Design Rationale for Requirement 1: Let it Grow!

Explanation of system

This system controls the trees and their actions. We have 3 types of trees: sprouts, saplings, and matures. One of their core actions is growing: the sprout is able to grow into a sapling after 10 turns, and the saplings are able to grow into matures after 10 turns. The matures also spread a new sprout every 5 turns, but also has a 20% chance to wither and die each turn, turning into dirt. Their other core action is spawning, the sprouts are able to spawn goombas at a 10% chance each turn, the saplings can drop coins at 10% each turn, and the matures are able to spawn koopas at 15% chance each turn.

Choice

Making Tree an abstract class

Justification

This enables the efficient creation of multiple different sub-classes of tree, namely the sprout, sapling and mature trees. Their common/shared attributes and functionality can be inherited from the base Tree class, which supports the Don't Repeat Yourself (DRY) principle and reduces redundancy.

Choice

Addition of the Sprout, Sapling and Mature sub-classes of Tree

Justification

These new sub-types of trees will have different functionality that distinguish them from each other. Their shared attributes and functionality will be inherited from the parent class Tree.

Choice

Using an interface GrowCapable to provide the functionality for certain trees to grow.

Justification

This implements the DRY principle, as by using an interface we are reducing redundancy of defining the similar method for each class. Instead, they can each have their own implementation of the single method under the same function name. It also supports the Open-Closed Principle from the SOLID principles of object-oriented design, as by using this interface to create the grow function, we are opening the system for the extension of adding new types of trees in the future that may use this method but closing the existing types of trees for modification.

Choice
Actors are not stored in locations (as initially planned)

Justification
The initial plan was to store actors in locations, which would enable easy movement between locations, as well as helping provide the functionalities of stopping trees from spawning new actors when an actor is already standing on them. This was not done, instead actors’ locations are stored in an array in the GameMap. Using this, the actors can be easily moved between locations, and locations are given the ability to check if an actor is there, which is used to determine whether trees can use their spawning abilities.

Design Rationale for Requirement 2: Jump

Added package for tree

A tree abstract class is created as the sprouts, saplings, and mature share the same methods. As saplings, sprouts, mature and trees are all related, a package called tree is created to ensure encapsulation.

Added package for implementedActions

As all the actions to be done by the different actors are related, a package called implementedActions is created.

Design Rationale for requirement 3: Enemies

Notes

All changes made to the existing UML diagrams to get the submitted UML diagrams are highlighted and justified in this document. Each class has its own behaviours and serves different purposes. This implements Single Responsibility Principle (SRP). The pros are the code is simpler and shorter and the program is easier to maintain. However, repetition of code may occur, and this violates DRY principle.

In the assignment, we are inheriting attributes and methods from abstract class. By extending an abstract class, DRY principle is implemented. This means codes can be reused and can be shared between classes that have common traits. As a result, codes do not need to be re-written, resulting in a simpler and shorter codebase.

Class

A new class Koopa was created, and several types of relationships are implemented for this class. (Includes changes made to abstract class Tree)

Choices/Explanations

This class was created to represent the other type of enemy called 'Koopa.' It shares similar methods and attributes to that of existing enemy type Goomba.

The class Koopa is also linked to the abstract Class 'Actor' as it is part of the game as an actor.

The class Koopa is linked to the interface Behaviour as it could potentially implement similar behaviour as other actors. By implementing an interface, related methods and attributes can be grouped together. An interface will ensure the child class implements these methods and attributes by overriding them. This would avoid careless mistakes like forgetting to write an important method for a certain class.

The abstract class Tree has a dependency to the class Koopa because it has a chance to create an instance of Koopa based on a specified probability and condition.

The Koopa class uses capabilities (via the actor class) to know if it can be killed based on whether the actor (Mario) possess a weapon (a wrench). This creates a dependency from Koopa to actor, which is not shown is it is overridden by the association between the same 2 classes (Koopa is an actor).

Class

The class Goomba has been modified to have additional relationships

Choices/Explanations

The abstract class Tree has a dependency to the class Goomba because it has a chance to create an instance of Goomba based on a specified probability and condition.

Application Class now has a dependency on the abstract Enemy class instead of having individual dependency on each type of enemies.

Design rationale for Requirement 4: Magical Items

Class

Magical items package with Super Mushroom and Power Star

Choices/Explanation

A magical items package was made to keep together similar items. Super Mushroom and Power Star both extend from Item as they are items and need to inherit all of the necessary methods and variables.

Powerstar uses the tick method to keep track of how many turns are left for consumption and effects. We are able to keep track of this using a powerStarCount variable that increments each turn.

Choice

Destroy Wall Action, Consume Item Action and Pick Up Magical Item actions added

Explanation

All of these actions were added to make it easier to implement the magical items. Destroy wall is an action used by the player that enables them to convert a wall to dirt when they are near it.

