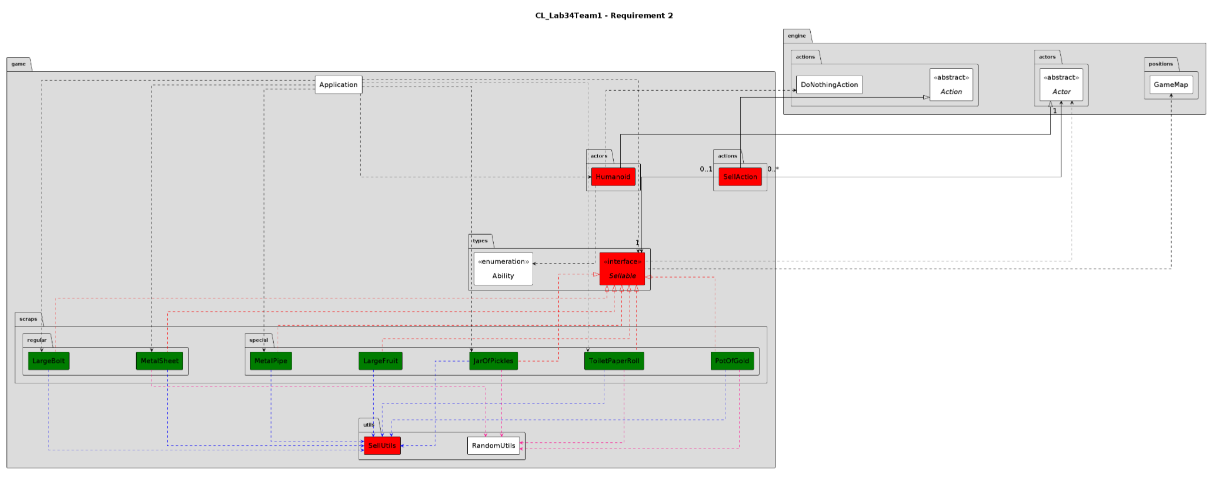
REQ2: The Static Factory

The Static Factory features a humanoid figure (H) in its spaceship parking lot. As the humanoid is a part of the factory, the intern cannot harm or hurt the figure. The intern can sell items to this figure for credits, but they must be careful, because in some situations the sale won't go as expected. Here is a summary of the items they can sell, and what can happen during the purchasing process:

- Large bolts: 25 credits each
- Metal sheets: 20 credits each, but with a 60% chance of the factory asking for a discount (10 credits)
- Large fruits: 30 credits each
- Jars of pickles: 25 credits each, with a 50% chance of the factory paying double (50 credits)
- Metal pipes: 35 credits each
- Pots of gold: 500 credits each, but with a 25% chance of the factory taking the item without payment
- Toilet paper rolls: 1 credit each, but with a 50% chance of the intern being killed instantly by the humanoid figure

(https://claude.ai was used to create a summary of the sellable items)

UML

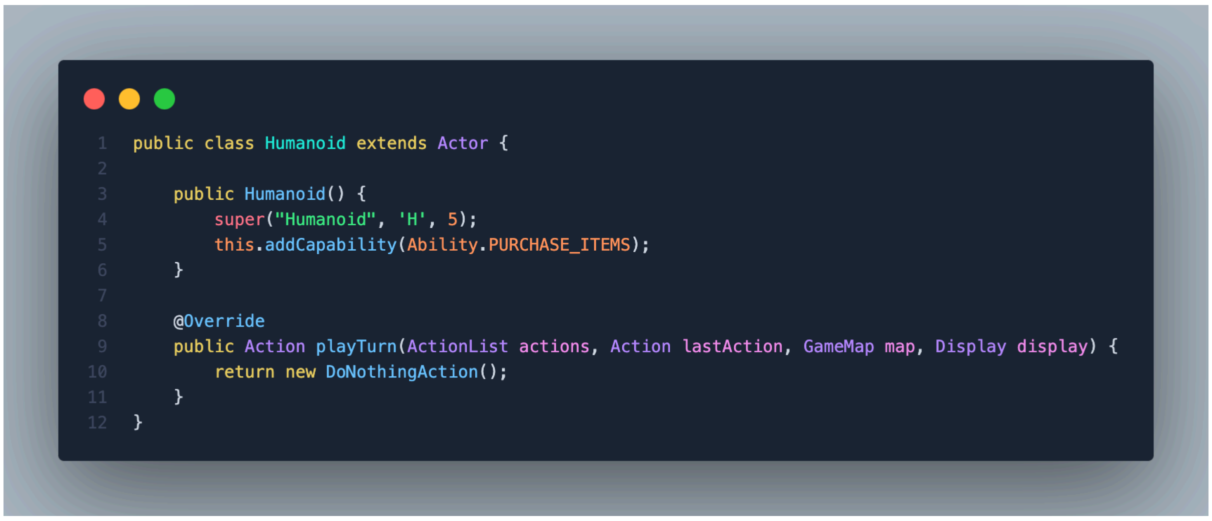


The diagram above represents an object-oriented system for REQ2.

Key Classes Introduced

https://claude.ai was used in parts of this section to summarise code.

