Assignment 1 - Report

Lachlan Riley MacPhee
33109176

Summary

Static Factory is a text-based “rogue-like” game written in Java that was inspired by the game Lethal Company.

Rogue-like games are named after the first such program: a fantasy game named Rogue. They were very popular in the days before home computers were capable of elaborate graphics and still have a small but dedicated following amongst gamers and computer enthusiasts.

This version of Static Factory, set on the Moon, uses object-oriented design patterns to implement four key requirements:

- A player, known as the Intern, of the factory who can interact with the world and fight creatures.
- Flora such as the Inheritree which starts as a sapling and can grow to a mature tree.
- Hostile fauna such as the HuntsmanSpider who wanders around the map until it finds an Intern to slash with its long legs.
- Special scraps (such as fruits and weapons) that appear on the ground and provide the player with health boosts or increased abilities.

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 above requirements, but does so in a manner that conforms to professional software development standards.

Extensibility

Ensuring the system's extensibility was a key goal. I wanted to make it easy to add new features, fauna, flora, 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.

Maintainability

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

How the game works

The game is built on top of an existing engine that was provided.
Here is a step-by-step overview of what is happening in the code, starting from when the Application class is run. These steps only include the key modifications I have made to the game, rather than what was in the existing codebase.

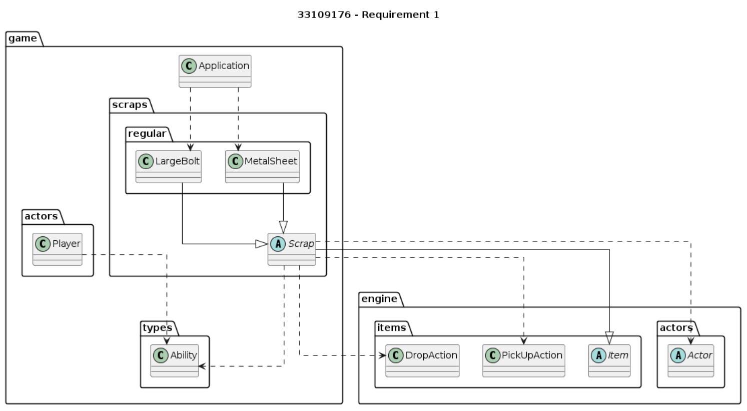
1. Define a game map that incorporates both types of Inheritree (sapling and mature)
2. Add the new scraps to the game map at various coordinates
3. Add the Crater to the game map at various coordinates (and pass them the enemy to spawn)
4. On each tick of the game, all living things on the game map can perform some action, for example:
    - Sapling Inheritrees have the chance to grow to mature Inheritrees if they are over the age of 5
    - Enemies such as the HuntsmanSpider can wander around the map, or attack the Intern if they are close to it
    - The Intern can pick up scraps from the floor (and use them now or in future) or fight enemies that are near them
5. If at any stage the Intern's health becomes 0, they will be fired from Static Factory and the game will be over.

Design Choices

This section is broken down by each key requirement and is accompanied by the UML diagrams for these requirements so that there is a visual aid for the explanations.

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

The Intern of the Static Factory



(this diagram is also available in */docs/design/assignment1/umls*)

Key Classes Introduced

- Scrap - an abstract class (extending from Item) that defines a portable piece of “scrap” for inheritance. It implements the pick up and drop methods so that its children don’t have to.

Pros

- Both types of scrap extend the abstract class Scrap to prevent repeated code across scraps (DRY).
- Only the Scrap class needs to be modified if we need to implement buying/selling (or other features) in the future. The child classes such as LargeBolt and MetalSheet can remain as is.
- Subtypes of Scrap are substitutable for the base type, ensuring consistent behaviour and adhering to Liskov’s Substitution Principle (LSP)
- New scraps can be added easily by extending the existing abstract scrap class (OCP)
- Adding the enum to pick up / drop the scraps in the Ability class prevents ambiguity by detailing exactly which Actors can pick up scrap. This can be overridden in the sub-classes if needed.

Cons

- Extra levels of inheritance may increase complexity for new developers, potentially violating the KISS (Keep It Simple, Stupid) principle.

How it satisfies design goals

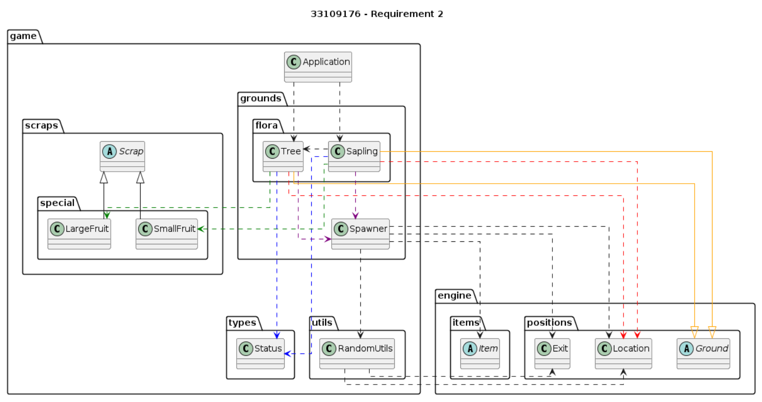
Extensibility

The use of the abstract Scrap class and adherence to the OCP and LSP allow for easy extension of the system with new types of scraps without modifying existing code.

Maintainability

The DRY principle helps reduce code duplication, making the codebase more maintainable. However, the extra levels of inheritance may negatively impact maintainability.

