FIT2099 Assignment 3 Design Rationale

By Aaron Dudley and Joshua Searle (RogueCoders)

NOTE: Combining all features into one UML class diagram would make the class diagram extremely complex and unbearable to add / remove / change anything. So we have TWO separate class diagrams. The first one contains features: Mambo Marie, Going to town, Bite Infection, and the previous assignments class diagram. The second diagram contains features: Ranged Weapons, Ammo, and Zombie Dogs.

New Weapons

Ammo

To implement the weapons we first need a concept of ammo in our world. We can create an enum AmmoTypes for the different types of ammo.

Players need to be able to loot ammo, so we create a LootableBox interface, to improve the extendability of the design, then create a new class AmmoBox that implements this interface. AmmoBox also extends Item class, but is not portable.

AmmoBox will have methods to addAmmo, takeAmmo (to take all ammo from box), to loot (enforced by interface), and a tick method (to gradually add ammo back into the box). When you call getAllowableActions, it will either return a LootAction, or null if there is no ammo to loot.

Making AmmoBox instances is tedious, so to improve readability of our code we create a new class AmmoBoxFactory. This class stores the types of ammo the boxes it creates will contain, and the average number of bullets of each type ammo will be in the box. When you call the getRandomAmmoBox it will create an AmmoBox with slightly randomised ammo counts.

LootAction simply calls the loot method of a LootableBox.

Next we need to create an ammo storage device for the player, so we create an AmmoPouch class, and give the player an AmmoPouch attribute. This class has methods that insert ammo, query the instance for how much ammo it has, and a method to take one bullet of a certain type of ammo. When you call the loot method on the AmmoBox, it inserts all of the ammo it has into the AmmoPouch.

Making the ammo pouch object allows us to add an ammo pouch to any other class we like and isn't just limited to the player, which improves the extendability and maintainability of the ammo feature. Making the loot action means we can create other objects that can be looted (maybe a chest with special weapons) and we don’t need to create a new loot action. This reduces repeated code and helps with the extensibility of the design.

(see loot ammo interaction diagram)

Ranged Weapons

To create ranged weapons, we create an abstract class RangedWeapon that extends WeaponItem. We also need to store the ammoPouch it uses, what ammoType it consumes, what the hitChance is, and finally, what its range is.

When we pick up / drop a ranged weapon we need to connect / disconnect the ammo pouch. This means when we call getDropAction we need to return a subclass of DropAction that is DropRangedWeapon that disconnects the AmmoPouch, and vice versa for getPickUpAction.

When we call getAllowableActions, we check that there is ammo of the type that the weapons in the ammo pouch it is connected to. If so, we add a UseWeaponAction, and if not, we return an empty actions list.

If a Player executes the UseWeaponAction, it will call the weapon it is connected to useWeapon method. The useWeapon method depends on which RangedWeapon concrete implementation is being used.

Making an abstract class that the other weapons implement allows to pull out common functionality that will be used by all concretions of the abstract class, which reduces repeated code. So now the concretions of the RangedWeapon class just need to implement a useWeapon method which makes adding a new weapon super simple, which conforms to the OCP as the program is open to extension but closed for modification.

Shotgun

The shotgun's useWeapon method creates four ShootConeAction based on each direction and a new enum Directions (stores N, S, E, W). It then calls the superclass's pickWeaponAction method that puts the list of actions in a menu, to get the user to pick which one to perform.

ShootConeAction extends ShootStrategy, which itself extends AttackTypeAction. When a ShootConeAction is executed, it takes a bullet, then uses the hitChance to determine whether the shot landed. If it did, we calculate all the locations the shot would hit (based on direction and range), and if there is an actor in that location, we will call AttackTypeAction's hurtActor method. This means that it will also damage teammates, which is intended.

Separating out the ShootConeAction allows us to add another weapon that uses the same style of shooting and helps reduce the responsibilities of the class, conforming to the SRP.

(see ShootShotgun interaction diagram)

Sniper

As it is conceivable that we will add more weapons with scopes, we will first create an abstract class, ScopedWeapon.