Furthermore, consume item action allows for items to be consumed. That is, removed from the inventory and from the game entirely but with effects added to the player. The action then is removed from the console. This is needed for magical items as they aren't just picked up and dropped but instead eaten by the player.

PickUpMagicalItem is an action added that is specific for magical items. It takes into account the slight differences between normal items and magical items as sometimes magical items cannot be dropped. And when some are picked up, they need to be consumed.

Class

BuyableItem interface

Choices/Explanation

The magical items implement the buyableitem interface as they all have a price. This is relevant for the trading with Toad requirement.

Class

Updating Status

Choices/Explanation

We have decided to update the Status enums with different capabilities as this is what determines and grants the player their effects and invincibility.

When a magical item is consumed using the consume item action, the player is granted certain statuses like TALL and INVINCIBILITY. This tells the program what effects need to be in place when these capabilities are added.

Design Rationale for Requirement 5: Trading

Explanation of system

This system enables the trading functionality, by which the player can interact with Toad, and trade coins for useful items. The system involves Toad's interactions, which can be to speak to the player, or trade with the player. It also shows that items can be held by Toad, and when traded for can be transferred to the player. (Note: items can also be found on the ground). It also shows that the player can collect coins while playing, and their value will be added to the player's wallet.

Choice

Adding Toad as a new actor

Justification

Toad is a friendly NPC actor who can be interacted with by Mario. He can speak to Mario, or trade with him.

We have implemented him as a child of the abstract class actor. This allows him to inherit some of the similar characteristics and methods like playTurn and allowableActions. This allows toad to have actions that can be performed on him like speaking and buying.

Choice

Giving Toad the interactions of trading

Justification

We have created a BuyAction method. These are added to Toad's allowableActions so that the player can perform these actions when they are nearby. The BuyAction checks the player's wallet to see if there are enough funds and if so, adds the item to be bought into the inventory and subtracts the relevant funds. We add a new buyaction method for each item toad is selling.

Choice

Collecting and tracking of Coins

Justification

Coin objects can be picked up from the ground or from sapling trees that spawned them. When picked up by Mario, the coin object will be removed, and the value of the coin will be added to Mario's wallet. The wallet will only be an attribute of Mario's player character, and so the coins are no longer necessary. We decided to implement the collection of coins this way as it is the most efficient since we do not have to store extra objects, and does not lose any functionality as the coin objects are no longer needed after being collected.

Design Rationale for Requirement 6: Monologue

Explanation of system:

Mario will have access to a Speak action whenever he is near Toad. The Toad class will use this speak action to determine which line to output to the console.

Choice

Adding a SpeakAction action.

Justification

We need an action that allows Mario to speak to Toad who is another actor. This means that we can easily implement an action that allows two actors to speak to eachother.

We add this SpeakAction to Toad's allowableactions as this allows it to appear in the console whenever mario is nearby Toad.

We have also checked every condition which determines different voice lines by toad. We have checked Mario's inventory for a wrench and a power star and give the console a chance to output the relevant lines based on what is in Mario's inventory.

Design Rationale for Requirement 7: Reset Game

Explanation of system

This system enables the game reset functionality, which the player can use once per playthrough, to clear an overwhelming map. Resetting the game will have the following effects:

- Trees have a 50% chance to be converted back to Dirt
- All enemies are killed.
- Reset player status (e.g., from Super Mushroom and Power Star)
- Heal player to maximum
- Remove all coins on the ground (Super Mushrooms and Power Stars may stay).

Choice

ResetManager class

Justification

This class pretty much runs the whole reset. Objects are added to the resettable list and then once the game is reset, we loop through each object and call the resetinstance method. This then runs all the necessary code to reset the game.

Choice

Resettable interface

Justification

This resettable interface is used so that objects that need to be reset can be. They all implement the interface and when instantiated, are added to the resettable list of objects. That is ultimately the interface.

Choice

Removing trees

Justification

When the reset is called, all trees have a 50% chance to be converted back to dirt. This includes any sub-class of the Tree abstract class, namely sprouts, saplings, or mature trees. This is done by getting a random double between 0 and 1. If it is less than or equal to 0.5, we set the ground to dirt, removing the tree. Otherwise the tree is kept.

Choice

Removing coins and killing enemies

Justification

These are done with quite standard implementation. Any coins on the map when reset is called are removed by accessing the locations removeItem() method.. The player's coins however, are maintained. Any enemies on the map are instantly killed/removed by using the removeActor method on the location.

Choice
Changing player health and status

Justification
When reset is called, the player’s current health should be reset to its maximum value. This may differ from the starting value, if the player has increased their maximum health through items such as the Super Mushroom. Any Status effects on the player at the time of reset, should also be cleared. This includes buffs from the Power Star, or Super Mushroom.

This is completed by using the set hp method in Player to update the player’s health to the maximum.

We then call removeCapability() to remove all buffs the player has been granted by from the super mushrooms and power stars.