FIT2099 Assignment 1 Design Rationale

- Linden Beaumont, Ricky Zhang & Rebekah Fullard

To help us understand how your system will work, you must also write a design rationale to explain your choices. You must demonstrate both how your proposed system will work and why you chose to do it that way. You may consider using the pros and cons of the design to justify your argument.

REQ1

How it works:

The tree class extends the ground class while the tree class itself is an abstract class. There are three child classes of tree; sprout, sapling, and mature. Each of these child classes have a dependency with another class in accordance with game functionality. Sprout has a dependency with the goomba class, sapling has a dependency with the coin class and mature has a dependency with koopa.

Tree extends the ground class because it requires methods from the ground class (such as tick).
The three classes that extend tree (sprout, sapling, and mature) work in a similar manner, each of the child classes needs some base methods from tree but require their own methods to carry out game functionality. Sprout will be able to create a goomba object using the goomba's constructor, similarly, sapling and mature will also be able to create the coin and koopa objects. Mature has functionality where it will be replaced by a dirt object randomly so mature will be able to also create a dirt object to replace itself at its position.

(HAVE TO UPDATE UML: REPLACE ACTOR CLASS WITH LOCATION)
The sprout and mature classes also have a dependency on the locations class so that they may use the containsAnActor(); method in locations to determine if an actor is on the same location as the sprout/mature, and if there is, don't spawn a goomba/koopa.

Why it works that way:

REQ2

How it works:

The JumpAction has associations with all the classes that it is able to interact (jump) with.

Why it works that way:

REQ3

How it works:

Goomba's base HP value will be changed so that it will always start with 20HP. Goomba objects will override the getIntrinsicWeapon method so that the damage is 10 and the verb is "kicks”. Each time a new turn ends, a method from Goomba will run that will have a 10% chance to destroy the Goomba and remove it from the map.

The constructor for Koopa will set its HP value to 100. Koopa will have a boolean attribute that represents whether the Koopa is dormant or not. The execute() method in AttackActions will be updated to check whether the target being attacked is a Koopa.

For implementing the ability of dormant Koopas to only be hurt with a wrench, when a Koopa is attacked, before trying to hurt the target the code will check if the target can be hurt with the weapon being held. If not, it will display a message saying that the attack was ineffective. If yes, the addItem() method for that location will be called and pass SuperMushroom as the item to add to that location and the Koopa will be removed.

Why it works that way:

Liskov Substitution Principle: the Koopas and the Goombas can completely replace the Actor class the inherent from

Dependency Inversion Principle: By having an abstract Actor class that both Goombas and Koopas are children of, it would be very easy to add another enemy in the future.

The actions, actors and behaviours in ‘game’ have their own subpackage like they do in ‘engine’. This is a good practice as it segments the program into objects and makes the purpose of each individual class clearer.

REQ4

How it works:

UseItemAction will implement a method that calls methods within the item that sets up what occurs when the item is consumed.

UseItemAction will have an attribute for the Actor who is using the item, and will be set up so that the effects of using the item are applied to the actor using these methods.

In the AttackAction class the execute() method - when a target is hit - will check if the Actor is under the effects of an item (stored in a boolean value to represent whether the actor is under the effects of an item). If yes, it will call a method from Actor that resets attributes back to how they were before the item was used.

For implementing the invincibility, when a target is attacked, before trying to hurt the target the code will check if the target can be hurt with the weapon being held. Since the target cannot be hurt regardless of the weapon it will return False. Only if the target can be hurt then the code will hurt the target.

Why it works that way:

Open-Closed Principle: the items extend the abstract item class. The abstract class demonstrates this principle as it allows many classes to extend it.

Similarly to question 3. The items and actions in ‘game’ have their own subpackage like they do in ‘engine’. This is a good practice as it segments the program into objects and makes the purpose of the classes clearer.

REQ5

How it works:

Coin is a subclass of Item. This class will store one attribute, and Integer ‘coinValue’. PickUpCoinAction is a subclass of Action. This class will override the execute() method from Action. The method will be implemented such that it updates an Integer attribute of Player called ‘wallet’. This attribute will track the total amount of money the Player has at any given time. The Coin will not be stored in the Player’s inventory.

Toad is a subclass of Actor. Toad has an ArrayList<Item> attribute that contains each of the Items he can sell. When the Player takes the BuyAction, that action ‘targets’ Toad and the relevant Item from his list of Items that he sells is appended to the Player’s inventory.

Sapling has a Coin attribute. This is set to null when the Sapling is instantiated (indicating that the Sapling has no currently spawned Coins). With each turn, there will be a check to

see if the Coin attribute in each Sapling is null. If it is not null, a coin cannot spawn that turn. If it is null, a coin may randomly be spawned.

(THINK ABOUT AN INTERFACE< SELLABLE> THAT IS IMPLEMENTED BY SELLABLE ITEMS TO GIVE THEM A PRICE _ THIS MAY ALSO BE GOOD FOR SOLID PRINCIPLES)

Why it works that way:

(NOT SURE THIS IS THE RIGHT PRINCIPLE) Avoiding violating Liskov Substitution Principle with the class for PickUpCoinAction. Picking up a coin has a very different effect that picking up any other item (not added to the inventory, value updates a different attribute held by the player). Thus is makes sense to create a separate class because otherwise we would need to use an ifelse to check whether the item is a coin within the pickupitemaction

REQ6

How it works:

Toad has a method that draws on a method in SpeakWithToad and prints a sentence that he says. It passes the method in SpeakWithToad boolean arguments that tell the method whether the player is holding a wrench and if the PowerStar effect is active.

SpeakWithToad has a static final attribute that is an Array of Strings. Each string is one of the potential sentences that Toad may say. SpeakWithToad has a method that randomly selects a sentence for toad to say and returns it. This method applies checks on the boolean arguments passed to it to know whether to disregard the relevant sentences when choosing. speakwithtoad gets the player actor passed to it. it can use this to access the getInventory method to check for wrench and check the capabilities list of the player to see whether the player is under the effects of the power star

toad implements canspeak. this is so that the program can
be added to in the future if more actors with the ability to speak are added

ConverseAction is a subclass of Action. This class has a ‘target’, which at this stage will only ever be Toad. Toad will implement the CanSpeak interface. It will provide the implementation the method speak(), which will define the process to determining which of Toad’s lines is randomly said when the Player takes the ConverseAction.

Why it works that way:

CanSpeak – so that if other actors that can speak are added they can also implement this method. Following the DRY principle so future classes that can speak will not rewrite identical methods to that in Toad

REQ7

How it works:

When the reset action is taken by the player, the run method in the ResetManager is called. This
method will traverse through the list of resettable instances and perform the actions detailed in REQ7.
After each single instance is reset, the cleanUp method will run to remove that instance from the list so that it is no longer resettable.
The ResetManager class will have a boolean atttribute that acts as a flag for whether the reset action
has been taken yet or not.

The World class will have an attribute of type ResetManager. Each time the processActorTurn
method is run, it will check the flag attribute in the ResetManager, and will only add the ResetAction
action to the ActionList if the flag reads true.

ResetAction class will set the hotkey so that the option is available in the menu.
This class will be a sublass of Action.
processActorTurn in the World class will add this action to the ActionList of the player

Why it works that way:

Having Tree, Coin, Koopa, Gooma, and Player implement Resettable follows the Open-Closed principle because it ensures that ResetManager doesn't