DESIGN RATIONALE - ASSIGNMENT 2

Utility

- new enum `DisplayChar` containing object names mapped with their corresponding displayChar

As more Actors and Items are introduced into the game, the number of displayChars required to represent them on the map also grew. As a consequence, it gets increasingly difficult to look up an object's displayChar or to ensure no character is used twice without searching through a large number of codelines.

The enum therefore provides a single point of reference of all the `displayChar`s in use to enhance the process of creating and using them. It also helps to reduce the program's dependency upon literals. Should the choice of character be changed at some point in the future, the change would be fairly easy and quick to track down while still ensuring the overall correctness.

Design principle: *Avoid excessive use of literals*

Requirement 1: Zombie attacks

- new classes IntrinsicPunch, IntrinsicBite extends from IntrinsicWeapon adding `HIT_PROBABILITY`, `heal()`(for IntrinsicBite):

The original design made Zombie's punch an `IntrinsicWeapon` object (it follows that the same should go for Zombie's bite). However, there is no attribute in `IntrinsicWeapon` to show the hit probability or how many health points the punch (or bite) redeems to the Zombie. Although it is still possible to implement without adding new classes, it is not ideal since a punch or bite would not be in control of their hit probability and point-redeeming properties.

Design principle: *DRY, Classes should be responsible for their own property*

- overriding `damage()` for IntrinsicPunch, IntrinsicBite:

The level of damage of a punch / bite is subject to probability (`HIT_PROBABILITY`). The `damage()` method conducts the checking of `HIT_PROBABILITY` to allow the probability to stay class-private. It will then return either 0 (if the punch / bite did not occur successfully) or full damage (if the punch / bite did occur successfully).

Since `damage()` is a public method defined in class `Weapon`, this way of implementation takes advantage of the abstraction whereby `Weapon` communicates with the calling class. Using polymorphism, the calling class (`AttackAction`) doesn't need to know whether a punch, bite or any weapon either physical or intrinsic is carried out in order to inflict the right damage.

Design principles: *Declare things in the tightest possible scope, Dependency Inversion Principle (depending upon abstraction)*

Requirement 2: Beating up the zombie

- new members in `Zombie` class:

- new attributes `private int numArms`, `private int numLegs` showing number of existing arms and legs of the zombie
- new methods `getArms()`, `getLegs()` to get the number of arms and legs
- new methods `loseArms(int lostArms)`, `loseLegs(int lostLegs)` update number of arms and legs when arms or legs are knocked off and if `lostArms` and `lostLegs` are within valid range
- new attribute `private GameMap map` showing the current `GameMap` of the zombie: At some points, the zombie loses arms and has to perform `DropItemAction`. The `execute()` method of `DropItemAction` requires a `GameMap` parameter to identify the location of the dropped item.

Design principle: *Classes should be responsible for their own property, Declare things in the tightest possible scope*

-modifying `playTurn()` **of** `Zombie`**, new dependency with** `MoveActorAction`**:**
The number of legs remaining determines Zombie's ability to move. When the number of legs is 1, Zombie's movement speed is halved or in other words, a `MoveActorAction` is not allowed at each alternate turn. Since `playTurn` method controls Zombie's action at each turn, it should also be in charge of ensuring these conditions are met. playTurn will check whether the `lastAction` (of the Zombie) is an instance of `MoveActorAction`. If true, it must return a non-moving Action (which is either an `AttackAction` or a `DoNothingAction`).

- overriding `getWeapon()` **of** `Zombie`**:**
The weapon used by a Zombie depends on how many arms it has left. When the number of arms is less than 2, the Zombie has high chance of dropping the fighting equipment it is holding (should there be one) and hence, a different procedure for choosing weapon is implemented.
These implementation details should go to `getWeapon()` of `Zombie` to make use of the abstraction layer of `Actor` class. When it comes to selecting weapons for an `AttackAction`, the calling method will interact with this abstraction layer to retrieve the right weapon without having to know the low-level implementation details of Zombies.

