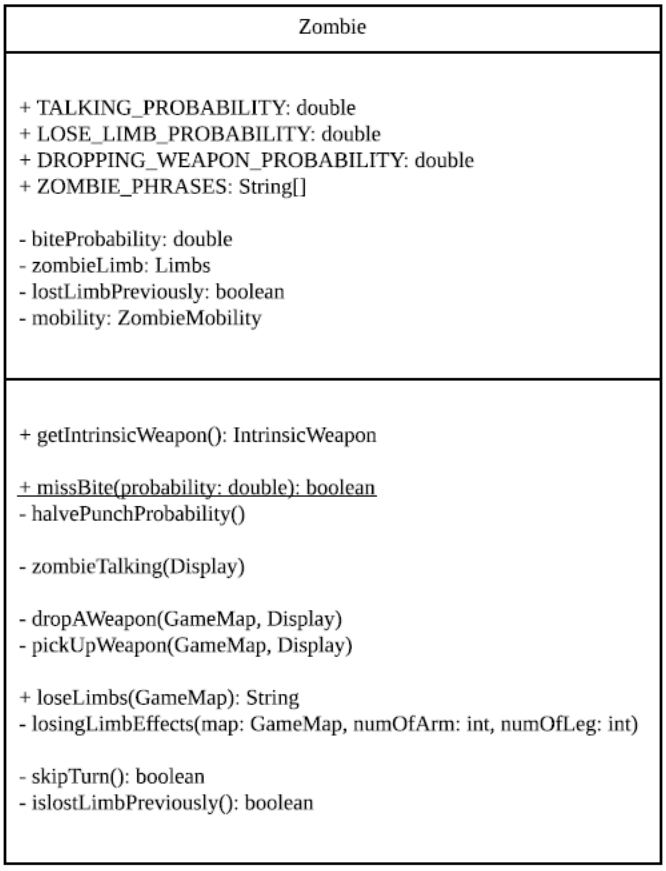
FIT2099 Assignment 1 – Design Rationale

Name: Isaac Lee Kian Min

StudentID: 31167462

Modification to Existing Classes

Zombie Class

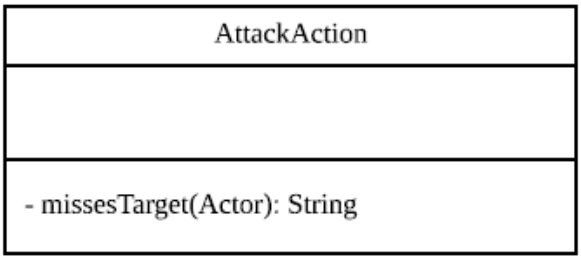


1) Static Fields/Variables
    a. TALKING_PROBABILITY – this variable helps determine the chances of a Zombie saying a word (e.g. “Braaaaains”). This helps reduce loosely embedded literals by making it a static constant (i.e. all Zombies should/would have the same probability to say something).
    b. LOSE_LIMB_PROBABILITY – this variable helps determine the chances of a Zombie losing at least a limb after being attacked by another Actor. This helps reduce loosely embedded literals by making it a static constant (i.e. all Zombies should/would have the same probability of losing a limb).

    c. DROPPING_WEAPON_PROBABILITY – this variable helps determine the chances of a Zombie dropping a weapon after it lost an Arm after an attack by another Actor. This helps reduce loosely embedded literals by making it a static constant (i.e. all Zombies should/would have the same probability of losing a limb).
    d. ZOMBIE_PHRASES – this variable stores an array of words (i.e. Strings) which determines the kinds of phrases a Zombie may say (e.g. "Braaaaains"). This helps reduce loosely embedded literals by storing the possible list of words inside an array.
2) Non-static Fields/Variables
    a. biteProbability – this variable helps determine the chances of a Zombie using its bite intrinsic weapon. This a not a static constant as the biteProbability may change based on the number of arms lost at a given point in gameplay (i.e. each Zombie has a unique biteProbability even though it starts out with the same probability when the game just started).
    b. zombieLimb – this variable stores a collection of limbs that a Zombie has, be it an Arm or a Leg (further elaborated in the Limbs class).
    c. lostLimbPreviously – this variable keeps track of whether a Zombie lost any of its Limbs as a result of an attack on it by another Actor. True if it lost any Limbs, otherwise False.
    d. mobility – this variable stores the current state of mobility of a Zombie based on the number of Legs it still has.
3) Static Methods
    a. missBite – this method is used to determine whether a Zombie missed its target while using its "bite" intrinsic weapon. Declared it a static method because all Zombie has the same mechanisms of missing a bite and that it's a property of all Zombies. This method is unique to Zombies so that's why its created inside this class.
4) Non-static Methods
    a. getIntrinsicWeapon – propose to modify this method so that it creates another IntrinsicWeapon that mimics the "bite" weapon option besides the "punching" weapon option. This helps reduce repeated codes and to keeps it simple by reusing an existing class (in this case, IntrinsicWeapon).
    b. halvePunchProbabililty – a new method which has the sole purpose of halving a Zombie's chances of "punching" after losing an Arm by manipulating the biteProbability field.
    c. zombieTalking – a new method which allows a Zombie to say phrases, which are listed in the ZOMBIE_PHRASES field. Selects a phrase from this field randomly.
    d. dropAWeapon – a new method which allows a Zombie to drop the first occurrence of a Weapon in its inventory. For now, dropAWeapon is placed in the Zombie class since a Human is unable to attack (with the exception of Player). In the event where all ZombieActor are able to attack, we may abstract this method in the ActorInterface interface and implement this method within the ZombieActor class to help reduce duplicated codes and make use of polymorphic code.
    e. pickUpWeapon – a new method which allows a Zombie to only pick up a weapon item at its current location. Similar to the dropAWeapon method, we may abstract this method in ActorInterface interface and implement this method within the ZombieActor class should all Actors in this game have the ability to pick up weapon item.

