

FIT2099 - Update Design Rationale for Assignment 2

Name: Isaac Lee Kian Min

StudentID: 31167462

Old Classes Modified

The words in **red** indicate a new variable/method being added into the class, whereas words in **black** indicate extending an already existing method.

Zombie
<div><div><div>+ FAILED_BITE: double</div><div>+ MAX_ARMS_AND_LEGS: int</div><div>+ TALKING_PROBABILITY: double</div><div>+ LOSE_LIMB_PROBABILITY: double</div><div>+ DROP_WEAPON_PROBABILITY: double</div><div>+ ZOMBIE_PHRASES: String[]</div><div>- biteProbability: double</div><div>- legKnockedPreviously: boolean</div><div>- zombieLimbs: Limbs</div></div><div><div>- setLimbs()</div><div>- setBehaviours(behaviours: Behaviour[])</div><div>+ getWeapon(): Weapon</div><div>+ getIntrinsicWeapon(): IntrinsicWeapon</div><div>+ playTurn(actions: Actions, action: Action, map: GameMap, display: Display): Action</div><div>- zombieTalking(display: Display)</div><div>+ knockOffLimb(map: GameMap): String</div><div>- halvePunchProbability()</div><div>- dropAWeapon(map: GameMap)</div><div>- losingArms(lostArms: int, map: GameMap)</div><div>- losingLegs(lostLegs: int)</div><div>- skipTurn(): boolean</div></div></div>

- Declare **FAILED_BITE**, **MAX_ARMS_AND_LEGS**, **TALKING_PROBABILITY**, **LOSE_LIMB_PROBABILITY**, **DROP_WEAPON_PROBABILITY** & **ZOMBIE_PHRASES** as static final constant as all Zombie objects shares the same attributes and that it helps reduce embedded literals in source code.

<i>ZombieActor</i>
typeOfZombieActor: TypeOfZombieActor # rand: Random
+ getTypeOfZombieActor(): TypeOfZombieActor + getWeapon(): Weapon

- A **typeOfZombieActor** attribute which indicates the type of ZombieActor this is (e.g. a player, zombie, etc)
- A **rand** attribute to generate random numbers used for determining the chances of a successful attack by the ZombieActor.
- A **getWeapon** override method that implements the chances of a ZombieActor missing its opponent while fighting with its preferred weapon of choice.

New Classes Added/Extended

The words in **red** indicate a new variable/method being added into the class, whereas words in **black** indicate extending an already existing variables/method.

Corpse
- deadActor: Actor - count: int - revivalCount: int
+ Corpse(actor: Actor) + tick(currentLocation: Location)

- A **Corpse** extends a **PortableItem**.
- A **count** attribute keeps track of the number of game time passed.
- A **revivalCount** attribute is assigned a value between 5-10 upon instantiation of a Corpse object.
- A **Corpse constructor** method that takes in the actor that has died.
- A **tick** override method which implements how a Corpse may be converted into a new Zombie.

DropLimbAction
+ DropLimbAction(limb: Item) + execute(actor: Actor, map: GameMap): String - random(num: int): int

- A **DropLimbAction** extends a **DroItemAction**
- An **execute** override method that drops the item specified.
- A **DropLimbAction constructor** method that takes in the limb item to be dropped.
- A **random** method is created to help determine a random location (either adjacent to the actor's current position or on it) in the execute method.

Limb
typeOfLimb: TypeOfLimb
+ Limb(typeOfLimb: TypeOfLimb, displayChar: char, portable: boolean) + getTypeOfLimb(): TypeOfLimb + makeCopy(): Limb + toString(): String

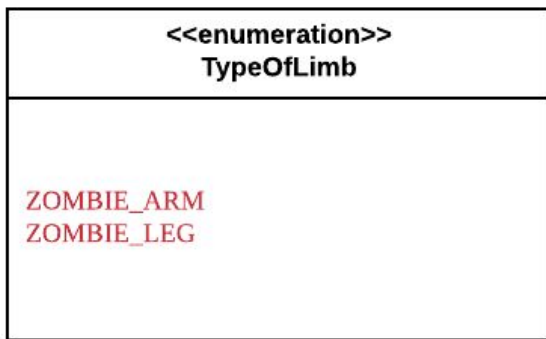
- A **typeOfLimb** attribute which stores the type of limb.
- A **getTypeOfLimb** method which returns the type of limb this object is.
- A **Limb constructor** method that takes in the type of limb it is.
- A **makeCopy** abstract method to get a copy of this Limb and referenced by this class.