Humanoid (concrete class)



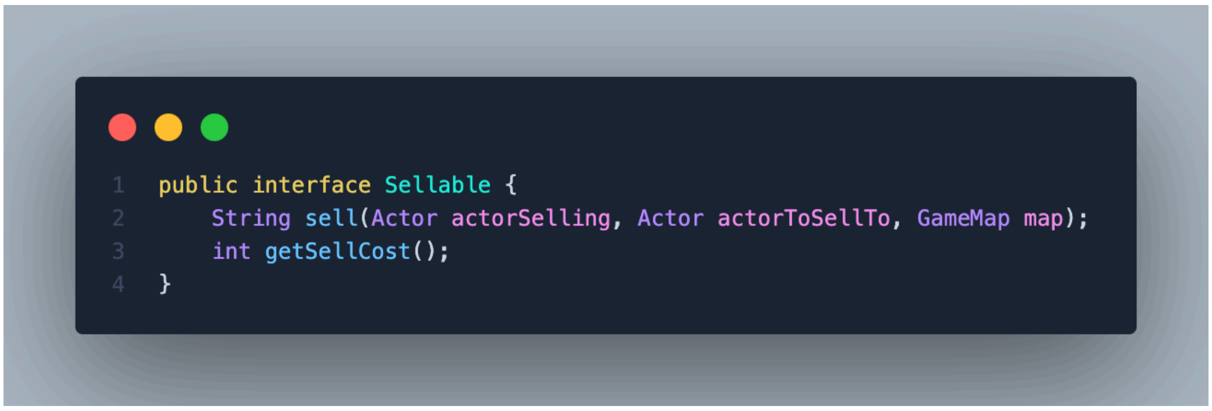
```java
public class Humanoid extends Actor {

    public Humanoid() {
        super("Humanoid", 'H', 5);
        this.addCapability(Ability.PURCHASE_ITEMS);
    }

    @Override
    public Action playTurn(ActionList actions, Action lastAction, GameMap map, Display display) {
        return new DoNothingAction();
    }
}
```

Humanoid extends the Actor class from the edu.monash.fit2099.engine.actors package to represent the Humanoid figure in the game. It has a constructor that initialises its name, display character, and hit points. It also adds a capability called PURCHASE_ITEMS which allows other actors to sell items to it. Since the Humanoid figure doesn't have a purpose at each turn, the playTurn method simply returns a DoNothingAction.

Sellable (interface)

This interface is used to define a contract for objects that can be sold within the Static Factory game. It specifies two methods that any class implementing this interface must provide, sell (which represents the actions of selling an item), and getSellCost (which returns an integer value representing the item's cost).

Any class that implements the Sellable interface must provide concrete implementations for these two methods. This interface is used in conjunction with items that can be bought and sold by actors within the game.



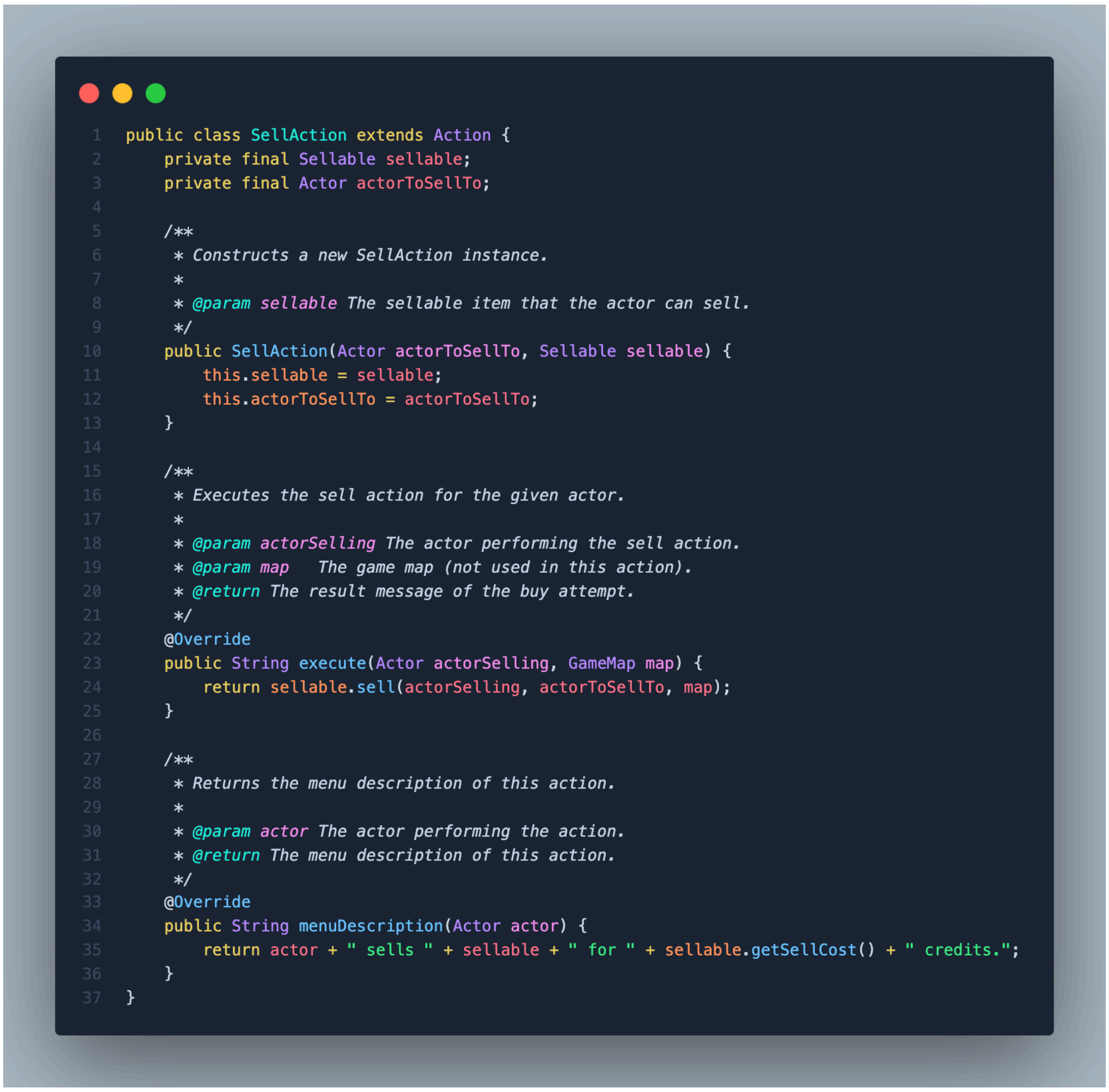
```java
public interface Sellable {
    String sell(Actor actorSelling, Actor actorToSellTo, GameMap map);
    int getSellCost();
}
```