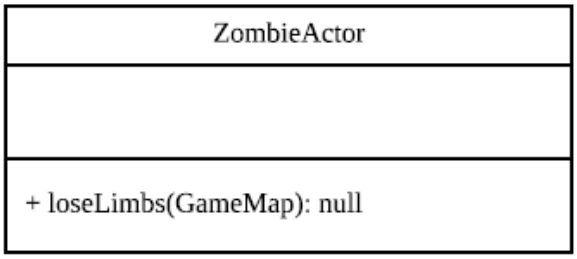
f. loseLimbs – a new method which carries out the process of a Zombie losing limbs after being attacked by another Actor. Overrides the main implementation in ZombieActor, where ZombieActor implements the newly abstracted method in ActorInterface interface (more on this in ActorInterface interface).
g. losingLimbEffects – a new method which implements the consequences of a Zombie losing a Limb (e.g. if lose both Legs, Zombie unable to move).
h. skipTurn – a new method which skips the Zombie's current play turn, equivalent to a DoNothingAction in this game.
i. isLostLimbPreviously – a new method which determines whether a Zombie lost a Limb previously (in specific, a Leg) as a result of being attacked.

AttackAction Class



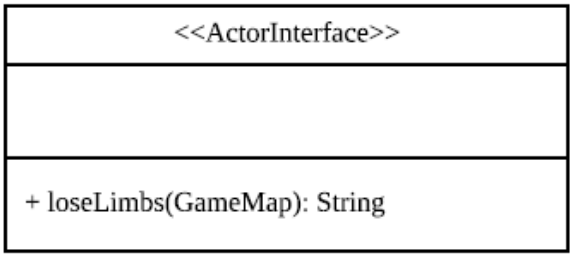
Added a missesTarget method that returns the description of an actor missing its target. This is to help reduce duplicated code, which may arise.

ZombieActor Class



Added a loseLimbs method which implements the abstract method defined in the ActorInterface. Implements it to return a null value as not all ZombieActor in the game has Limbs currently.

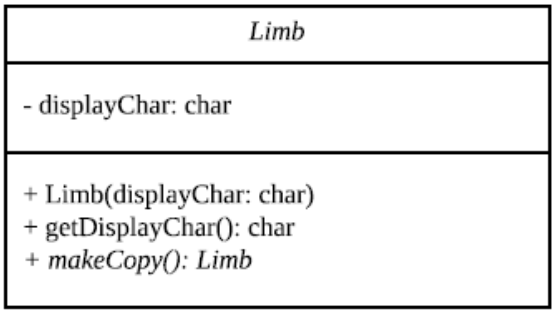
ActorInterface Interface



Added an abstract loseLimbs method that needs to be implemented by any class who implements this interface. Declared loseLimbs method inside here to prevent downcasting an Actor and take advantage of polymorphism. If an Actor has limbs, the loseLimbs method defines the consequences/effects of losing a Limb to that Actor.

New Classes

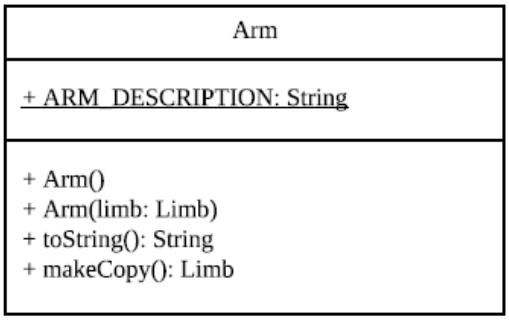
Limb

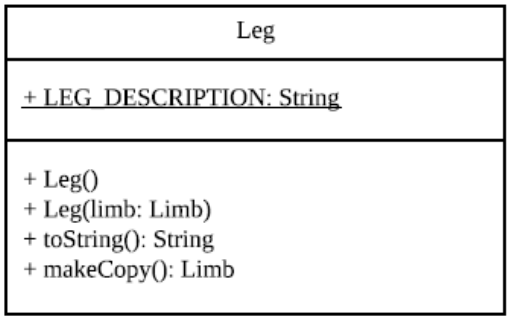


This is an abstract class which serves as a base to a Limb. We declare this class abstract as a Limb has certain attributes that all limbs have, which helps reduce duplicated codes.