Limbs
- MAX_LIMBS: int - limbs: List<Limb>
+ Limbs(maxLimbs: int) + getLimbs(): List<Limb> + addLimb(newLimb: Limb): boolean + removeLimb(aLimb: Limb): boolean + noMoreType(typeOfLimb: TypeOfLimb): boolean + count(typeOfLimb: TypeOfLimb): int + totalNumberOfLimbs(): int

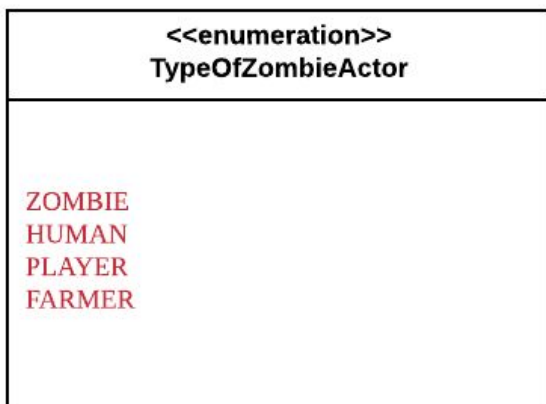
- A **MAX_LIMBS** final attribute to determine the maximum number of limbs allowed.
- A **limbs** attribute to keep a collection of the limb objects.
- A **Limbs constructor** method that takes in the maximum number of limbs.
- A **getLimbs** method to get an unmodifiable list of the collection of limb objects.
- An **addLimb** method to add a limb to the collection of limb objects.
- A **removeLimb** method to remove a specified limb from the collection of limb objects.
- A **noMoreType** method to determine whether this Limbs still has a certain type of Limb object.
- A **count** method to count the number of a certain type of limb.
- A **totalNumberOfLimbs** method to get the total number of limb objects in the collection.

PickUpItemBehaviour
- targetClass: Class<?>
+ PickUpItemBehaviour(parameter: Class<?>) + getAction(actor: Actor, map: GameMap): Action

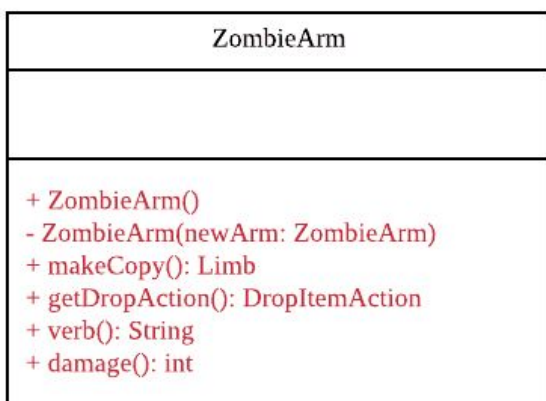
- A **PickUpItemBehaviour** implements a **Behaviour**.
- A generic **targetClass** attribute which determines the type of Item to pick up.
- A **PickUpItemBehaviour constructor** method that takes in the kind of item to be picked.
- A **getAction** override method which only picks up an Item specified by **targetClass**.



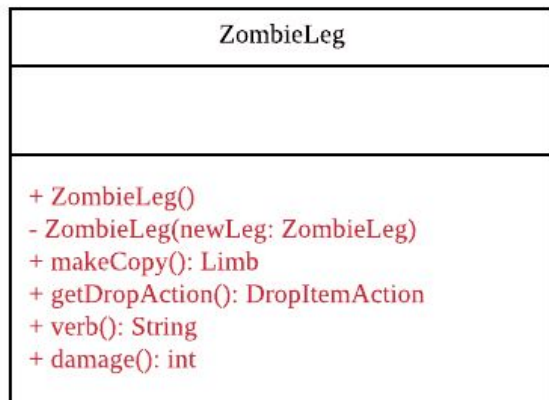
- A **TypeOfLimb** enum class that defines the type of **Limb** there is.
- Currently defines 2 constant values, **ZOMBIE_ARM** & **ZOMBIE_LEG** as types of limb.



