DESIGN RATIONALE

FIT2099

BENJAMIN CHEE-WEN RAJAN | 30170397 | braj0006
LIOW GIAN HAO | 30666910 | glio0001

The design rationale is written in a topic structure that we think best represents the features.
Existing Class or Method modifications will be listed down.
All new classes will have its attributes and methods available for reference at the UML Diagram.
Design principles are indicated in BOLD and [number] as a reference.

Design Principles Utilized

[1] Avoid excessive use of literals

[2] Don't Repeat Yourself

[3] Classes should be responsible for their own properties

[4] Reduce dependencies as much as possible

[5] Declare things in the tightest possible scope

[6] Avoid variables with hidden meanings

[7] Liskov Substitution Principle

[8] Command-query seperation

Zombie Bite

Modified methods

- **getIntrinsicWeapon() in Zombie class**
- **execute() in AttackAction class**

Added classes

- **PunchIntrinsic extends IntrinsicWeapon**
- **BiteIntrinsic extends IntrinsicWeapon**

Added attributes

- **punchProbability: int**

Instead of getIntrinsicWeapon() returning the same intrinsic weapon every time, we have written 2 new classes known as zombiePunch and zombieBite, which both extends IntrinsicWeapon. This **avoids excessive use of literals[1]** in the zombie class and enables the classes to deal with the damage and verbs of itself, adhering to design principle **Classes should be responsible for their own properties.[3]** As sub classes, constructers and methods will have the same arguments as well as return value to ensure **Liskov Substitution Principle[7]** is followed. Also through inheriting, we avoid code duplication and **do not repeat ourselves[2]**.

Method getIntrinsicWeapon() in the Zombie class will first initialize an array of IntrinsicWeapon containing a zombiePunch and zombieBite, the decision to keep it a local variable is because no other methods in the Zombie class would require it, allowing us to follow the principle **Declare things in the tightest scope possible[5]**. The probability will also be an int variable, and an if else statements combined with Random.nextInt() provides the chances which will then return either a zombiePunch or a zombieBite. The probability of punch occurring over bite is also stored as a class level attribute, this is to enable further features to influence it.

Moving on to AttackAction class. As of now, a Zombie object and a Human object will have different ways to attack. We should separate them to enable a more clear representation of an attack execution with conditional statements for Zombie and Human. However, the statements will not include the block of code that checks for target's conscious as this block of code will be run regardless of Zombie or Human, ensuring that we **do not repeat ourselves[2]**. The conditional statement for Zombie will check the weapon being used for the attack, if the weapon is of class zombieBite, it contains a different probability for miss chance. If it does not miss, heal() will be called on the current actor.

The design decision to include new classes for bite and punch is also to support flexibility. In the future, if the game is required to upgrade the punch or bite damage, it can be easily referred to by name instead of having to find them through verbs. It also enables our AttackAction to clearly differentiate between the weapons selected and execute accordingly. Creating conditional statement in AttackAction instead of creating a new ZombieAttackAction class is to **reduce dependencies as much as possible[4]**. AttackAction also contains dependency through different classes and methods, modifying the Zombie attack to use a different AttackAction will require a lot more changes and may also touch on the source code, which will then create even more dependencies and duplicated code which goes against our design principles.

Picking Up Weapons as a Zombie

Modified methods

- **playTurn() in Zombie class**

Added attributes

- **behaviours[] array added new PickUpBehaviour in Zombie class**

Added classes

- **PickUpBehaviour implements Behaviour**

For a zombie to know how to pickup a weapon, it must first have a behaviour for it. A new PickUpBehaviour class is added as **classes should be responsible for their own properties[3]**. Following the **Liskov Substituion Principle[7],** the new class will also implement its own getAction() to ensure consistency between subclasses. The getAction() in the new class will get the actor as well as the map and find the location of the actor through locationOf(). The location will then call getItems() that returns a list of items in the location. The zombie is dumb, so it will select a random item regardless of damage. The getAction() will return the action through item.getPickUpAction(). If there are no items, null is returned.

The Zombie will also prioritize picking up items, this means that the new PickUpBehaviour will be added onto the behaviours array at the first position to ensure that the zombie will always look for items first. PickUpBehaviour should also include a method that modify getAction() to always return null, this method will prove useful in the following tasks.

By including all the functions within PickUpBehaviour, we ensure **that dependencies are reduced[4]** and **features that belong to a class stays in a class[3]**. Zombie keeps its simplicity by only calling PickUpBehaviour without having to worry the implementation of it.

Zombie Speech

Added attributes

- **zombieSpeech: String**

Modified methods

- **playTurn()**

As this is a game ran entirely on the console, we keep it simple for now by printing the zombieSpeech at the start of the playTurn() method call. The zombieSpeech is an attribute instead of a local variable is because it **avoids excessive use of literals[1]**, and would be easier to change it later on, it does not contradict principle **Declare everything in the tightest scope[5]** because the zombieSpeech is of String object, it **does not add complex dependencies[4]** to the class and therefore can be kept as an attribute.