The Moon's Flora



(this diagram is also available in */docs/design/assignment1/umls*)

Key Classes Introduced

- 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
- Tree/Sapling are plants that have the status of ALIVE and in the case of sapling, can grow into another plant (the Tree)

Pros

- The decision not to add unnecessary complexity around tree growth aligns with the YAGNI (You Ain't Gonna Need It) principle, especially as this requirement is not detailed in the Assignment 2 specs.
- The Spawner is given the responsibility for creating fruits, separating concerns (SoC) from the Tree and Sapling classes.
- The RandomUtils class is solely responsible for generating random values/things (e.g., finding a random exit), adhering to the SRP and promoting maintainability.

Cons

- If we need to implement more trees with multiple growing layers in the future, we may need to re-architect the current design, potentially hindering extensibility.

Alternative approach

- An abstract Tree/Plant class with default implementations and the ability to pass in what it will grow into could improve extensibility for evolving plants.

How it satisfies design goals

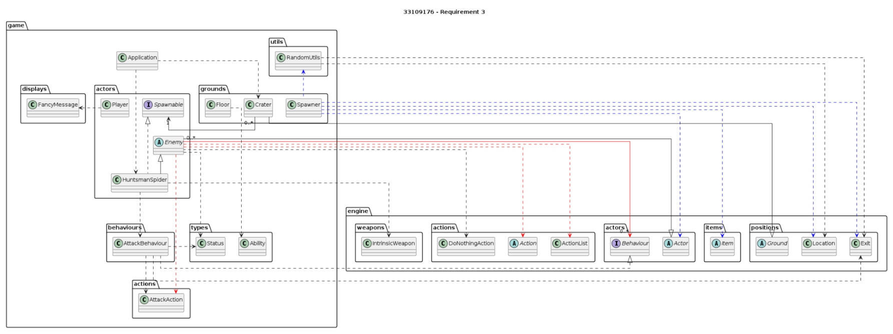
Extensibility

Any other objects that can spawn actors/items can easily use the spawner class in future. Though, the current design may not be optimal for extending the system with evolving plants.

Maintainability

The use of YAGNI, SoC, and SRP promotes maintainability by avoiding over-engineering, separating concerns, and adhering to good design principles.

The Moon's (Hostile) Fauna



(this diagram is also available in */docs/design/assignment1/umls*)

Key Classes Introduced

- Spawnable - an interface for spawnable actors that allows them to create new instances of themselves (for extensibility purposes)
- AttackBehaviour - a class that defines the behaviour of enemies (such as HuntsmanSpider) to attack actors who are hostile to them
- Crater - a class that spawns a given actor (currently just enemies) with a specific spawn chance
- Enemy - an abstract class that defines an actor that is an enemy of the player

Pros

- Classes such as the HuntsmanSpider implement the Spawnable interface so they can be spawned by a crater, promoting loose coupling and ensuring there is no multi-level inheritance.
- The Crater depends on the Spawnable interface, adhering to the DIP and promoting modular design with no direct dependence from the Spawner to the actors.
- We can add other spawnable actors without modifying existing code, adhering to the OCP.

Cons

- The added complexity of the Spawnable interface and related classes may violate the KISS principle, potentially hindering maintainability.

Alternative approach for the crater

- A factory (eg EnemyFactory) that can instantiate enemies could be used, but this would require an enum or keys and extending a switch statement each time a new enemy was added.

How it satisfies design goals

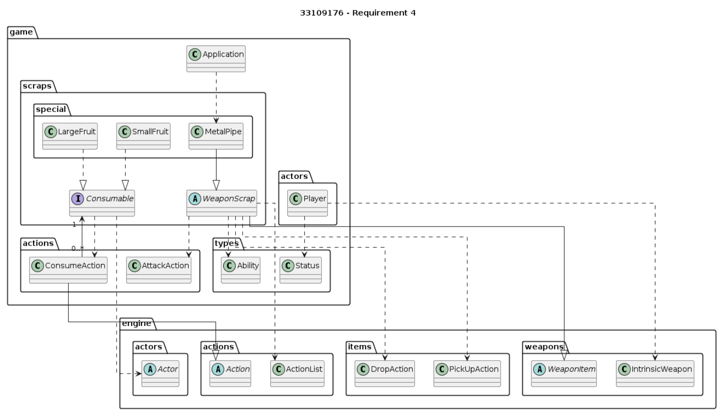
Extensibility

The use of the Spawnable interface, adherence to the OCP, and loose coupling promote extensibility by allowing the addition of new spawnable actors without modifying existing code.

Maintainability

While the added complexity may hinder maintainability to some extent, the use of the DIP and loose coupling promote modular design and maintainability.

Special Scraps



(this diagram is also available in */docs/design/assignment1/umls*)

Key Classes Introduced

- Consumable - an interface for anything that is considered consumable (ie able to be consumed by the player
- ConsumeAction - an action that accepts a Consumable and consumes it when the menu option is called, it also handles what is displayed in the menu.
- WeaponScrap - similar to the Scrap class, but for WeaponItems rather than Items.

Pros

- The *game.scraps.special* package defines scraps that have the ability to do something unique, adhering to the SRP and promoting high cohesion within the package.
- The ConsumeAction depends on an item that implements the Consumable interface, adhering to the DIP and promoting modular design.

Cons

- The WeaponScrap class has some repeated code that is also present in the Scrap class, violating the DRY principle. This is due to the fact that we have Items and WeaponItems in the base class, so is more of a constraint created by the engine code.

How it satisfies design goals

Extensibility

The use of the Consumable interface and adherence to the DIP promote extensibility by allowing the addition of new consumable scraps without modifying existing code.

Maintainability

The adherence to the SRP and high cohesion within the package promote maintainability. However, the violation of the DRY principle in the WeaponScrap class may hinder it.