Design Rationale

The purpose of this document is to elaborate on the rationale behind the design decisions taken, in designing the system such that the proposed requirements are met.

The document is structured based on the different actors involved in the system

- Zombie
- Human
    - Farmer
- Player

The following facts with regard to the existing implementation of the project are best elaborated on, for effective explanation of the design decisions taken.

- The game runs as long as the player is conscious and offers each actor (Zombie, Human, Farmer, Player) their turn and executes their action, for the turn, through a common interface. (playTurn and execute methods, for each actor, called from World)

- Each separate action has a specific class associated with it. Execution of the action occurs when the execute method of the action class is called, through the common interface.

    Eg - AttackAction Class for attacking
    PickUpItemAction for picking up an item

- The execution of a particular action for any actor occurs through the class associated with the action.

    Attacking for both Zombie and Player occurs through the AttackAction class

- For the player, the actions available are gathered and each made available as menu options. However, the actions available for the other actors are introduced through “behaviors”, implemented through behaviour classes

    Eg - Zombie’s turn consists of going through the behaviors attributed to the Zombie

        AttackBehaviour
        HuntBehaviour
        WanderBehaviour

    and getting an action corresponding to a particular behaviour, which is then executed.

Notable facts with regard to implementing attacking

The attack action class in particular, has existing functionality to enable

- Attacking with a weapon, if the actor has a weapon or use it's intrinsic weapon instead
- Accounts for when an actor's attack is successful and when it misses
- Dropping of items and removal of actor when an attack kills an actor

Weapons are created using a WeaponItem class, which is an extension of the Item class. When creating a weapon, it's display character, attack damage, can be specified.

Intrinsic Weapons are implemented with a separate class, for intrinsic weapons. It is accessed for each actor through a getIntrinsicWeapon method, which creates and returns an intrinsic weapon relevant to the actor

Eg - Punch for Zombie

Picking up

Picking up is enabled by a class called PickUpItemAction, which is the class corresponding to the action of picking up. For the player, this works by enabling the player to chose the Pick Up action from a menu.

The existing system does not provide any functionality for actors other than player, to implement picking up.

Gathering up actions for the player

The actions for the player are implemented by gathering up all allowable actions and calling the playTurn method which will facilitate the menu selection and from which the chosen action will be executed.

Gathering up all allowable actions is done by looping over all of the items in the inventory of the player, getting all allowable actions corresponding to the player's specific location, etc.

In the existing system, allowable actions are only set, while looping through here, adding each new action to an "Actions" class.

Behaviours

The actions for all actors other than the player, are decided using behaviors. The Zombies and Humans playTurn method consists of looping through the behaviors attributed to it. While the human just has a WanderBehavior.

Ticks

Ticks count the turns passed during gameplay.Ticks denote the passing of time in the game and will be called in instances such as crop growth and human to zombie conversion.

Zombie		**Design Rationale**
Implement bite: Bite-Punch 50% ratio Lower chance of hitting More health damage Bite = +5 health for zombie (Decide on percentages)	Create new Intrinsic weapon In getIntrinsicWeapon method, return intrinsic weapon based on probability Specify health damage	In implementing bite, with emphasis on following design principles, notably DRY and using abstraction and making use of the existing system provided, in a way that the system is extensible, the bite is implemented by creating another IntrinsicWeapon instance for biting, within, the getIntrinsicWeapon method. Choosing between punch and bite will be done with a random selection method. In creating the intrinsic weapon, the damage it'll cause can be specified. The chance of hitting is dictated by a limb check method.
Weapon: ● Zombie picks up weapon if weapon is near (at location) ● If zombie picks up a weapon then punching will not be an option ● zombies can only pick one weapon If arm lost greater chance of biting and a 50% chance of dropping the weapon(**Dropping weapon) ---> Related to limb dropping, which is mentioned below**	Implement a PickupBehaviour to check if the zombie is at a weapon location (The pickup behavior will return a PickUpAction or null accordingly) Existing code accounts for: Moving weapon to inventory, No punching when a weapon is in the inventory If the zombie has a weapon in its inventory, no more inventory slots for items i.e. pickupBehaviours getAction willreturn null	As actions for actors other than player are implemented using behaviours, the design decision was taken to write a new Behaviour class, PickUpBehaviour which inherits from behaviour. This decision was taken, as implementing this similarly to how other behaviours are implemented allows to use existing code effectively, while keeping the code extensible. The PickUpBehaviour will return a PickUpItemAction within the playTurn method, enabling the Zombie to Pick up the item.