Design principle: *DRY, Dependency Inversion Principle (depending upon abstraction)*

- overriding `hurt()` **of** `Zombie`**:**
When `Zombie` is hurt by an attack, one of its limbs may also be knocked off and become simple weapons on the ground. Same as above, these zombie-specific details should be hidden under the abstraction layer of `Actor` class to excuse `AttackAction` from depending on low-level implementation of a concrete class.

Design principle: *DRY, Dependency Inversion Principle (depending upon abstraction)*

Requirement 3: Crafting Weapons

- new class CraftWeaponAction (extends Action class):

This class overrides the `execute` method and `menuDescription` method of `Action class`
A new action specific to the player, designed to craft weapons using zombie's dropped limbs. A dropped `ZombieLimb` can give the player the option of crafting a new weapon that deals large damage compared to normal punches. A crafted `ZombieClub` deals 30 damage while a crafted `ZombieMaze` deals 50 damage. The `execute` method checks if the player's inventory has either a zombie arm or leg and iterates using a `for loop` to create a new crafted weapon object and adds to the player's inventory. An `if-else` statement checks if the item is a `ZombieArm` or a `ZombieLeg`. During the player's `playTurn` method, this action is called if the player's inventory is not empty.

Design principle: *DRY, Classes should be responsible for their own property*

- new classes ZombieArm, ZombieLeg, ZombieClub, ZombieMaze (extends WeaponItem class):
The following classes are all weapons (2 of which are dropped by zombies and 2 crafted), having unique names, characters and damage points. Follows the same design as `Plank` class from the initial game classes. `ZombieArm` and `ZombieLeg` are items dropped at zombies location by a zombie when attacked (they are created in the LimbOffAction class during `execute`). If the player moves over the location of a dropped item, the player can pick up the zombie arm or leg. The player is then asked if he/she wants to craft a weapon using the collected item. `ZombieClub` and `ZombieMaze` are weapons crafted using a zombie arm or leg in the game. This is an action performed by the `CraftWeaponAction` class

Design principle: *DRY, Classes should be responsible for their own property*

Requirement 4: Rising from the dead

- new class `Corpse` extends from `PortableItem`:
+ adding new attribute `private int turns` and `public static final int REBIRTH_TURN`:
`turns` informs the number of turns that have passed since the corpse's creation.
`REBIRTH_TURN` is the minimum number of turns for a corpse to be able to transform into zombie. All corpses share the same `REBIRTH_TURN` (which is currently set at 5).

Design principle: *Classes should be responsible for their own property, Avoid excessive use of literals*

+ overriding `tick()` and new method `riseFromDeath()`:
To keep track of game turn, the `tick` method is a perfect candidate as it is called once every turn. Since the method currently does nothing, overriding is needed to monitor the number of turns that have passed since the corpse's creation.
`tick` will increment `turns` at each call. If `turns` equals or exceeds `REBIRTH_TURN`, it will then call an internal method `riseFromDeath()` which handles the transformation of a corpse into a zombie. The delegation from `tick()` to `riseFromDeath()` was to enable

`tick()` to be solely in charge of querying the current location and / or the actor carrying the corpse and `riseFromDeath()` to be in charge of making the changes in data state.

Design principle: *DRY, Command-Query Separation Principle*

Requirement 5: Farmers and Food

- new class Farmers (extends Human class) :

Has the same behaviours as normal humans. The exception is that if the `Farmer` is standing next to `Dirt`, it has the ability to sow a crop at the location of dirt. This implements a new `Crop` class and inside farmers `playTurn`, it checks if there is dirt next to the farmer's location and sows a crop at a 33% probability. Second feature of `Farmer` is that if there is an unripe crop at the farmer's location, the farmer can fertilize it. I.e it reduces the time to ripe by 10 turns. Last feature of the `Farmer` is that it is able to harvest crops. If Farmer is standing on or next to a riped crop, it removes the riped crop from its location and creates a food item at the crop's location. To achieve this feature, the `Map` class's methods are used.