1) Variables/Fields
    a. displayChar – this variable gives a character representation of a Limb.
2) Methods
    a. Limb – the constructor of a Limb (however we are not to instantiate any objects of Limb)
    b. getDisplayChar – a new method which returns the character representation of this Limb.
    c. makeCopy – a new abstract method which up-casts' the reference variable to this class for any objects that are a Limb.

Arm Class & Leg Class

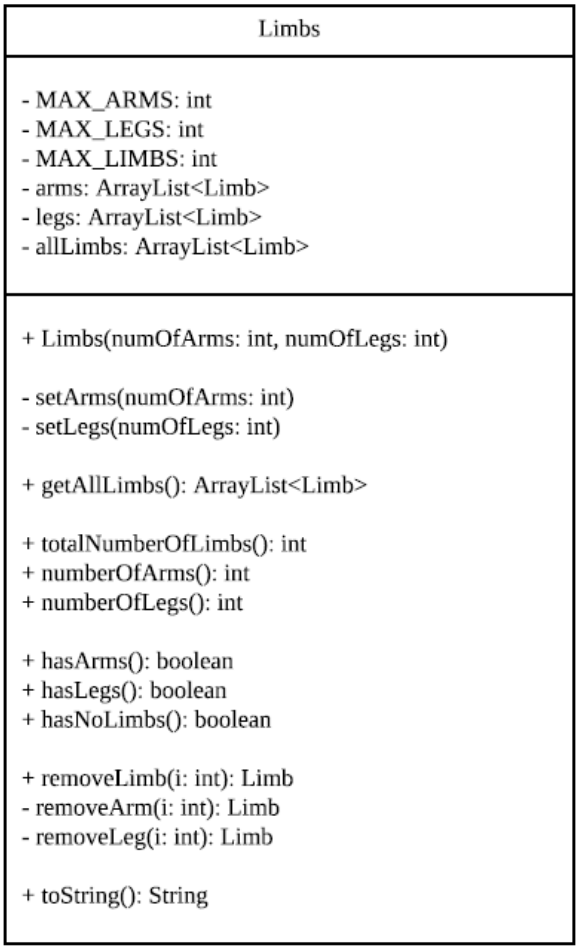




Arms & Legs are both a Limb. This 2 classes inherits (i.e. extends) the Limb class and represent 2 different kinds of Limb.

1) Arm
    a. Static Variables/Fields
        i. ARM_DESCRIPTION – a static constant that serves as a description to this Limb. In this case, it's an Arm. Helps reduce embedded literals.
    b. Methods
        i. Arm – the constructor for creating an Arm object.
        ii. Arm(Limb) – a parameterized constructor which serves as a copy constructor to an Arm.
        iii. toString – an overridden method which returns the Arm's description.
        iv. makeCopy – a newly implemented method to have an Arm object referenced by its parent class (i.e. referenced by Limb).
2) Leg
    a. Static Variables/Fields
        i. LEG_DESCRIPTION – a static constant that serves as a description to this Limb. In this case, it's a Leg. Helps reduce embedded literals.
    b. Methods
        i. Leg – the constructor for creating a Leg object.
        ii. Leg(Limb) – a parameterized constructor which serves as a copy constructor to a Leg.
        iii. toString – an overridden method which returns the Leg's description.
        iv. makeCopy – a newly implemented method to have a Leg object referenced by its parent class (i.e. referenced by Limb).

Limbs Class

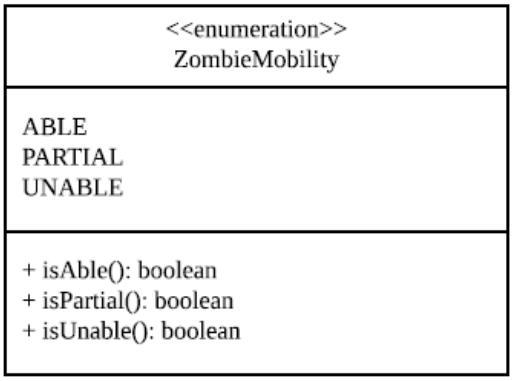


This class serves as a Collection of limbs that an Actor may have. An Actor may have limbs such as an Arm or a Leg. In this game, for now, only Zombies have Limbs.