SellAction (concrete class)
This class represents an action that allows an actor to sell a Sellable item to another actor.

Instance Variables
sellable: An object implementing the Sellable interface, representing the item being sold.
actorToSellTo: The Actor to whom the item will be sold.

Execute Method
Responsible for executing the sell action. It calls the sell method on the sellable object, passing in the actorSelling, actorToSellTo, and map (to allow for player deaths). The method returns the result message returned by the sell method of the Sellable item.

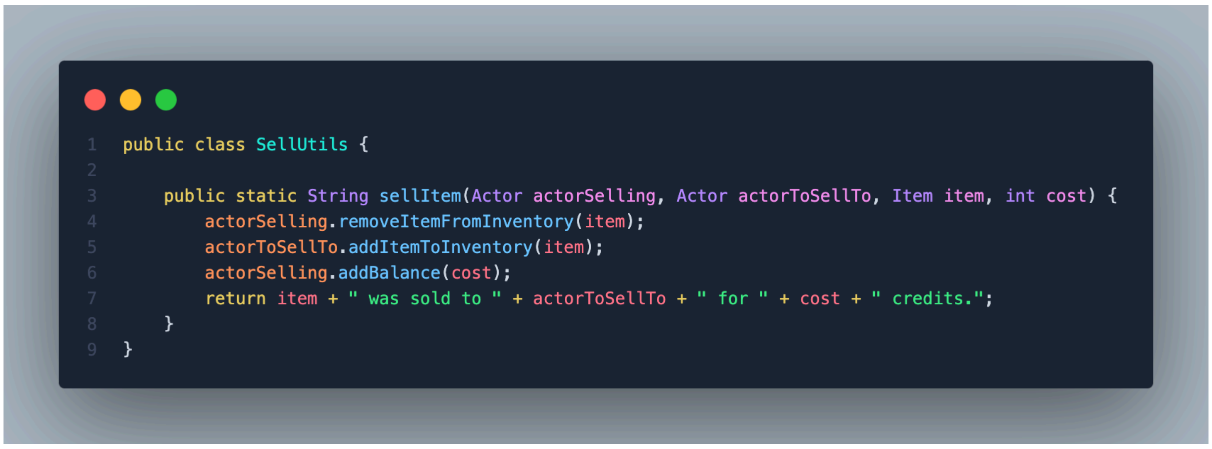
Menu Description
Provides a description of the sell action for display in the game menu. It returns a string in the format: "<actor> sells <sellable> for <sellCost> credits."



```java
public class SellAction extends Action {
    private final Sellable sellable;
    private final Actor actorToSellTo;

    /**
     * Constructs a new SellAction instance.
     *
     * @param sellable The sellable item that the actor can sell.
     */
    public SellAction(Actor actorToSellTo, Sellable sellable) {
        this.sellable = sellable;
        this.actorToSellTo = actorToSellTo;
    }

    /**
     * Executes the sell action for the given actor.
     *
     * @param actorSelling The actor performing the sell action.
     * @param map   The game map (not used in this action).
     * @return The result message of the buy attempt.
     */
    @Override
    public String execute(Actor actorSelling, GameMap map) {
        return sellable.sell(actorSelling, actorToSellTo, map);
    }

    /**
     * Returns the menu description of this action.
     *
     * @param actor The actor performing the action.
     * @return The menu description of this action.
     */
    @Override
    public String menuDescription(Actor actor) {
        return actor + " sells " + sellable + " for " + sellable.getSellCost() + " credits.";
    }
}
```

SellUtils (static utility class)

A utility class that provides a static method sellItem to handle the process of selling an item from one actor to another within the game. The sellItem method performs the necessary operations to transfer the item from the seller's inventory to the buyer's inventory, adjusts the seller's balance by adding the cost, and returns a descriptive message indicating the successful sale of the item along with the involved actors and the cost.



```java
public class SellUtils {

    public static String sellItem(Actor actorSelling, Actor actorToSellTo, Item item, int cost) {
        actorSelling.removeItemFromInventory(item);
        actorToSellTo.addItemToInventory(item);
        actorSelling.addBalance(cost);
        return item + " was sold to " + actorToSellTo + " for " + cost + " credits.";
    }
}
```

Alternative Design Considered

https://claude.ai was used in parts of this section.