		Returning the PickUpItemAction from within a dedicated method, allows to easily introduce conditions like, checking whether a Zombie already has a weapon and returning null instead, whether or not ZombieLimbs are detached. As mentioned, the action of a Zombie is decided by looping over the behaviors. A design decision was taken to have the PickUpItem Behaviour as the first behaviour to loop over. This is because then, the zombie will immediately pick up a weapon, if it's at its location. This decision is needed, to accomplish the intended behavior.
Every turn : Zombie saying braaaains or something similar ---- 10%	Call a method within playTurn to randomly select a phrase.	This needs to be done, to achieve the intended purpose, while making use of functionality provided by the existing code base, so that redundancies are reduced. The chance of when the zombie says a random phrase will be calculated randomly. This will have to be implemented within the play turn method.
Body If one arm lost, ● punching probability halved ● 50% chance of dropping weapon If both arms lost, weapon dropped If a leg is lost, movement speed halved If both legs lost, can still bite and punch	Based on arm limb count, modify probabilities. In the method that detaches arms and legs, pass in location details. Drop weapons accordingly (Similar to code in attack action) Speed change accounted for, by more misses.... Helper methods can be useful in the regard	This involves changing the probabilities based on different conditions, such as the loss of an arm. These are necessary steps for achieving the intended functionality. Achieved with a common method to determine what to do upon limb fall.

<u>Humans</u> If killed turn into zombies in 5-10 turns	In execute method, when an actor dies, creates a Corpse class, check if the actor is a human and if so, set a flag to true to enable mutating to a zombie	When attack Action is called it will create a Corpse class that is a portable item. If the actor is a human then the death flag will get set to true to indicate that it is mutable. The corpse will tick itself and then create a new zombie when the ticks have exceeded the required amount of ticks using the mutate method. Checking if the location is valid is done by the locationValid method. Will return true if the location of the corpse has no actors in it.If an actor is at the location or has the corpse in their inventory the locationValid method will return null
<u>Farmers</u> ● 33% probability of sowing crop : If left alone ripen in 20 turns ● Fertilize crops(reducing growth time) : 10 turns ● Farmers harvest crops on dirt(every patch of dirt is a crop????)	Create farmer class Add to code creating human type: farmer, in Application Call method for sowing crops. (33% probability) (playTurn) : Or sowCrop behavior returning sow crop action, the execute method in sow crop action will create crop class at location. Crop class: Extending Ground. displayChar is changed upon ticks and a flag for harvestability is set to true In crop class, Specify ripening time : Operate under default, Specify ticks (Similar to tick method in tree)	The farmer class will extend the human class. This decision was taken to reduce redundancy since most of the behaviours will be similar to the other humans in the game. Farmers will have a behaviour sowCrop behaviour. Which will be in their list of behaviours. This will give them a sowCrop action where a farmer can sow the crops at their location if it is an instance of Dirt. Crop will extend Ground. the crops will grow for 20 ticks and be tagged as harvestable. This was done to trigger the harvestAction in farmers/players and give them the opportunity to harvest the crop