1) Variables/Fields
    a. MAX_ARMS – this final variable determines the total number of limbs that are Arms. Each Actor who has Limbs can only have a maximum number of Arms equal or lesser than MAX_ARMS.
    b. MAX_LEGS – this final variable determines the total number of limbs that are Legs. Each Actor who has Limbs can only have a maximum number of Legs equal or lesser than MAX_LEGS.
    c. MAX_LIMBS – this final variable determines the total number of limbs it has. Each Actor who has Limbs can only have a maximum number of limbs equal or lesser than MAX_LIMBS.
    d. arms – this variable stores a collection of Arms.
    e. legs – this variable stores a collection of Legs.
    f. allLimbs – this variable stores a collection of all limbs.
2) Methods

a. Limbs – the constructor to construct Limbs. Actor's that have Limbs must specify how many limbs they have.
b. setArms – a new method which instantiates Arm objects and stores it in arms. Declare this method private to prevent any unauthorized adding/creation of Arm objects within Limbs.
c. setLegs – a new method which instantiates Leg objects and stores it in legs. Declare this method private to prevent any unauthorized adding/creation of Leg objects within Limbs.
d. getAllLimbs – a new method which gets a collection of all the Limb objects in this Limbs object.
e. totalNumberOfLimbs – a new method which returns the total number of limbs it has (i.e. how many arms and legs it has).
f. numberOfArms – a new method which returns the total number of limbs that are Arms.
g. numberOfLegs – a new method which returns the total number of limbs that are Legs.
h. hasArms – a new method which checks whether it has limbs that are Arms.
i. hasLegs – a new method which checks whether it has limbs that are Legs.
j. removeLimb – a new method which removes a Limb from Limbs.
k. removeArm – a new method which removes a Limb that is an Arm from Limbs.
l. removeLeg – a new method which removes a Limb that is a Leg from Limbs.

ZombieMobility Class

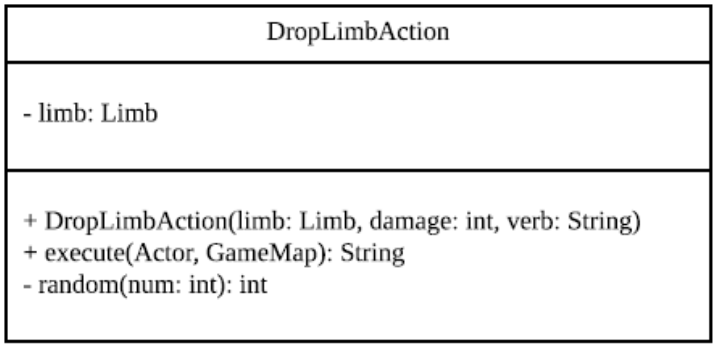


This is an enum class which can be used to represent the current state of mobility of an Actor. I chose an enum class representation as the mobility of an Actor can be thought of as an ability to move. Therefore, this class defines 3 distinct values to indicate the current state of mobility.

1) Enum Values
    a. ABLE – value that represents the ability to be able to move.
    b. PARTIAL – value that represents the ability to be partially able to move.
    c. UNABLE – value that represents the inability to move.
2) Methods

a. isAble – a new method which checks whether the current enum value is set to an ABLE value.
b. isPartial – a new method which checks whether the current enum value is set to a PARTIAL value.
c. isUnable – a new method which checks whether the current enum value is set to an UNABLE value.

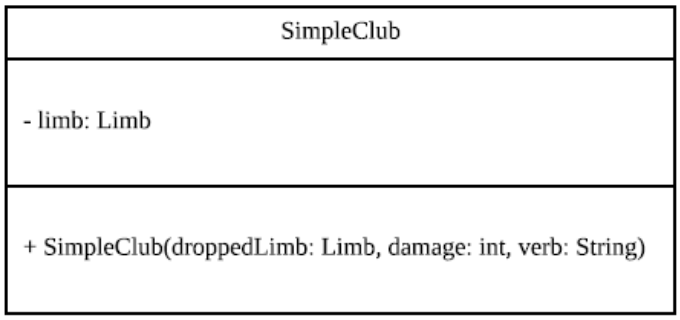
DropLimbAction



This class serves an action to help drop any knocked off limbs to the ground on the Gamemap. It has a Limb attribute to keep track of the Limb it is going to drop to the ground. This class extends the DropItemAction class.

1) Variables/Fields
    a. limb – this variable stores the Limb that is to be dropped.
2) Methods
    a. DropLimbAction – the constructor to create a DropLimbAction object.
    b. execute – a new method which overrides its parent method and creates a SimpleClub object to be dropped on the ground, which has an associated Limb to distinguish between different types of Limb objects.
    c. random – a new method which generates a random coordinate that is adjacent to the Actor that has its limb knocked off.