Description

An alternative design consisted of creating the selling system the opposite way, such that the responsibility of generating the actions for what could be bought was given to the Humanoid rather than the items themselves. This meant that downcasting would be required to check whether the items in the Humanoid's surrounding actors' inventories were Sellable or not. Although this didn't violate the Open/Closed principle, as it still incorporated the abstraction layer of a Sellable item, it created tight coupling between the Humanoid and Sellable items.

Cons

Coupling

By requiring the Humanoid class to check if an item is Sellable by downcasting, it introduces a tight coupling between the Humanoid and the specific implementations of Sellable items. This tight coupling makes the system more rigid and harder to maintain or extend.

Connascence

The design exhibits a high degree of connascence, specifically "Connascence of Type" (CoT), where the Humanoid class depends on the specific type of Sellable. This kind of connascence makes the system more fragile and harder to change.

SOLID Principles

By giving the responsibility of generating sell actions to the Humanoid class, it violates the SRP, as the Humanoid now has responsibilities beyond what it is intended for, namely determining what items are sellable.

Final Design
https://claude.ai was used in parts of this section.

Description
The final design follows an object-oriented approach to implement the system for selling scraps to the Humanoid by introducing the Sellable interface, the SellAction class, and the SellUtils static class. It adheres to several SOLID principles, including the Single Responsibility Principle, Open/Closed Principle, Interface Segregation Principle, and Liskov Substitution Principle. It promotes code reusability, extensibility, and maintainability by separating concerns, using interfaces, and decoupling the implementation details from the clients. Additionally, the design incorporates concepts like design by contract and low connascence, contributing to better code understanding and reduced dependencies between components.

Pros
Single Responsibility Principle (SRP): The design separates concerns by introducing different classes for different responsibilities. For example, SellUtils is responsible for the actual logic of selling an item, while SellAction encapsulates the selling behaviour for actors.

Open/Closed Principle (OCP): The design is open for extension by introducing the Sellable interface. New sellable items can be added by implementing this interface, without modifying the existing code.

Interface Segregation Principle (ISP): The Sellable interface defines a specific contract for selling items, segregated from other concerns such as buying or performing other actions.

Liskov Substitution Principle (LSP): The scrap classes adhere to the LSP by correctly implementing the Sellable interface, allowing it to be substituted for any other Sellable implementation.

Design by Contract: The Sellable interface and the sell method define a contract that must be followed by any sellable item. This helps in ensuring correct behaviour and facilitates code understanding.

Decoupling: The use of interfaces (Sellable) and utility classes (SellUtils) decouples the implementation details from the clients, reducing tight coupling and improving maintainability.

Low Connascence: The design exhibits low levels of connascence, as the classes and interfaces are not overly coupled or dependent on each other's internal implementations.