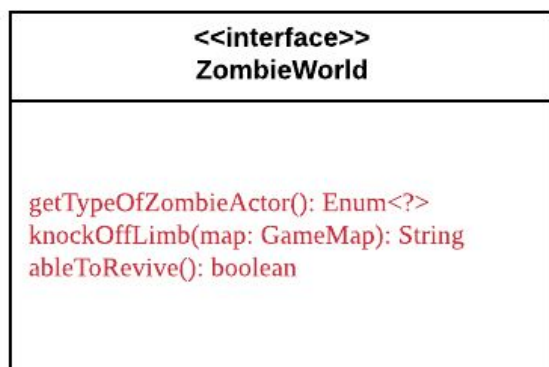
- A **TypeOfZombieActor** enum class that defines the different types of **ZombieActors** in the ZombieWorld.
- Currently defines **ZOMBIE**, **HUMAN**, **PLAYER** & **FARMER** as different types of ZombieActors in the ZombieWorld.



- A **ZombieArm** class that extends and implements a **Limb & Weapon** class respectively.
- A **ZombieArm constructor** method to create a ZombieArm object.
- A **ZombieArm copy constructor** method to make a copy of a ZombieArm object.
- A **getDropAction** override method to return a **DropLimbAction**.
- A **verb** method implemented to determine the verb used when this Limb is used as a weapon item.
- A **damage** method implemented to determine the amount of damage dealt by this Limb weapon item.



- A **ZombieLeg** class that extends and implements a **Limb & Weapon** class respectively.
- A **ZombieLeg constructor** method to create a ZombieLeg object.
- A **ZombieLeg copy constructor** method to make a copy of a ZombieLeg object.
- A **getDropAction** override method to return a **DropLimbAction**.
- A **verb** method implemented to determine the verb used when this Limb is used as a weapon item.
- A **damage** method implemented to determine the amount of damage dealt by this Limb weapon item.



- A **ZombieWorld** interface used to show what an Actor of the ZombieWorld game can do.

- A **getTypeOfZombieActor** method which returns the type of actor.
- A **knockOffLimb** default method which determines the behaviour of losing a limb if an actor has a limb.
- A **ableToRevive** default method which indicates this actor's ability to rise from the dead.

How are these classes used to implement the features described in the Assignment:

From the **Zombie Attacks** questions,

- 1) Zombies should be able to bite. Give the Zombie a bite attack as well, with a 50% probability of using this instead of their normal attack. The bite attack should have a lower chance of hitting than the punch attack, but do more damage – experiment with combinations of hit probability and damage that make the game fun and challenging. (You can experiment with the bite probability too, if you like.)
 - a) Override the `getWeapon` method in the `ZombieActor` class to also determine whether a `ZombieActor` misses while using its weapon to fight.
 - b) Extend the `getIntrinsicWeapon` method in the `Zombie` class to create a new bite `IntrinsicWeapon` and determine the chances of using it by comparing a randomly generated number with `biteProbability`.
 - c) Override the `getWeapon` method in the `Zombie` class to check for a bite `IntrinsicWeapon`. If it's a bite, determine whether its bite attack is successful by comparing with `FAILED_BITE`.
 - d) Designed it this way because:
 - i) Zombie chances of using its bite intrinsic weapon and its chances of biting successfully is encapsulated inside the `Zombie`.
 - ii) Avoid dependencies and making use of polymorphic code by overriding already existing methods to implement behaviours specific only towards a `Zombie`.
 - iii) Reusing the `IntrinsicWeapon` class to create a bite intrinsic weapon reduces duplicated codes.
- 2) A successful bite attack restores 5 health points to the `Zombie`
 - a) While checking for a successful bite in the `getWeapon` method of the `Zombie` class, if the bite was successful, reuse the `heal` method to gain hitpoints.
 - b) Designed it this way because:
 - i) Only a `Zombie` will be able to heal after a bite attack, therefore encapsulating such behaviours specific only to a `Zombie` inside its own class.
 - ii) Reusing existing methods, thus reducing duplicated codes.
- 3) If there is a weapon at the `Zombie`'s location when its turn starts, the `Zombie` should pick it up. This means that the `Zombie` will use that weapon instead of its intrinsic punch attack (e.g. it might "slash" or "hit" depending on the weapon)
 - a) Inside the `Zombie` class, add a new `PickUpItemBehaviour` object to the behaviours array. Ensure to pass the parameter `Weapon.class` during the instantiation process.