Zombie Limbs

Modified methods
- Constructor of Zombie class
- execute() of AttackAction class

Modified attributes
- behaviours[] array

Added classes
- Limbs (methods and attributes shown in UML class diagram)

Added methods
- loseAnArm() & loseALeg() in Zombie class
- armExists() & legsExists() in Zombie class
- movementTracker() in Zombie class

Added attributes
- zombieArm: Limbs
- zombieLeg: Limbs
- canMove: boolean

New Limb class is created to represent arms and legs of the Zombie class. Attributes of zombieArm and zombieLegs are both of class Limb, this is because the features do not differ much, allowing us to **avoid repeating codes[2].** In addition, the use of a proper Limb class and attributes for arms and legs will **avoid having variables with hidden meanings[6]** if we were to use a simple int to count the limbs instead. The methods loseAnArm(), loseALeg(), armExists(), legsExists() will all call their counterpart in the Limb class* as well as additional code which will be explained further. The Limb class contains the guardian code to ensure limbs does not drop to negative, it is in the Limb class because **classes should be responsible for maintaining their own properties[3]**.

In previous explanations, AttackAction has been modified to be conditioned towards a Zombie or Human attacking. If a Human performs a successful hurt() and if target is alive, it will have a probability of 25% to lose a limb, where losing a limb will have a 50-50 chance to call loseAnArm() or loseALeg() on the Zombie. A condition to check whether there are limbs left to lose should also be in place to ensure there are no unnecessary method calls.

When loseAnArm() is called, it will reduce a limb count from zombieArm. In addition to that, it will also alter the punchProbability stated in the Zombie Bite section, where if there is still an arm, the probability is halved and if arms don't exist, the probability becomes 0. Relating to the topic of zombie picking up items, losing both arms will nullify the PickUpBehaviour of a zombie class as stated before. Back to AttackAction, the condition where loseAnArm() executes will also call upon the inventory of the Zombie with getInventory(). If the inventory is not empty, the Zombie will have the first item removed with removeItemFromInventory() and DropItemAction() executed at the location. The reason this feature is implemented in the AttackAction, is to **reduce dependencies[4]** from Zombie to GameMap, the Zombie class itself does not contain methods that directly depend on the map or location except in playTurn(), while AttackAction already contains the information needed, making it unnecessary to add more complexity to the Zombie class. It also adheres to the principle **command query**

separation principle[8], where the loseAnArm() method does not need to drop item, lose a limb or drop an arm.

When loseALeg() is called, it will reduce a limb count from zombieLeg. Adding on to that, a new movementTracker() method and canMove attribute is created. The movementTracker will contain 3 conditions: if both legs still exists, it sets canMove into true ; else if there are no legs, set canMove into false ; else if there is only one leg, canMove = !canMove, which flips the Boolean of the attribute. The method movementTracker() will be called at the very start of playTurn(). At the behaviour loop, before checking if an action is not null, we first set a condition where if canMove is false and the behaviour's action is an instance of MoveActorAction, it will skip the iteration. This implementation **does not use excessive literals[1]** where we break the loop at certain indexes, and provides flexibility to future implementation of behaviours.

Limbs As Items And Weapons

Modified methods

- **execute() of AttackAction**

Added classes

- **simpleClub extends WeaponItem**
- **armClub & legClub extends simpleClub**
- **zombieClub extends armClub**
- **zombieMace extends legClub**
- **upgradeClubAction extends Action**

Added methods

- **dropArm() & dropLeg() in Zombie class**

The purpose of choosing to have a superclass of simpleClub and two separate clubs for arms and legs, is **to not repeat code[2]**, as the main functionality of armClub and legClub will be near identical. However, separating them gives us the ability to differentiate whether the club came from a zombie arm or a leg, which **avoid uses of literals[1]** if we were to use "keys" in simpleClub to differ between the two. Other new classes such as zombieClub and zombieMace are subclasses to armClub and legClub as they are upgrades of one another, new classes were made to ensure flexibility in the future where features requires us to differentiate between the two. Method dropArm() and dropLeg() returns a armClub and legClub.

Dropping a limb will be done in execute() of AttackAction. The justification is to ensure everything is **done in the tightest possible scope[5],** and the action only occurs when the zombies are being attacked by humans. The zombie is required to call dropArm() or dropLeg() because the damage and verbs of the new club will be stored in the methods, the reason behind this is it confusing if AttackAction was in charge of determining the damage and verb of a weapon from a Zombie class. Doing so ensures that classes are still **in charge of their own properties[3]**. When a new club is returned, from execute() we can get the location of the Zombie being attacked and from there we are able to call gameMap which will allow us to place the club on the ground, allowing players to pick it up.

A new action called upgradeClubAction is created in extension of Action class, it inherits to **reduce code duplication[2].** The execute() method is overridden. It will still take the same arguments and returns the same object type to ensure **Liskov Substituion Principle[7]** is followed. In execute(), the actor will remove the current simpleClub item and a new zombieClub or zombieMace is created and added into the inventory, damage will be calculated based on multiplication of the simpleClub's damage. The simple club class will also include a upgradeClub() method that is similar to getPickUpAction(), but will execute and return the newly added upgradeClubAction instead.