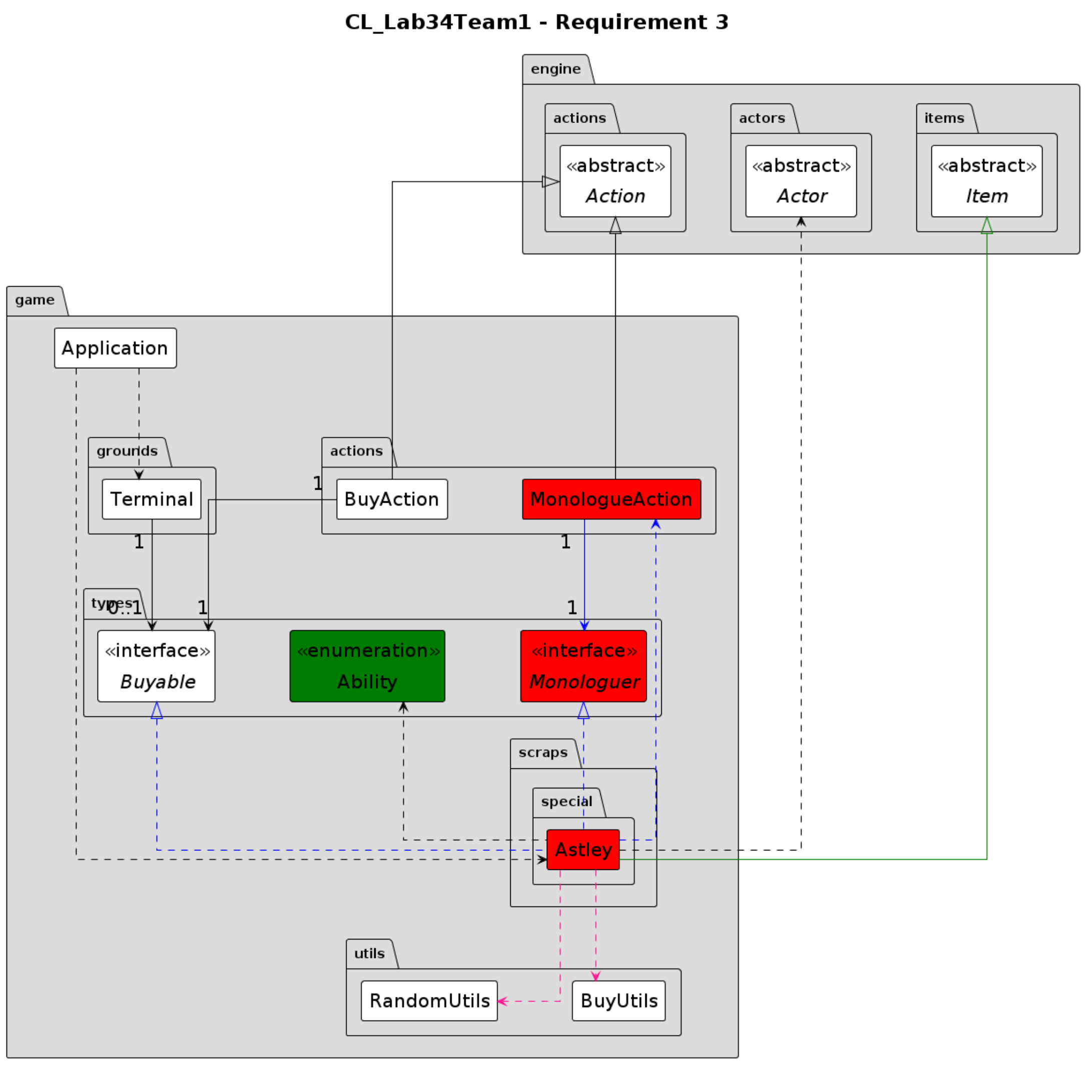
Cons
Potential Violation of the Single Responsibility Principle (SRP): While SellAction encapsulates the selling behaviour, some scrap classes implement both Buyable and Sellable interfaces, which could be seen as violating the SRP. It might be better to separate the buying and selling concerns into separate classes.

Complexity: While the design follows good principles, it might be considered overly complex.

REQ3: dQw4w9WgXcQ

In REQ3, the Intern can now purchase an AI device called Astley for 50 credits from the computer terminal. Astley requires a subscription fee of 1 credit every 5 ticks while in the Intern's inventory. If the Intern cannot pay the fee, they cannot interact with Astley until they have sufficient credits. When the Intern has Astley, they can listen to its monologue. However, some options are only available under specific conditions, such as the intern's inventory size, balance and health.

UML



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

Key Classes Introduced

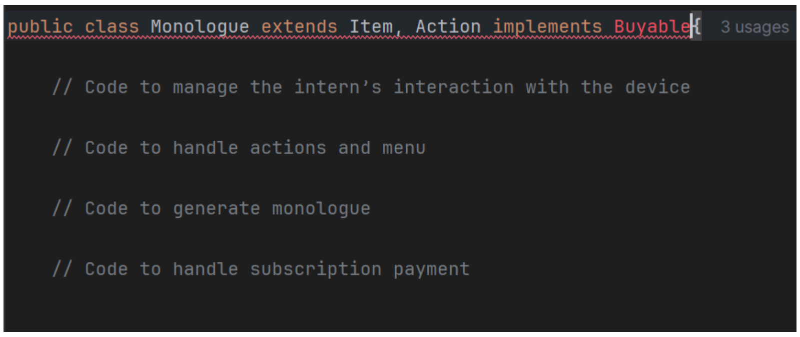
- The Astley class represents the AI device that the Intern can purchase and interact with. It encapsulates the logic for managing the subscription fee, and handling the monologue selection. It extends the abstract Item class from the game engine and implements the Buyable and Monologuer interface.

- The Monologuer interface defines a single method generateMonologue(Actor actor), which provides a contract for any entity that can monologue in the game.

- MonologueAction represents the action that allows an Actor to listen to a monologue generated by a Monologuer.

Alternative Design Considered

One alternative approach could involve using a single class to handle all aspects of the AI device, monologue generation and subscription management. This class would directly manage the intern's interaction with the device, generate monologues and handle subscription payments, all within one large class. However, this violates SOLID principles and does not integrate well with the provided game engine. It also does not make sense in the context of the game engine.



```
public class Monologue extends Item, Action implements Buyable{  3 usages

    // Code to manage the intern's interaction with the device

    // Code to handle actions and menu

    // Code to generate monologue

    // Code to handle subscription payment
```

Pros:

- Simple and easy to understand
- All the relevant code is centralised in one place, making it easily accessible.