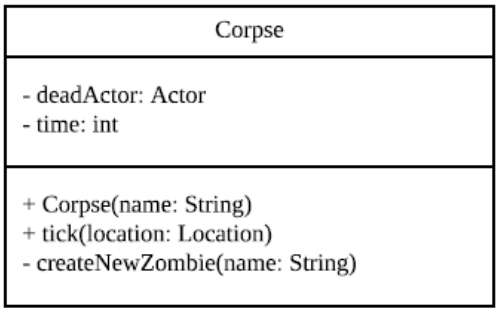
SimpleClub Class



This class is used to model a Limb as a weapon item when dropped to the ground. The reason I created this SimpleClub class is because a Limb is not an Item or WeaponItem and that only when it's dropped to the ground can it be considered as a weapon item. As a result of this, a SimpleClub has a Limb attribute to distinguish the type of Limb that was dropped and derived from.

1) Variables/ Fields
    a. limb – this variable stores the type of Limb it was derived from.
2) Method
    a. SimpleClub – the constructor to create a SimpleClub object.

Corpse



This class is used to model after a dead Actor. Has an attribute deadActor which determines the Actor that has died.

1) Variables/Fields
    a. deadActor – this variable stores the Actor that has died.
    b. time – this variable keeps track of the time that has passed (i.e. how many turns has been played)
2) Methods
    a. Corpse – the constructor for creating Corpse objects.
    b. tick – a new method which overrides its parent class tick method. This method will create a new Zombie object if the deadActor is a human and after 5-10 turns.
    c. createNewZombie – a new method which creates a new Zombie object.

