REQ2 – Design Rationale – Assignment 1 – Dean Mascitti- 33110360

Design Goals

The aim of this requirement is to implement the functionality of Inheritree in that its ability to drop small and large fruits every tick, and its ability to mature after a certain number of ticks and therefore start dropping large fruits as opposed to small fruits.

The goal in the design for this requirement is to ensure that the program is extensible in the sense that if more plants are added to the program, they are easy to implement and don't require a lot of refactoring or repeated code. The use of abstraction therefore is a goal of this requirement through use of interfaces as well as abstract classes.

Design & What OOP Principles Were Applied

In terms of the Inheritree implementation the main strategy of the design was to create an abstract class called DroppablePlant, in which plants who are able to drop fruits, or any other consumable would inherit. In addition, the use of a Spawnable interface was utilized to give DroppablePlant the ability to drop items and to group them with other object/classes/grounds that can also spawn/drop things. DroppablePlant inherits from Ground as since plants are not mobile it would suit the functionality of the Ground, in which is programmed in the engine code.

The implementation of the Inheritree functionality of changing its display character, dropping fruits depending on a drop rate and maturing was done primarily in the DroppablePlant abstract class, as you would expect these functionalities would be similar for all plants in which can drop consumables.

In terms of the implementation of the fruits themselves I made a Consumable abstract class in which every item that can be consumed inherits from 9used in later requirements more so), therefore SmallFruit and LargeFruit classes inherited from this class. Consumable inherits from the engine code's Item abstract class due to the fact that consumables adapt the baseline functionality of items represented in the engine code.

In my design for this requirement the following OOP design principles were applied: extensibility, DRY, abstraction, open-closed principle, dependency inversion and single responsibility principle.

How They Were Applied & Why in This Way

- The reason the DroppablePlant abstract class was made and used for the Inheritree to inherit is because there would be a lot of common functionality between Inheritree and other plants that drop consumables, such as applying the drop rate, having a consumable it drops and dropping the actual consumable, therefore all of this code can be written in the DroppablePlant abstract class. Therefore, this code is not to be repeated in other Inheritree and other future plants that drop consumables, so the DRY principle was achieved. In addition, as plants which drop consumables are extremely easy to add and can be added without changing existing code this design decision also adheres to the extensibility and open-closed principles. This class also allows for Inheritree to handle the ticking function only, and not the dropping off its fruits and the inner functionality of that (drop rates/ maturity status etc.), therefore adhering to the single responsibility principle.

- The reason why a Spawnable interface was made/used is because there may be multiple things in the game in future that will be able to spawn things that aren't plants, therefore the Spawnable interface acts as an easy way to give these classes the ability to do this spawn functionality and groups all of these classes together for future dependency inversion applications in the future.
- The addition of the Consumable abstract class for which the large and small fruit classes inherited was added because there is likely to be many other fruits/vegetables that may be added to the game and may have the similar capabilities, such as being dropped from a plant and be consumed. Therefore, having this abstract class allows for this functionality to be written in this class and not repeated again abiding by DRY. This also allows for dependency inversion in the DroppablePlant class as we do not have to specify what exact subclass or consumable a plant is dropping, we can just set this attribute as a Consumable, therefore reducing dependencies.

Design Pros and Cons

Pros

- High level of abstraction
- Extensible code – easy to add to for future features
- When similar classes are added there will not have to be much refactoring of existing code
- Polymorphism capabilities
- Reduction of dependencies

Cons

- Spawnable for dependency inversion is not used at this point
- Multi-layer abstraction (SmallFruit -> Consumable -> Item, for example), which at times may be confusing or may cause issues in overriding methods.

Alternative Designs

An alternative design would be not to use abstraction and just make the e classes as there description, however this would result in code that is not extensible at all, and a lot of repeated code for future applications as you would have to remake very similar classes and re write these similar functionalities.

An idea I had was just to make a droppable abstract class for every class that can drop something, but this didn't end up eventuating because it seemed too generic and I didn't want to make too many assumptions in that other classes would have the same rules of dropping as plants (maturity, drop rates etc.), therefore I decided against this.