FIT2099 Justifications

Requirement 1 Justification - Tree, sprout, sapling and mature

The creation of the sprout, sapling and mature class were done through the moving of Tree from an original concrete class into an abstract class with the abstract methods. These methods are Tick(), setAge() and Spawn(). This is to provide abstraction into the class forcing the 3 children classes to implement these abstracted methods whereby the tick method controls the action of the sprout/sapling/mature and the setAge to provide some protection to the private variables to not accept invalid inputs. Spawn is used as all child classes spawn in an item or enemy. This added layer of abstraction with the Tree allows for future implementation of different classes which follow the same rules of Tree. This aligns with the Open-Close principle where Objects and entities are open for extension without having to directly modify the parent class itself for different types of objects. The age of each different type of Tree is dealt with separately within their own class which reduces coupling between the different stages of Tree in contrast with other methods such as keeping the age constant through the different growth types. A Utils class was created to handle the randomness by creating a random number from 0 to 100 in order to avoid repeats of Math.Random calculations, this allows us to calculate our percentage calculations through the whole file.

Requirement 2 Justification - Mario Jump

Mario jump extends the action class. Unfortunately, this class was unable to be implemented due to the deadlines of the assignment. However the rationale behind its creation is through inheritance from abstract methods. An Interface would be created which contains 2 methods that return success rate and damage given if failed. Each jumpable entity would then have a final variable denoting its success rate and damage. This can then be called when the interface is implemented on certain classes that desire jumpable capabilities. This allows for modularity in our code where future additions of jumpable world items (such as mario pipes) can be added.

Requirement 3 Justification – Enemies

This activity required the extension and creation of two enemies in the game, the koopa and goomba. Both classes inherit from their parent class Actor. The extending of this parent class allows both enemies to be able to implement their own ActionList() which is done in accordance with the DRY principle whereby the actionList() method will not have to be re-written for different enemies. This is built with future development in mind which also follows the Open-Close principle.

Enemies are able to implement Attack Behaviour in order to attack the Player, whilst the Player can do the same. Attack Behaviour, in relation to the enemies, has 2 big sections of code: 1 checks for any enemy that can be attacked generally, and the 2nd checks for any dormant koopas that can have their shells broken. This was implemented within Attack Action and NOT koopa, in case any future enemies are to have this same functionality.

Requirement 4 Justification - Magical items

Super Mushroom and Power Star implement the Consumables interface, which allows for future items to also implement from this interface. A Consumable Action inherits from Action, which navigates the way these magical items can be consumed by the Player.

All functionality was placed within the Items themselves rather than the Player, as once the game is expanded, other actors may also be able to use the items. This minimises the amount of code that will need to be written later on.

Some of the magical items' implementation is written within Attack Action, as the attack causes certain changes in the way these items work. This implementation allows all future actors to deal with these changes.

Enums were used for both of the items in order to implement their functionality.

Requirement 5 Justification - Trading

In this segment an array list of Tradable Items is implemented. This singular array list is then instantiated only once which follows the properties of a singleton class. This is done in order to minimise dependencies between the different classes whereby a single list for all items is used and called upon in the different classes in contrast to implementing a list into each Actor capable of trading. This also opens the floor to further implementation of future items following the open close principle where the list will not have to be changed in all future vendors in the event of an item addition. The items that are added to this array list implement the TradeableItem interface which provides an empty method which gets the price of the item for trading, and a default method that allows the item to be added to the array list. While Toad is being created, an instance of each item is added to Toad's inventory so that they can be sold. Toad is also given a TradeAction instance in his action list if the actor provided to getAllowableActions has the status CAN_TRADE from the enum status. The wallet class also uses the singleton instance to keep its value throughout the game as it is called. There is also a method to get the value to check if the player has the correct funds and a method to increase the value when a coin is picked up as well as a method to decrease the value when an item is purchased. The coin was given a new pick up method that did not add the item to the inventory. This allowed for the item to still be collected by the player but instead of having the coin sit in the inventory, the value of the coin would be added to the player's wallet as stated above.

Requirement 6 Justification - Monologue

The SpeakAction class was created as a way to give Toad this action if the player it is interacting with has the status CAN_SPEAK from the enum. The SpeakAction class uses switch statements to return Toad's dialogues with a randomly generated number which will get the value of a certain line. This random value is generated from the Random function in Java. If the player has a wrench the random value will continue to generate until the value is not the one that represents the wrench line and the same goes for if the player has a power star active.

Requirement 7 Justification - Reset Game

Each class that was required to be reset (Coin, Player, All tree classes, enemies) was made to implement the functions from the resettable interface. In the constructors of these classes the instance was registered using the default method from resettable which added the class to the list of resettable items in the reset manager. The resetInstance methods were implemented based on the requirements (partly, due to lack of time to complete).

Future Improvements

Due to time constraints, not all ideas that were generated were realised in our code which lead to certain requirements being unfulfilled. However our implementation of the SOLID principles and decoupling our classes, allowed the game to run without certain required classes being met. Improvements could be made to the Tree class where it contains a full list of abstract methods. Jumpable could potentially be implemented better with better methods which could potentially be the reason that the jump requirement was not fully implemented. Some of the functional from magical items was also unable to be implemented due to time constraints such as its integration with jump and certain interactions with status effects.