Design principle: *DRY, Classes should be responsible for their own property*

- class Player (extends Human class) :

During the player's `playTurn` method, if the player is standing next to, it has the option to harvest food. If there is a ripe `Crop` around the player, the player can choose to harvest the `Crop` using `HarvestAction`. The harvested `Crop` is added to the player's inventory as a Food item. If the player has been hurt by a `Zombie`, and has food in their inventory, the player can choose to recover some health (10 health points) using `EatAction`.

Design principle: *DRY*

- class Human (extends ZombieActor) :

During its `playTurn` method, if the human has item `Food` in its inventory, and has sustained damage, health is automatically recovered by eating the food (uses `EatAction`). Should check if `Human` is not a `Zombie` as `Human` extends `Zombie Actor`, therefore even Zombies would start collection and consuming food!

Design principle: *DRY*

- new class Crop (extends Ground class) and new default method `fertilize()` in `GroundInterface` :

A class that extends the `Ground` class, similar to `Dirt` and `Tree` classes. Overrides the `tick()` method from Ground. An unripe crop and ripe crop is differentiated using two different `displayChar`. Extended implementation of `tick ()`, counts the number of turns. Once the number of turns reaches 20, the crop ripes (i.e displayChar changes to `ripe`). A check must be done so that if the number of turns exceed 20, crop still ripes and changes its display character. However, if the `Farmer` decides to harvest a crop, 10 turns are added to the number of turns. This is done using the crop's `setRipe()` and

`getRipe()` methods. If the number of turns is already greater than 10, the crop immediately ripes.

`fertilize()` is a default method with empty implementation which gets overridden in `Crop` class to implement the fertilize process as detailed above.

Design principle: *DRY, Classes should be responsible for their own property*

- new class Food (extends PortableItem class) and new default method `getHealth()` **in** `ItemInterface` **:**
A portable item that the player can carry in his/her inventory. Has a recovery level of 10 that when consumed adds to the player's hit points. A getter `getHealth()` would allow an outside method to access Food's hitpoint value.
`getHealth()` is a default method with empty implementation which gets overridden in `Human` class to recover damaged Humans with its hitpoint value.

Design principle: *DRY, Classes should be responsible for their own property*

- new class EatAction (extends Action class) :
Allows the player to heal if their current hitpoint is less than `maxHitPoints`. Uses the `heal` method of `Action` class. However, the Actor has to be alive for this action to work. This action is called in the player's `playturn` method when the player has sustained damage and has food in his/her inventory. For `Human` actors, food is consumed automatically when they are on a `Food` item on the `Ground`. However the option to eat `Food` from the inventory is given to the Player.

Design principle: *DRY, Classes should be responsible for their own property*

- new class HarvestAction (extends Action class) :
Allows the player to harvest a ripe `Crop`. This action takes the `Actor` and `Location` of the ripe crop as parameters. During execution, the location of the crop is given a new `Dirt` ground and the `Item` (Food) is added to the player's inventory. This action also has a `menueDescription`, where if the player is next to a ripe crop, the player is given the option to harvest the crop. `Farmer` follows the same implementation however it does not use `HarvestAction` as `Farmer` has to automatically harvest a crop.

Design principle: *DRY, Classes should be responsible for their own property*

Limitations:
During Farmer's playTurn the code runs as follows, Farmer checks if there is dirt around to sow a Crop, next checks if Farmer is standing on an unripe crop and fertilizes it, lastly if farmer is on or next to a ripe crop, Farmer harvests it. The limitation is that if there is always dirt around a Farmer, it is highly unlikely that Farmer would look to fertilize or harvest a crop. Assignment design states that if a Farmer performs any of the three above mentioned features, then the Farmer cannot perform another feature during the same turn.