● Farmers drop crops after harvest ● Players put crops into their inventory ● If hurt humans can eat crops to regain health +10?	If farmer, speed up ripening time(reduce growth ticks) (If location has farmer) Overwrite ground with Displaying crops -getDisplayChar() (Existing code will do this as the map is drawn each tick) If ripe, and farmer in location, harvesting Upon harvest, create food class, inheriting from portable items and add items to location. Also remove crop ((Food extend PortableItems)) if pickupitem instance of harvested crops/crop, call heal method, inherited from actor.	If farmers are on the crop then the crops growth tick will be reduced by 2 Initially crops will have a display character to show that it is still growing after the required number of ticks pass, crops will change their display character to represent a fully grown crop. When the crop has grown fully, the getDisplayChar method will return a different character. Farmers will check their location every play turn and if a crop with a harvestable tag exists the farmer will harvest the crop and drop a food item on the ground. Once the crop is harvested the crop will be dropped as a food item. allowing humans and players to pick it up The harvest action will put the food object directly into the inventory of the player. Zombies will not pick it up because it is not a weapon item When a crop is in a humans inventory they will automatically eat it to gain health. This will be called after a human has less than 75 health.
<u>Player</u> Can harvest and store in inventory	If player at location, call harvest procedure	

Eat food to restore health +10 **(Additional - Print inventory)**	Will have to set access inventory as an action and put it into allowable actions. If the player wants to access inventory print the list of items in the inventory get inventory from player actor and print the list	Farmers have a method to harvest grown crops at their location.
<u>Weapon crafting</u> Zombie arm- club -25 damage Zombie leg – mace – 35 damage	A craft action class is needed. Once the user selects the menu option. The execute method of the craft action class is called. This will create mace/club (This can be put to a craft method in Player, if needed) and add to inventory and remove limb from inventory	As all actions are executed through a common interface and as each action has a dedicated class for it, (This is needed in order for the common interface to work), the design decision was taken to make a craftAction class. The craftAction class will be responsible for creating the maces, clubs and adding them to inventory and removing limbs from inventory. For ease of maintaining a limb count, creating a method for crafting, which would be called by CraftAction, in Player seems beneficial, and makes sense as crafting is an action done by the player, and as such should be preferably encapsulated within the player class. Craftable items have a dedicated class that sets allowable actions
<u>Attack on Zombie</u> Arm/leg falls off 25-40% chance??	A count is needed to limit the number of limbs falling)	
Lost limbs fall to ground at location or adjacent Cast off limbs as weapons, even without crafting	Write a method in zombie that returns a limb, call this method after hurt. (Arrange for it to be called multiple times, with rng involved). Add this returned limb to location as portable item. (@ Player's location)	Attacking on Zombie occurs through the call to the AttackAction method, by the player. In the existing functionality, the AttackAction calls a method "hurt" to reduce health points

	Create a new Actions class and add a CraftAction for the returned limb to it. A menu description should now be visible. If wanting to implement limbs as weapons, prior to crafting, create by extending WeaponItem Implemented as such, the option for whether or not to craft the item, should now be visible.	Our design decision is to call a method to break limbs after the hurt method call: dismember method This method, written within Zombie, will use random selection method to decide whether a limb is broken or not. Further, we have decided to create Arm, Leg classes that we will inherit from a Limb class, which will have common helper methods for the actions of crafting and adding, removing from inventory. When a limb is broken, (using dismember), they are returned. The Arm, Leg classes are only created upon detachment. Further, it is decided to maintain a count of arms and legs in the zombie class, as attributes. When an Arm or leg class is created, the count is reduced as appropriate. The design decision to return the limb through a method call to the AttackAction class is to enable the functionality of crafting these, later on. To enable crafting, the option to do so, must be visible as a menu item. (As mentioned above, the player's menu shows up by getting a list of all the actions) Therefore, crafting should be entered to the list of allowable actions. This is to be done within the AttackAction class, as the Zombie class does not have access to classes such as "Actions", given that they are in the engine package.

		This is why, it was chosen to return the detached limbs .

Our design involves the creation of the following classes.

- Limbs
    - Arms
    - Legs
- Farmer
- Club
- Mace
- Crop
- CraftableItem
- SowCropBehavior
- EatBehaviour
- HarvestBehaviour
- SowCropAction
- CraftAction
- EatAction
- HarvestAction
- ChooseWeaponAction

The exact relationship between these classes is elaborated in the provided UML class diagram, while the design decisions behind adding them are explained above.