Cons:

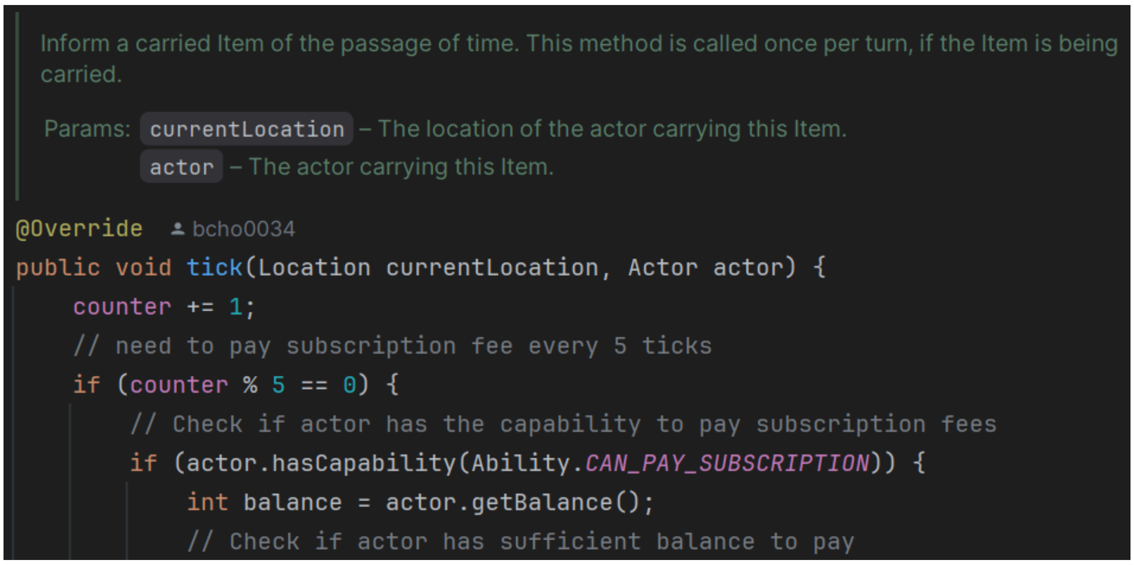
- Violates the Single Responsibility Principle (SRP) as the Monologue class handles multiple responsibilities
- Violates the Open/Closed Principle (OCP) as adding new monologue objects would require modifying the Monologue class
- Potential code duplication if multiple entities need to monologue

- The design does not align with the modular architecture of the game engine, making it less compatible and harder to integrate with other game components.
- Changes in one part of the class could have widespread, unintended effects on other parts of the class, leading to increased difficulty in managing changes. (High Connascence)

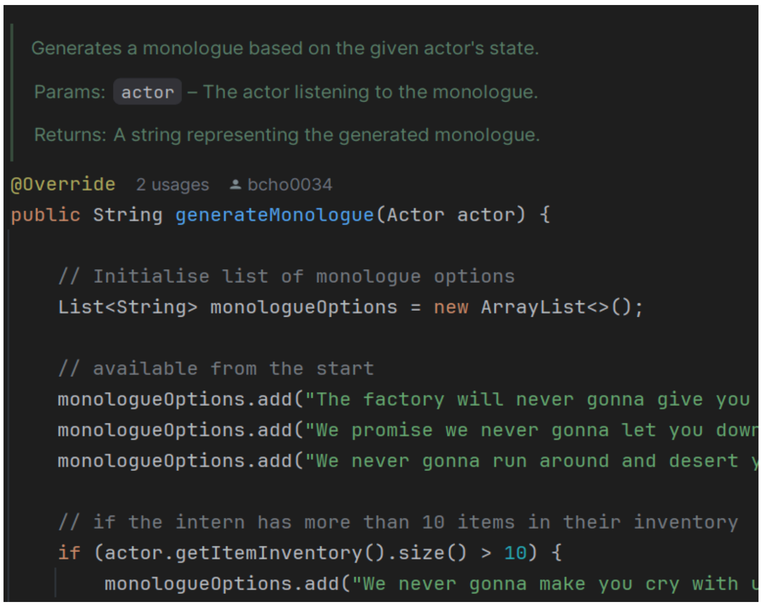
Final Design

The current design uses a modular approach with the Astley class, MonologueAction and Monologuer interface. The Monologuer interface defines the contract for generating monologues, while MonologueAction encapsulates the logic for performing monologue actions. The Astley class implements the Monologuer interface and manages the AI device's subscription logic. This is similar to the design pattern used with the Consumable interface and ConsumeAction classes.

This separation ensures each class has a single responsibility and adheres to SOLID principles. Connascence, design smells and refactoring techniques have been applied to ensure the design is robust and follows our design goals.