- b) For every play turn, if there's a weapon item, it will pick it up.
- c) Designed it this way because:
 - i) Makes use of already existing methods to implement the behaviour of picking up weapons through the playTurn method.
 - ii) May reuse the PickUpItemBehaviour class for other Actors to implement picking up item behaviours.
- 4) Every turn, each Zombie should have a 10% chance of saying "Braaaaains" (or something similarly Zombie-like)
 - a) Inside the Zombie class, create a new method zombieTalking which determines the chances of saying something by comparing a randomly generated number with TALKING_PROBABILITY. If successful, it will randomly say a phrase specified in ZOMBIE_PHRASES.
 - b) Designed it this way because:
 - i) Only Zombies are able to say zombie-like phrases, hence encapsulating this property unique to a Zombie inside of the Zombie class.

From the **Beating Up Zombies** questions,

- 1) Any attack on a Zombie that causes damage has a chance to knock at least one of its limbs off (I suggest 25% but feel free to experiment with the probabilities to make it more fun)
 - a) Create a new ZombieWorld interface with a default knockOffLimb method which returns null since not all ZombieActors have limbs and are able to have its limbs knocked off.
 - b) Override the knockOffLimb in the Zombie class to determine the chances of knocking off a Zombie limb by comparing a randomly generated number with LOSE_LIMB_PROBABILITY.
 - c) If successful, the Zombie gets a copy of all the limbs it has and randomly removes any limbs by generating a random number to specify the number of limbs to be knocked off.
 - d) Designed it this way because:
 - i) To avoid downcasting of data types by making a knockOffLimb interface and make use of polymorphism.
 - ii) This behaviour is again unique only to Zombies, hence override the knockOffLimb method to implement its own behaviours of having its limbs knocked off. This helps reduce dependencies.
- 2) On creation, a Zombie has two arms and two legs. It cannot lose more than these.
 - a) Add a new Limbs attribute, zombieLimbs, to the Zombie class that models a Zombie who has a collection of limbs.
 - b) In the Zombie class, create a new method setLimbs which adds ZombieArm & ZombieLeg objects, within the range of MAX_ARMS_AND_LEGS, which equals 2.
 - c) Every time the Zombie knockOffLimb method is called, it checks first whether the Zombie still has limbs.
 - d) Designed it this way because:

- i) Limbs class can be reused for other Actors (in case asked to implement) which can help reduce duplicate codes in the future.
 - ii) Have the setLimbs method declared private to make sure this operation can only be performed by the Zombie (because Zombie's only have 2 arms and 2 legs)
- 3) If a Zombie loses one arm, its probability of punching (rather than biting) is halved and it has a 50% chance of dropping any weapon it is holding. If it loses both arms, it definitely drops any weapon it was holding.
 - a) In the Zombie class, create a private method losingArms which helps implement the consequences of a Zombie losing 1 arm or both arms.
 - b) Then, we also create a method dropAWeapon which helps drop the first occurrence of a weapon item in the Zombie's inventory.
 - c) Next, we create a method halvePunchProbability which reduces the probability of using the Zombie's punch intrinsic weapon by increasing the biteProbability variable.
 - d) If lost 1 arm, calls halvePunchProbability while comparing a random generated number to DROP_WEAPON_PROBABILITY before calling dropAWeapon.
 - e) If it loses both arms, it resets the behaviours attribute through the setBehaviours method to remove the PickUpItemBehaviour of a Zombie.
 - f) In the Zombie class, after successfully knocking a Limb in knockOffLimb, it calls the losingArms method to implement the consequences.
 - g) Designed it this way because:
 - i) The consequences as a result of a Zombie losing an arm should be encapsulated within the Zombie class (i.e. this property/behaviour is unique to a Zombie only).
 - ii) Created several methods to help modularise code so that methods don't become overly huge and easily maintainable.
- 4) If it loses one leg, its movement speed is halved – that is, it can only move every second turn, although it can still perform other actions such as biting and punching (assuming it's still got at least one arm)
 - a) In the Zombie class, create a private method losingLegs which helps implement the consequences of a Zombie losing 1 leg or both legs.
 - b) Then, we create a new attribute legKnockedPreviously to keep track of whether the Zombie had its leg knocked off by another Actor during an attack.
 - c) Next, we create a method skipTurn that depending on the state of legKnockedPreviously, determines whether to skip a turn.
 - d) If a Zombie loses a leg, it sets the legKnockedPreviously attribute to a value of true, which will be used in the skipTurn method to determine whether to skip a turn.
 - e) In the Zombie class, after successfully knocking a Limb in knockOffLimb, it calls the losingLegs method to implement the consequences.
 - f) In the playTurn method of the Zombie class, there is a call to the skipTurn method to determine whether to skip a turn by returning a DoNothingAction.
 - g) Designed it this way because:

- i) Again, the consequences as a result of a Zombie losing a leg should be encapsulated within the Zombie class (i.e. this property/behaviour is unique to a Zombie only).
 - ii) Makes use of existing methods to skip a turn while creating new methods to help implement the behaviour.
 - iii) Created several methods to help modularise code so that methods don't become overly huge and easily maintainable.
- 5) If it loses both legs, it cannot move at all, although it can still bite and punch
 - a) Inside of the losingLegs method of the Zombie class, if it loses both legs, it resets the behaviours attribute (similar to how loseArms method did as well) through the setBehaviours method, which will remove the Zombie's ability to Wander and Hunt.
 - b) Designed it this way because:
 - i) To keep things simple
 - ii) Make use of existing methods to implement this behaviour, reducing duplicated codes.
- 6) Lost limbs drop to the ground, either at the Zombie's location or at an adjacent location (whichever you feel is more fun and interesting)
 - a) A ZombieArm & ZombieLeg both extends a Limb and implements a Weapon. Hence we override the getDropAction method of the Item abstract class for both classes to return a DropLimbAction (which will determine a random location adjacent to the Zombie or the current location) to drop the ZombieArm or ZombieLeg.
 - b) Inside the knockOffLimb method of the Zombie class, while removing a limb from limbs, we will get a drop action by calling the getDropAction method and execute the action to drop the limb.
 - c) Designed it this way because:
 - i) Avoid downcasting by overriding existing methods to make use of polymorphic code, which can also help reduce unnecessary dependencies.
 - ii) The DropLimbAction extends a DropItemAction, and this helps encapsulate its property of dropping a limb at a random location within it. This also prevents duplicated code by inheriting already existing ones.
- 7) Cast-off Zombie limbs can be wielded as simple clubs – you decide on the amount of damage they can do
 - a) Since both ZombieArm & ZombieLeg objects are Items and Weapons, we are able to pick up these objects using the getPickUpAction of the Item class.
 - b) Since both ZombieArm & ZombieLeg implement Weapon, they both have unique damages when using it.
 - c) Designed it this way because:
 - i) Make use existing methods from the Item class, hence reducing duplicated codes.
 - ii) Only ZombieArm & ZombieLeg can be used as weapons, hence the reason both classes implement Weapon. This makes ZombieArm and ZombieLeg special limbs with its own attributes and capabilities.

From the **Rising from the Dead** questions,

- 1) As everybody knows, if you're killed by a Zombie, you become a Zombie yourself. After a Human is killed, and its corpse should rise from the dead as a Zombie 5-10 turns later.

- a) Create a new Corpse class that extends a PortableItem (since a corpse was previously a PortableItem).
- b) In the Corpse class, we override the tick method to implement the behaviour of having a killed Human Actor become a Zombie after 5-10 turns by creating a new Zombie.
- c) Upon instantiation of a new Corpse, a count and revivalCount variables are initialised to 0 and a random number between 5-10 respectively, to help keep track of the game time that has passed by.
- d) If a Corpse is in an Actor's inventory or when an Actor is standing on a location with a Corpse, it will create a new Zombie at a location adjacent to its current location.
- e) Determines whether an Actor can be revived if the ableToRevive method of the ZombieWorld interface and that this actor is of type human.
- f) Designed it this way because:
 - i) We can then encapsulate the behaviours of a corpse within its own class. This will help maintainability of the class.
 - ii) We override methods to make use of polymorphic code to prevent downcasting and reduce dependencies.