Explanations on interactions and mechanisms

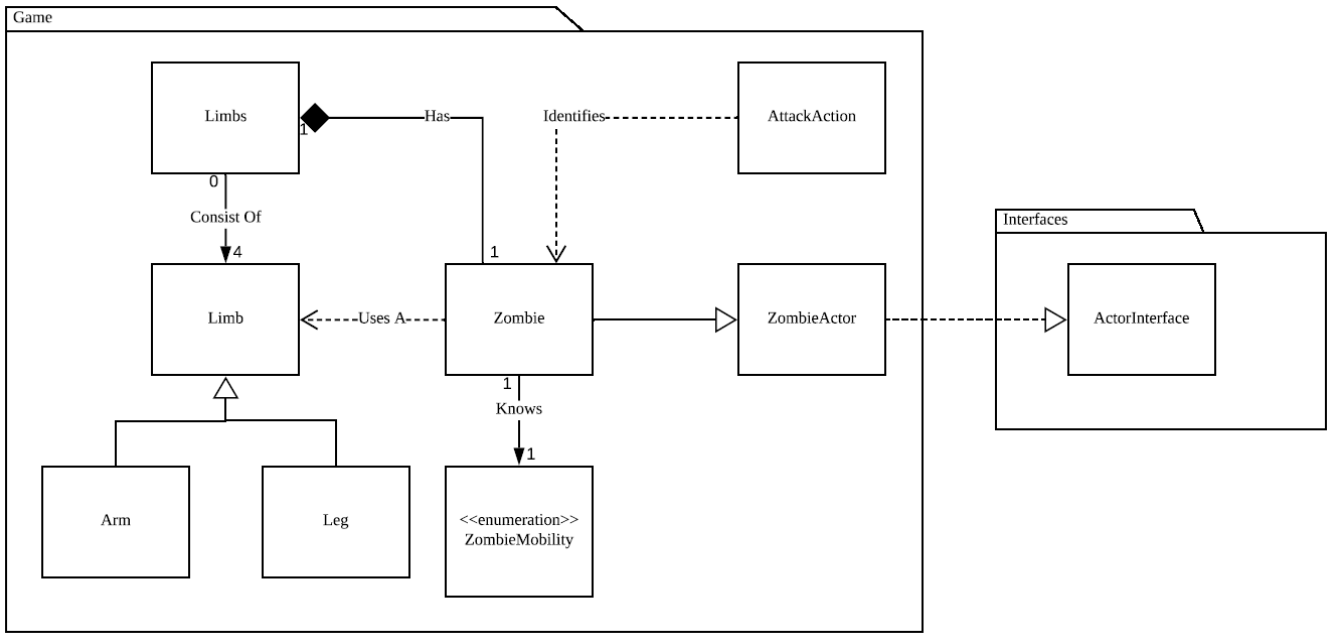


Diagram 1

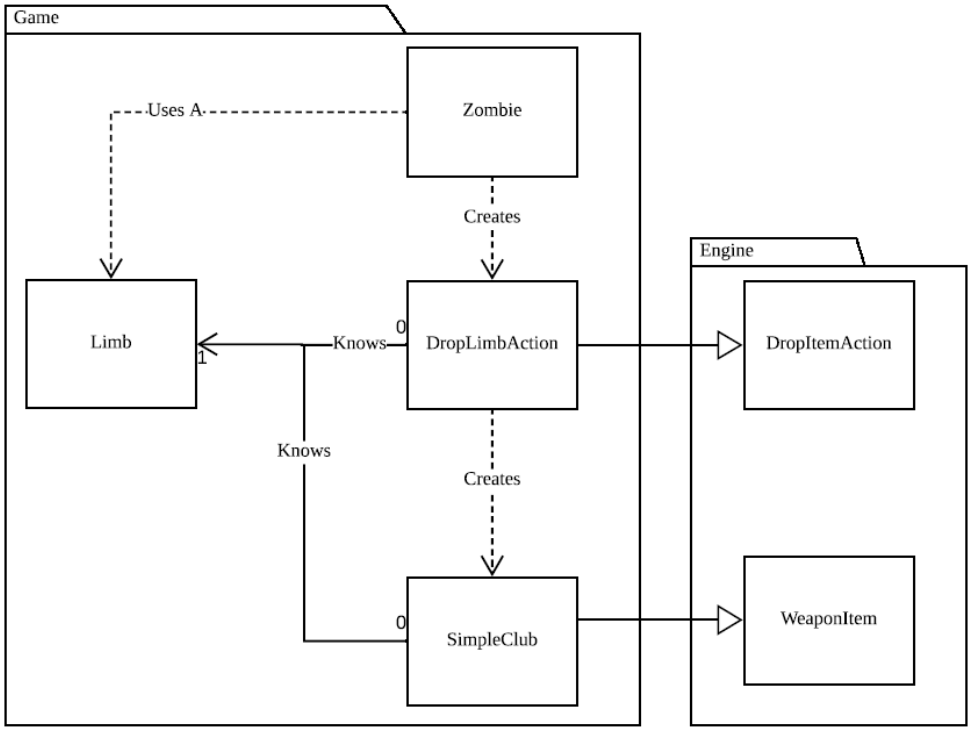


Diagram 2

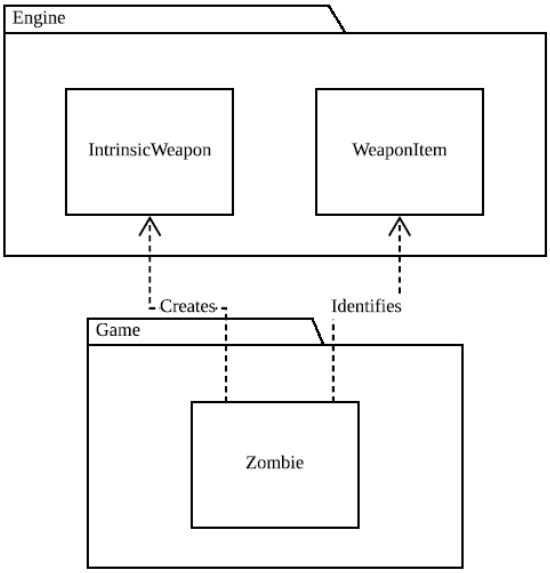


Diagram 3

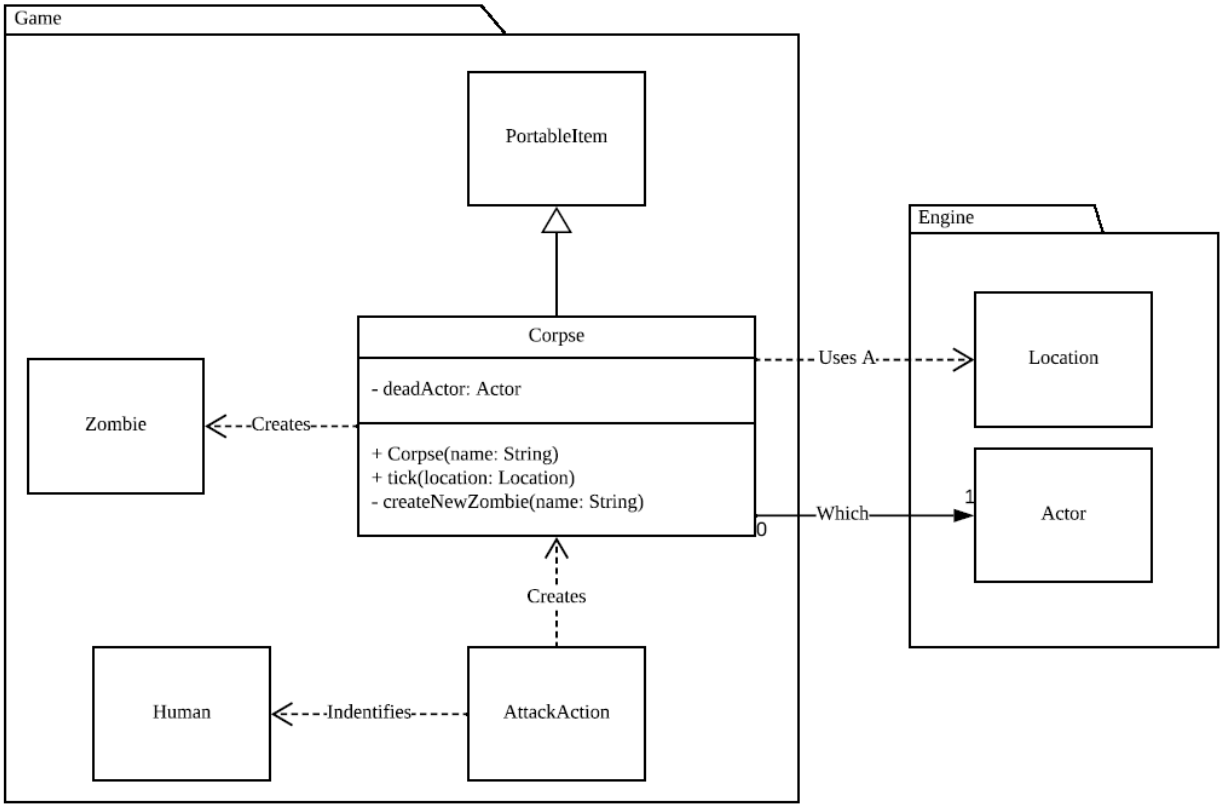


Diagram 4

The diagrams above give's a general overview of the relationships between the different classes and how we intend to extend the original designs based on the questions **Zombie Attacks**, **Beating Up Zombies** and **Rising From The Dead** of this 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.)
   I propose to extend the getIntrinsicWeapon method by creating another IntrinsicWeapon that adds a "bite" weapon option besides the "punching" weapon option and a biteProbability variable to determine the probabilities of punching and biting. On the other hand, we extend the AttackAction class to check for a "bite" IntrisicWeapon through its verb while using the missBite method of the Zombie class to determine the chances of biting. This helps reduce duplicated codes and to keeps it simple by reusing existing implementations (in this case, IntrinsicWeapon). However, there would be a dependency link between the Zombie class with the IntrinsicWeapon class and AttackAction class.
2) A successful bite attack restores 5 health points to the Zombie
   I propose to extend the execute method of the AttackAction class that checks if a bite was successful through the missBite method of the Zombie class. If so, we heal the attacker through the heal method of the Actor class. This makes use of already existing codes to heal the attacker. However, there would be a dependency link between the Zombie class and the AttackAction class.
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)