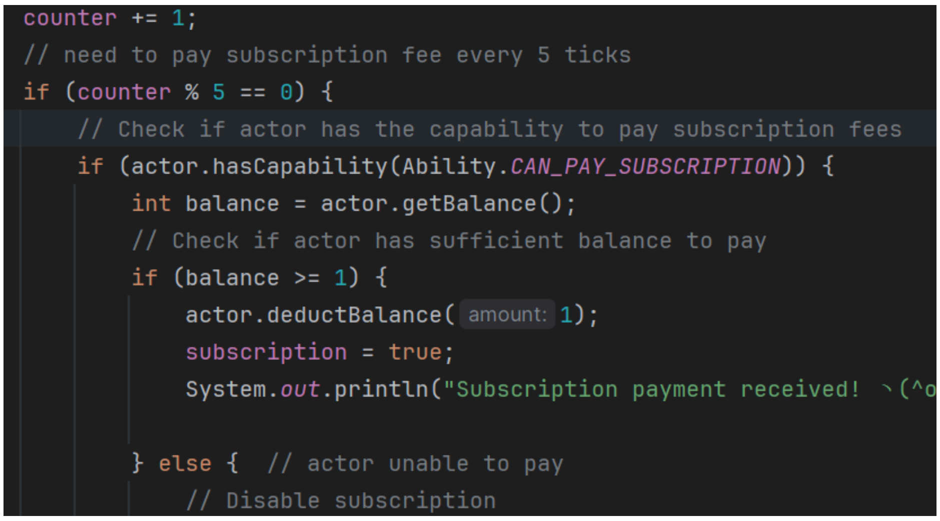


```
Inform a carried Item of the passage of time. This method is called once per turn, if the Item is being carried.
Params: currentLocation – The location of the actor carrying this Item.
        actor – The actor carrying this Item.
@Override  bcho0034
public void tick(Location currentLocation, Actor actor) {
    counter += 1;
    // need to pay subscription fee every 5 ticks
    if (counter % 5 == 0) {
        // Check if actor has the capability to pay subscription fees
        if (actor.hasCapability(Ability.CAN_PAY_SUBSCRIPTION)) {
            int balance = actor.getBalance();
            // Check if actor has sufficient balance to pay
```



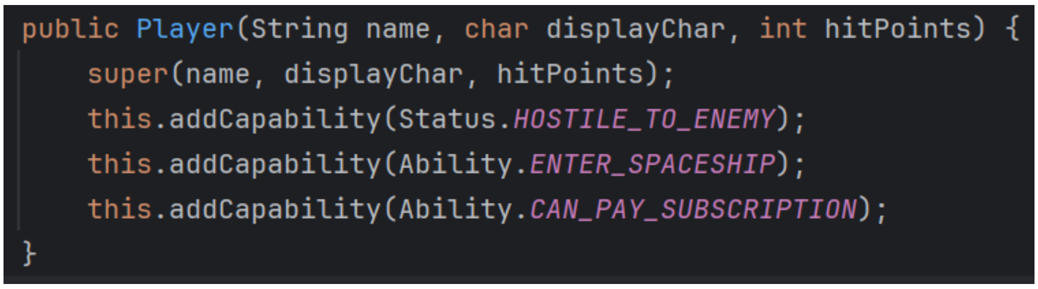
```
Generates a monologue based on the given actor's state.
Params: actor – The actor listening to the monologue.
Returns: A string representing the generated monologue.
@Override  2 usages  bcho0034
public String generateMonologue(Actor actor) {

    // Initialise list of monologue options
    List<String> monologueOptions = new ArrayList<>();

    // available from the start
    monologueOptions.add("The factory will never gonna give you
    monologueOptions.add("We promise we never gonna let you dow
    monologueOptions.add("We never gonna run around and desert y

    // if the intern has more than 10 items in their inventory
    if (actor.getItemInventory().size() > 10) {
        monologueOptions.add("We never gonna make you cry with u
```

In implementing the Astley class, the decision was made to move all the monologue generation logic into the generateMonologue() method specified instead of clumping it together in the tick() method. The tick() method remains focused on its primary responsibility which is handling the subscription fee logic, while the generateMonologue() method encapsulates the logic for generating monologues. This promotes better code organisation and readability.



```
counter += 1;
// need to pay subscription fee every 5 ticks
if (counter % 5 == 0) {
    // Check if actor has the capability to pay subscription fees
    if (actor.hasCapability(Ability.CAN_PAY_SUBSCRIPTION)) {
        int balance = actor.getBalance();
        // Check if actor has sufficient balance to pay
        if (balance >= 1) {
            actor.deductBalance( amount: 1);
            subscription = true;
            System.out.println("Subscription payment received! ヽ(^o

        } else {  // actor unable to pay
            // Disable subscription
```

To ensure proper handling of subscription fee payments and promote code extensibility, I have also introduced a new CAN_PAY_SUBSCRIPTION ability. This ability is checked before deducting subscription fees from an actor, and it is assigned to actors like the Intern (Player class) capable of paying subscription fees as shown below.



```java
public Player(String name, char displayChar, int hitPoints) {
    super(name, displayChar, hitPoints);
    this.addCapability(Status.HOSTILE_TO_ENEMY);
    this.addCapability(Ability.ENTER_SPACESHIP);
    this.addCapability(Ability.CAN_PAY_SUBSCRIPTION);
}
```

Connascence of name is also considered in the design by defining clear interfaces (Monologuer) and method names (generateMonologue) that convey their purpose without relying on implementation details.

Pros:

- Each class has a single responsibility, making the code easier to maintain and extend (SRP).
- This design also avoids the large class smell by breaking down functionalities into smaller, cohesive classes (Monologuer, MonologueAction, Astley) with distinct responsibilities (Large Class Smell).
- New objects that can monologue can be easily added just by implementing the Monologuer interface, without modifying existing code (OCP).
- The Astley class can be substituted for the Monologuer interface without violating correctness, as it adheres to the contract defined by the interface (LSP).
- The classes depend on abstractions (Monologuer) rather than concrete implementations, promoting loose coupling and flexibility (DIP).
- By following the SOLID principles and design patterns, the classes have low connascence, reducing tight coupling and promoting code extensibility and maintainability.