Rising from the dead

Modified methods

- **execute() of AttackAction**

Added classes

- **HumanCorpse extends Actor**

A human corpse class is created to represent a human corpse, the naming choice is to differentiate between a zombie corpse, which does not have the ability to come back to life. A HumanCorpse is an extension of actor, but it is unable to perform any action, thus the playturn() method is overridden to return DoNothingAction.

In AttackAction, there will be 2 attack conditions, one for human attacks and one for zombie attacks (explained in Zombie Bite above). When a zombie attacks a human and the human dies (isConcious() is false), after removeActor(), addActor() is called and creates a new HumanCorpse object at the same location.

Back to HumanCorpse, it will contain attributes that saves int values representing minimum and maximum turns taken to respawn, also it will contain an attribute to keep track of turns. They are all contained in the HumanCorpse because it is a good design principle to have **classes be in charge of their own properties[3]** and the turns taken to respawn are kept as attributes to **avoid excessive use of literals[1].** In playturn(), when the counter is within the range or it is at maximum, it will remove the HumanCorpse actor from the map and add a Zombie actor into the same location. The majority of the feature runs in playTurn() because it already has information on the location, which **reduces dependencies[4]** as opposed to creating methods that needs to also depend on the location of an actor.

Crops

Added classes

- **Crop extends Ground**

Crop class is an extension to ground as the feature implementation of crops will mostly work as a ground object with the exception of a few overridden methods or new methods. With this, we **minimize code repetition[2]** and also overridden methods will not change input arguments and return values to adhere to **Liskov Substitution Principle[7]**.

The crop class will have display char of '$'. It contains two attribute ripeTurns of int that represents how many turns a crop takes to ripen and turnsTracker that represents the amount of turns remaining. A method is also written that specializes in decrementing the turnsTracker, this contains the guardian code and ensures turnsTracker doesn't go over limits or does weird stuff. The crop also contains a fertilize method, which decrements the turnsTracker by 10 through decrement method. The amount of turn reduced through fertilizing is declared in the method instead of class level because the variable isn't utilized anywhere else, allowing us to **declare things in the tightest possible scope[5].** Crop class also contains method isRipe() that returns true if turn tracker is 0.

Crop class will utilize the inherited method tick(), which is overridden to include the method to decrement turnsTracker.

Farmers and food

Modified attributes

- **behaviours[] array in Human class**

Added classes

- **Farmer extends Human**
- **Food extends PortableItem**
- **FarmingBehaviour extends Behaviours**
- **ConsumeBehaviour extends Behaviours**
- **SowAction extends Action**
- **HarvestAction extends Action**
- **FertilizeAction extends Action**
- **ConsumeAction extends Action**

Farmer class is created as a sub class of Human as a farmer is a more advanced Human in the game. Much like zombies, the farmer will contain new behaviours that defines what it does every turn, which are FarmingBehaviour and ConsumeBehaviour, allowing **the Farmer class to be responsible for how it functions[3]**. ConsumeBehaviour will also be implemented in the Human class. Much like any non-player actor, the farmer will also override playTurn() that iterates through behaviour array. This ensures that methods in **subclasses functions similarly as its superclass[7]**. The Food class extends a PortableItem as it can be picked up in the game, represented by a "*" character.

For actions, the SowAction, HarvestAction, FertilizeAction works in a similar fashion which it overrides execute() and contains a constructor with Ground object (or crop, as it is a subclass). For SowAction, it replaces a Dirt object with a Crop object as its execute(). FertilizeAction will call the fertilize method of the Crop class. HarvestAction will remove the Crop object from the map and replace it with a new Food object. ConsumeAction will instead call upon a Food object in the actor's inventory and call heal() on the actor, followed by removing the item from their inventory. All action subclasses will have their menu description overwritten as well.

FarmingBehaviour is exclusively a Farmer class behaviour. In playTurn() of Farmer class, it iterates through FarmingBehaviour, ConsumeBehaviour and WanderBehaviour. FarmingBehaviour will override getAction(), where it will iterate through the Exit of the Farmer, if there is Crop object, it proceeds to check isRipe(), which it will then decide between HarvestAction or FertilizeAction. If no Crop exists, it proceeds with SowAction, however this will be conditioned with a probability of 33%. The ConsumeBehaviour will be present for both Human and Farmers. It iterates through Exits, and if there is a Food object nearby, the object is added into the Actor's inventory and ConsumeAction is returned. For Player class, ConsumeAction is added as an allowable action in Food item, which enables them the option to consume it.

As ConsumeAction is shared between Human and all its subclasses, we **avoid duplicating code[2].** Consistency is also maintained by writing a new kind of Actor based on its super class, making it easier to read and change the features. By having a behaviour array that calls upon its actions, it prevents a method trying to **alter values and return values at the same time[8]** as now the classes will **deal with their own functions[3]**.