   I propose to extend the Zombie class by adding a pickUpWeapon method into it that allows the Zombie to pick up a weapon item through existing methods. This would help reduce duplicated codes by making use of existing ones. After this, we add this pickUpWeapon method into the playTurn method of the Zombie so that it will look for a weapon to pick up in every play turn.
4) Every turn, each Zombie should have a 10% chance of saying "Braaaaains" (or something similarly Zombie-like)
   I propose to extend the Zombie class by adding a zombieTalking method into it that allows the Zombie to say something. After which, this method should be added into the playTurn method of a Zombie so that there's a chance of it saying something for every play turn. I chose to create this method inside the Zombie class because a Zombie saying something is a characteristics of a Zombie, thus encapsulating it behavior inside its own 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)

I propose to first add an unimplemented method into the ActorInterface interface called loseLimbs which will determine the chances of an Actor to lose its Limbs, if it has Limbs. In this game, the ZombieActor will implement this method to a return a value of null initially as not all ZombieActors have Limbs for the moment. Next, we will override this method in the Zombie class to implement our own behaviors of knocking off a limb based on the given preferences and the appropriate chances. Finally, we will call this loseLimb method in the AttackAction class to check for whether the target would lose any Limbs given that it was attacked. I designed it this way to prevent any reference errors by declaring a new method within an interface, which will prevent any unwanted downcastings (through polymorphic codes) which may lead to dependencies. Besides that, it ensures that the chances of a Zombie losing its Limbs is encapsulated and defined only within its own class, making it responsible for its own behaviors.