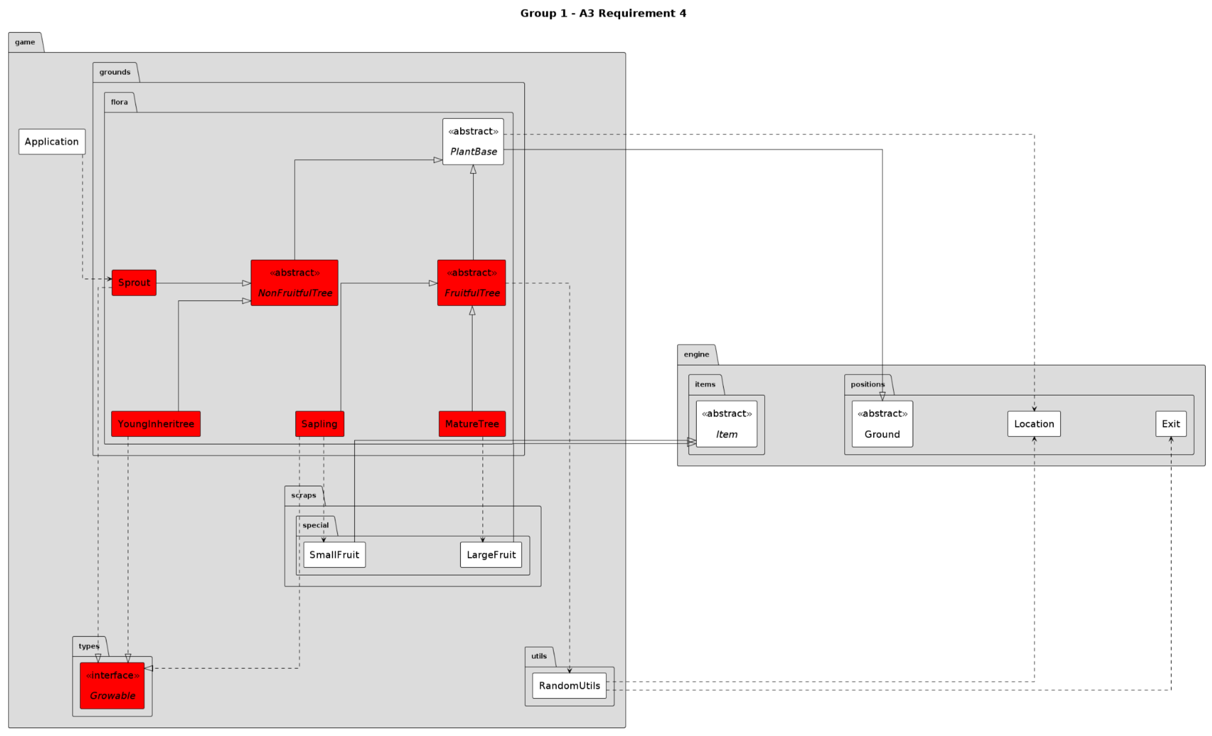
Cons:

- Increased complexity compared to the single class implementation, as the design involves multiple classes and abstractions.

REQ4 [Survival Mode]: Refactorio, Connascence's largest moon

In requirement 4, the assignment 1 Inheritree needs to be refactored and further stages need to be added.

UML



The diagram above represents an object-oriented system for REQ4 with the red indicating new classes.

Key Classes Introduced

- FruitfulTree - an abstract class that is used for other classes that are at a stage that can produce fruit
- NonFruitfulTree- an abstract class used for other classes that are at a stage that cannot produce fruit
- Growable - an interface that is used to provide the blueprint for classes that can grow to a new stage of tree

- Sprout, Sapling, YoungInheritree and MatureTree - all represent different stages of the Inheritree life and have different functionality.

Alternative Design Considered

My alternative design was to implement all stages of the tree from the PlantBase abstract class and have two interfaces, Growable and Fruitful for the functionality to drop fruit and grow to a new stage. However with this, I was seeing myself repeating lots of code within the tick method to drop new fruit for the stages that could. Besides this, this design did follow all OOP with not many other trade offs.

Pros:

- Adheres to the Single Responsibility Principle (SRP) by representing all stages of the trees separately and providing functionality as interfaces
- Follows the Interface Segregation principle as no class has to implement functionality that wasn't required.

Cons:

- Lots of repeated code in all the produce fruit tick methods

Final Design

The new approach that I implemented was to remove the Fruitful Interface and instead create another level of abstraction between the PlantBase and the Tree stages. I created two abstract classes FruitfulTree and NonFruitful Tree. The first benefit I saw with this is that the Fruitful tree could handle the spawning of all the fruits and I would not have to repeat any of that code like in my previous design, following the DRY principle.

Keeping the grow interface for all the stages that could go to another stage was helpful in providing the required methods for a tree that could grow and follow the ISP as MatureTree cannot grow and it would be redundant if it had to implement that method.

Pros:

- New stages of the tree can easily be added without modifying existing code. Also little code needs to be done to add these new classes (OCP).
- Interface is used to provide the logic for all classes that need it (ISP)
- The ProduceFruit method code is used for all trees that produce fruit and there is no repeated code like the previous design (DRY)
- All classes have their own responsibility (SRP)

Cons:

- Requires another level of abstraction which is now 3
- The tick method still has some repeated code for the tree to grow