Design rationale

REQ1

Single Responsibility Principle (SRP):

Each class in the design has a single responsibility, focusing on one aspect of the system's functionality.

- TeleportAction handles the logic for teleporting an actor to a random location.
- Theseus represents an item that can be printed from a computer and has the capability to teleport an actor.
- ComputerItem serves as a base class for all computer-generated items, providing common attributes and methods.
- InsufficientBalanceException and UnavialableTeleportExecption handles specific exceptions related to item purchase and teleportation.

Pros of the design:

- Enhances code maintainability and readability by ensuring each class has a clear and focused purpose.
- Reduces the likelihood of bugs and makes it easier to modify the code in the future.

Open/Closed Principle (OCP):

The classes are designed to be open for extension but closed for modification.

- New types of computer items can be created by extending this class without modifying its existing logic.
- TeleportAction can be extended or reused in different contexts without altering its core functionality.

Pros of the design:

- Promotes code reusability and extensibility.
- Minimises the risk of introducing bugs when adding new features, as existing code remains untouched.

Inheritance:

- ComputerItem Acts as a base class for Theseus and potentially other items, allowing them to share common functionality and be extended to add specific behaviours.

Concrete Class (Theseus): Theseus is a specific implementation with concrete behaviours, such as the special method and teleportation action, which are not just a contract but actual functionality.

REQ2

Single Responsibility Principle (SRP):

Each class in the system has a single responsibility, focusing on one aspect of the game's functionality.

- NewMoon and Polymorphia represent specific game maps with predefined layouts.
- Refactorio is another specific game map with its own layout and additional world management features.
- HumanoidFigure represents a non-player character in the game that can buy items and award players.

Open/Closed Principle (OCP):

The classes are designed to be open for extension but closed for modification.

- TraversalMap Can be extended to create new maps like NewMoon, Polymorphia, and Refactorio without modifying its core logic.
- SellAction can be extended to handle different types of sale actions without modifying its core functionality.
- New SaleableItem can implement this interface without altering existing implementations.

Pros of the design:

- Promotes code reusability and extensibility.
- Minimises the risk of introducing bugs when adding new features, as existing code remains untouched.

Polymorphism

- SaleableItem defines a contract for items that can be sold, allowing different implementations to be treated uniformly.

Cons of the design:

- There are some minor code duplications on the allowableAction method in items that can be sold.

REQ3

Single Responsibility Principle (SRP):

- SaleableItem is an interface that defines items that can be sold.
- SellAction is responsible for handling the sale of an item.
- Astley class manages the behaviour of an AI device named Astley, including its purchasing logic, lifecycle management, and ability to speak.

Enhances code maintainability and readability by ensuring each class has a clear and focused purpose and makes it easier to understand and modify the code in the future.

Open/Closed Principle (OCP):

Classes are designed to be open for extension but closed for modification.

- New behaviours or features can be added to the Astley AI device by extending the class or adding new methods without modifying existing code.

Polymorphism:

The SpeakAbleEntity interface defines a contract that specifies a method Speak(). The Astley class implements this interface and provides its own implementation of the Speak method. This allows for polymorphic behaviour, as objects of different classes that implement SpeakAbleEntity can be used interchangeably where a SpeakAbleEntity is expected.

Dependency Inversion Principle (DIP):

The Astley class depends on abstractions (ComputerItem, SpeakAbleEntity) rather than concrete implementations. This reduces coupling between components and makes the system more modular and easier to maintain.

Pros of the design: Increases code flexibility and reduces dependency on specific implementations and makes it easier to swap out components without affecting other parts of the system.

REQ4

Single Responsibility Principle (SRP):

Each class in the system has a single responsibility, focusing on one aspect of the game's tree functionality.

- The sprout class represents the initial stage of a tree with a specific display character and growth to the next stage (Sapling).

- Polymorphia represents a specific game map with its own layout, name, and terminal coordinates.
- Refactorio represents another specific game map with its unique layout, name, and terminal coordinates.
- Sapling class represents a stage of the tree that drops small fruits and grows into a YoungInheritree.
- YoungInheritree represents a young stage of the tree with growth to LargeInheritree.
- LargeInheritree represents the final stage of the tree that drops large fruits.
- Abstract class DropFruitPlant for plants that can drop fruits, handling the fruit dropping logic.
- Abstract class Plant for general plant behaviour, including ageing and stage transition.

Pros of the design:

Enhances code maintainability and readability by ensuring each class has a clear and focused purpose and Reduces the likelihood of bugs and makes it easier to modify the code in the future.

Open/Closed Principle (OCP):

The classes are designed to be open for extension but closed for modification.

- TraversalMap acts as a base class for specific maps like Polymorphia and Refactorio, allowing new maps to be created by extending it without modifying its core logic.
- Plant class can be extended to create new tree stages like Sprout, Sapling, YoungInheritree, and LargeInheritree without modifying its core logic.
- New fruit-dropping plants can extend DropFruitPlant class and implement specific fruit-dropping behaviour.
- Inheritree and LargeInheritree: Different stages of the tree can be added by extending the Plant or DropFruitPlant class.

Pros of the design:

- Promotes code reusability and extensibility and minimises the risk of introducing bugs when adding new features, as existing code remains untouched.

Polymorphism:

- DropFruitPlant defines a contract for plants that can drop fruits, allowing different implementations to be treated uniformly.
- abstract class Plant provides a template for various plant behaviours.

Inheritance:

- Plant class acts as a base class for specific stages like Sprout, Sapling, YoungInheritree, and LargeInheritree.

- TraversalMap serves as a base class for specific maps, enabling shared behaviours and attributes while allowing specific implementations.
- Polymorphia and Refactorio extend the TraversalMap class, inheriting its properties and methods while providing specific implementations for map layouts.
- DropFruitPlant serves as a base class for fruit-dropping plants, enabling shared behaviours and attributes while allowing specific implementations.

Pros of the design:

- promotes code reuse and flexibility by allowing different implementations to be used interchangeably and enhances maintainability by providing a clear structure for extending functionality.

Design:

- The Plant class handles both the ageing and the transformation to the next stage, which are closely related responsibilities. This seems reasonable because ageing and growth are inherently linked.

- The DropFruitPlant class extends Plant and adds the responsibility of dropping fruit. While this adds another responsibility, it is logically connected to the plant's life cycle and does not overly complicate the class.

- The design allows for easy extension. Adding another tree stage would involve creating a new class that extends Plant or DropFruitPlant, specifying the next stage and its age. No existing code needs to be modified, adhering to the Open/Closed Principle.

- For a new type of tree, one would similarly extend Plant or DropFruitPlant and implement the required methods. This design supports easy extension without modification of existing classes.

- The design avoids forcing trees that cannot drop fruit to implement unnecessary methods. Only DropFruitPlant and its subclasses need to implement the dropFruit method. This keeps the Plant class free from unnecessary methods, adhering to the Interface Segregation Principle.

- There is some duplication in the tick methods of DropFruitPlant subclasses, where each subclass has its own implementation for dropping fruit. This duplication is minimised by having a common tick method in DropFruitPlant that handles the drop rate logic. Each subclass only implements the specific dropFruit method.

- To further reduce duplication, a helper method for the random exit selection could be abstracted into a utility class or added to the DropFruitPlant class.