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 separation

Zombie Bite

Modified methods

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

Added attributes

- **punchChance: double, in Zombie class**
- **biteHeal: int, in AttackAction class**
- **biteMissChance: double, in AttackAction class**

Instead of getIntrinsicWeapon() returning the same intrinsic weapon every time, we have modified it to return either a intrinsicWeapon that represents a bite or a punch based on the probability rolled. It is contained in the Zombie class to **avoids excessive use of literals[1]** in the zombie class and enables the class to deal with the damage and verbs of itself, adhering to design principle **Classes should be responsible for their own properties[3].**

Method getIntrinsicWeapon() in the Zombie class will first create two IntrinsicWeapon of bite or punch, 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 double variable, and an if else statements combined with Random.nextDouble() provides the chances which will then return either a punch or bite. 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, which is done by checking if the attacking actor has capability of ALIVE(human) or UNDEAD(zombie). This allows us **to avoid literals[1]**, as the capability is an ENUM instead. 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]**.

To differentiate the IntrinsicWeapon used, the verb() represents a unique string that allows us to differentiate between a bite or punch. When a bite is used, the heal amount will be based on the class level attribute **to avoid literals[1]**. Also, the chance of using a bite is also stored as a class level attribute. We used rand.nextDouble() instead of rand.nextBoolean() despite being 50% anyway is to ensure flexibility later on when changes are to be made.

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**
- **speechChance: double**

Modified methods

- **playTurn()**

Instead of making a zombie speech a whole new action, we will utilize the Display class of the ZombieWorld project, which allows us to display text. This method is chosen over System.out.println, which is a terrible idea, and also creating a new speech action. An action will not make sense as the zombie should be able to attack and say “Braaains” at the same time. With Display, it will display the zombieSpeech at the start of the playTurn() method call. The zombieSpeech and speechChance 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
- hurt() of Zombie class

Modified attributes

- all behaviours now have capability of ENUM { NON_MOVEMENT, MOVEMENT }

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

- arms: Limbs
- legs: Limbs
- canMove: Boolean
- zombies now contain capability of ENUM {ARM, LEG}

New Limb class is created to represent arms and legs of the Zombie class. Attributes of arms and legs are both of class Limbs, this is because the features do not differ much, allowing us to **avoid repeating codes[2].** In addition, the use of a proper Limbs 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 our previous design rationale, AttackAction has been modified to be conditioned towards a Zombie or Human attacking. We have since changed our previous implementation, as it does not adhere to **classes being responsible for their own properties[3].** Now, losing a limb will be inside the hurt() method.

When loseAnArm() is called, it will call arms.breakALimb() and it will half the punchChance or make it 0 if armsExist() is false. It will also remove PickUpBehaviour from the behaviours. At Zombie's playturn(), if zombie has 1 drop an item and if he has no arms he drops all the items. The loseAnArm() method will also add LostLimbCapability to the zombie, which represents that the zombie has recently lost a limb. When loseALeg() is called, it will call legs.breakALimb(). 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(). When executing a behaviour, it will now check if the behaviour has capability IS_MOVEMENT, combining this with canMove, the program will be able to decide when the zombie can execute a movement behaviour. 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

- **ArmWeapon & LegWeapon extends WeaponItem**
- **zombieClub, zombieMace extends WeaponItem**
- **UpgradeItemAction extends Action**

Added methods

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

The purpose of choosing to have a ArmWeapon, LegWeapon, ZombieClub, ZombieMace extend WeaponItem, is **to not repeat code[2]**, as the main functionality is that they act as weapons. However, an ArmWeapon has capability CLUB while LegWeapon has capability MACE.

To follow up the previous section, when a zombie loses a limb, it will contain the capability LoseLimbCapability.ARM or LEG. From attack action, after hurting a target, we check whether the zombie has recently lost a limb, if it contains the capability, then it will act accordingly by dropping either an ArmWeapon or LegWeapon. When the zombie doesn't lose an arm or leg from hurt(), it will not contain the capabilities. Dropping the limb weapon is 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.

By using Capabilities, it ensures that classes are still **in charge of their own properties[3]**. A Zombie is I charge of losing their limbs and letting other methods know that it lost a limb. While AttackAction is in charge of what happens when zombie loses limbs.

A new action called UpgradeItemAction 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 weapon item and a new ZombieClub or ZombieMace is created and added into the inventory. It uses capabilities to tell if the weapon is meant to be upgraded into a club or mace or both.

Since upgrading items is exclusive to player, in the player's playTurn() method, it will check if such weapon can be upgraded, if possible, it will add the upgrade item action as an option to the player.

Rising from the dead

Modified methods

- execute() of AttackAction

Added classes

- Corpse extends PortableItem

A corpse class is created to represent the corpse, it extends PortableItem as per assignment requirements. In AttackAction, when the target is not Conscious, it will check if the target is human. Which will then create a Corpse portable item.

Back to Corpse, 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 Corpse 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 tick(), it will call canRespawn() which checks if the corpse is eligible to respawn. If true, it will call respawn(), which removes the corpse and adds a new zombie at the same location. If the item was in an inventory, it will still increase in turns, but is not allowed to respawn as that would freak people out.

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 ripeAge of int that represents how many turns a crop takes to ripen and age that represents the amount of turns remaining. The crop also contains a fertilize method, which increments the age by 10 through decrement method. The amount of turns reduced through fertilizing is declared in the method instead of class level to **avoid literals[1].** Crop class also contains method isRipe() that returns true if age is past the ripeAge().

Crop class will utilize the inherited method tick(), which is overridden to increment the age.

Farmers and food

Modified attributes

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

Added classes

- **Farmer extends Human**
- **Food extends Item**
- **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, ConsumeBehaviour and WanderBehaviour, 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 Item 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 or directly add the Food into inventory if it is the player. 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, while iterating through the inventory, if food is in the inventory, ConsumeAction is added as an option.

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]**.