2) On creation, a Zombie has two arms and two legs. It cannot lose more than these.
I propose to create 4 new classes: Limbs, a Limb abstract class, Arm and Leg. Arm and Leg inherits a Limb. Limbs represents a collection of all the Limbs that a ZombieActor may have. Limbs therefore knows a Limb. A Limb can be either an Arm or a Leg, as of now. Finally, a Zombie should have a Limbs attribute, which stores all the limbs a Zombie may have. I designed it this way to modularize every component instead of putting everything within a single class. This also allows for future extensions as we can model other types of Limb objects besides Arms and Leg while having a Limbs class should help new/existing characters to have Limbs which would prevent any form of duplication of codes and improve reusability. Hence, a Zombie has a composite aggregation relationship with Limbs as without an Actor, there is no Limbs. Limbs will have an association with Limb while Arm and Leg are subclasses to Limb.
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.
I propose to add into the Zombie class methods called loseLimbEffects, dropAWeapon and halvePunchProbability. loseLimbEffects would help to implement the effects towards a Zombie after losing an Arm or a Leg depending on how many Limbs it loses after an attack. In this case, if it loses a Limb that is an Arm, depending on the 50% chance, it calls the dropAWeapon method which drops the first occurrence of a weapon item in its inventory by getting a DropItemAction and also calling the halvePunchProbability to reduce the probability of punching. If it loses all its Arms, it will call the dropAWeapon method until there are no weapon items in its inventory. I chose to have all these methods in the Zombie class because a Zombie should be responsible for its own behavior, in this case, its behavior after losing an Arm or both. Besides that, the dropAWeapon method reuses existing methods from Item class to prevent any duplicated codes. However, there would be a dependency relationship between the Zombie class and the WeaponItem class.
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)
Similarly to the question above, I propose to use the loseLimbEffects method and also creating methods skipTurn and isLostLegPreviously. When a Zombie loses a Leg, it will set a newly creaed field of the Zombie class called lostLimbPreviously to a value to help keep track of when to skip a turn. The skipTurn method is then called in the playTurn method of the Zombie class to

determine whether to skip a turn. The zombie also keeps track of whether it is still mobile (i.e. can still walk) by creating a ZombieMobility attribute within the Zombie class. This will help to keep track of the current state of mobility of the Zombie based on the number of Legs it has. The Zombie continues move every turn be continuously reassigning the lostLimbPreviously variable. I chose to have all these methods in the Zombie class because a Zombie should be responsible for its own behavior, in this case, its behavior after losing a Leg. Hence, there is an association between the ZombieMobility class and the Zombie class.

5) If it loses both legs, it cannot move at all, although it can still bite and punch
   I propose to use back the loseLimb method mentioned above to check whether it loses both its Legs because of an attack. If so, it will reassign the behaviors field of the Zombie class to having only an AttackBehavior object only while also indicating the immobility of this Zombie through the ZombieMobility variable we created as well. I chose to reassign behavior array with just one AttackBehaviour object element to prevent any additional code being written by just reusing existing classes.
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)
   I propose to create new DropLimbAction and SimpleClub classes which inherits DropItemAction and WeaponItem classes respectively. This inheritance relationship allows us to reuse the existing methods already defined, preventing any duplications. Whenever a Limb is being knocked off a Zombie, it will create a DropLimbAction object in the Zombie class which has a Limb attribute to indicate the Limb to be dropped and then performs the dropping action by creating a new SimpleClub object, which also has an associated Limb, to be thrown onto the ground. The reason why I created a SimpleClub class instead of a Limb extending an Item class is because a Limb is not an Item. To determine where would the Limb be dropped, we porposed a new method called random which helps determine a random location on the map. I chose to design it as such to modularize the mechanism of dropping a Limb and so that the mechanism is not only defined within one single class, which helps in reducing duplicated code and also improve reusability of the DropLimbAction. However, there would a dependency link between DropLimbAction class and SimpleClub class and also a dependency link between Zombie class and DropLimbAction.
7) Cast-off Zombie limbs can be wielded as simple clubs – you decide on the amount of damage they can do
   As mentioned from the above, when a ZombieActor stumbles upon a SimpleClub on the ground, it will be able to pick it up as a SimpleClub is a WeaponItem, which is portable. We decided to have every SimpleClub have the same amounts of damage as it is considered as just a Limb. This allows us to keep things simple by not creating other methods to wield a Limb as a simple club and to reuse existing codes inside of WeaponItem to pick up the SimpleClub.

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.
   I propose to create a new class called Corpse which inherits a PortableItem. When an Actor is found unconscious, it will create a new Corpse within the AttackAction class. This corpse object

has an associated Actor to determine who this corpse belongs to and a time variable to keep track of the time passed. Everytime the tick method of the GameMap is called, it will invoke this items overridden tick method which checks for the given conditions (5-10 turns later) to be true and that the Actor is a human before creating a new Zombie object. I chose to design it this way to take full advantage of polymorphism through the tick method and also to modularize this process. As a result of this, there is a dependency relationship between AttackAction and Corpse and also a dependency relationship between the Corpse class and Zombie. Besides that, it also has an association relationship between a Corpse and an Actor.