The ScopedWeapon class extends RangedWeapon. It stores attributes target, roundsAiming, and aimedLastRound.

ScopedWeapon's useWeapon method finds all the actors on the map that are attackable. It adds a ShootScope action for each target, and an AimScope action for each target, unless the target has already been aimed at for 2 rounds. Then again, calls the superclass's pickWeaponAction method that puts the list of actions in a menu, to get the user to pick which one to perform.

If the Player chooses the shoot, we call execute on ScopeShoot, which extends SHootStrategy. We first take a bullet, then query the weapon for the random miss chance, which is determined by the actor aiming at and the rounds they have been aimed at for. If the shot hits, we damage the target based on the damage which is also determined by the actor being aimed at and the rounds they have been aimed at. We then simply call the hurtTarget method.

To handle the damage and hitChance logic, we have methods in ScopedWeapon damage(Actor target), that returns normal damage if they haven't been aimed at, double damage if they've been aimed at once, and instakill damage if they have been aimed at for two rounds. And hitChance(Actor target), that returns normal hitchance if the target has not been aimed at, 1*2 times the normal hitChance if they have been aimed for one round, and finally, 100% chance to hit if they've been aimed at for 2 rounds.

If the Player chooses to AimScope, we call the aim method of the weapon, which checks if the target was last to be aimed at, and if so it increments their roundsAiming. If not, we set the current target to the new target, and set the rounds aiming to one. We won't get the option to aim at a target that has been aimed at for 2 rounds.

We still have a problem that if a Player aims at a player, does a different action then shoots the player, it will still do extra damage. To fix this we add a tick method that sets a variable aimedLast to false. If a ScopedWeapon ticks two rounds in a row without being aimed, it sets target to null, and roundsAiming to zero.

To add the feature that if the Player gets hurt, it resets the roundsAiming counter, we give the ItemInterface a new method, disrupt. Now whenever a ZombieActor is hurt, we call disrupt on all of the items in their inventory. By default, disrupt does nothing, but in ScopedWeapon we can override this and make it set the roundsAiming to zero, and current target to null.

Again, we separate out the aim and shoot functionality into their own classes so that other weapons we add in the future could also use the same weapon functionality. This improves the extensibility and maintainability of the program as well as conforming to the SRP.

(see ShootScoped and AimScoped interaction diagram)

Ending the Game

To alter the game ending logic we need a new class that extends the engines world class, so we created WorldWithEndings class. We also changed the Application's reference to world to this new class.

WorldWithEndings has new Attributes Actor mamboMarie, and endGameMessage with default message of "Game Over. You have died."

We need to alter the logic of processActorTurn so that it checks if the player is alive at every call, so that if a player quits, the player doesn't have to wait for all the other actors to do another move.

To allow the Player to quit, we create a new Action QuitGameAction, which sets a new variable stored, hasQuit, to true. In each processActorTurn we check if the player has quit using a new method hasQuit(). If they have, we set the endGameMessage to "Game Over. You have quit.", and remove the player from the game map. Now every other actor's turn will not occur and once every actor has had their turn, the game will end and the set endGameMessage will print.

If our player.hasQuit() query returns false, we then check if there are Zombies still left on the map, and that Mambo Marie is dead. If this is true, we set the endGameMessage to "Mambo Marie and all of the zombies are dead. You Win!!!" and remove the player from the map. Then again, every other actor's turn will not occur and once the game ends, the game will end and the set endGameMessage will print.

If the player dies as normal, we will use the default endGameMessage.

The way we have added this functionality allows us to easily add more end game conditions (e.g. if there are 1000 rounds, the game ends in stalemate) as we just have to add another check.

Zombie Dogs (2/3 Bonus Marks)

First we create a new class ZombieDog, that extends ZombieActor. It has a list of Behaviours:

- BiteBehaviour (same one the Zombie uses)
- BarkBehaviour
- LShapeHuntBehaviour

- LShapeWanderBehaviour

Every round the dog will go through this list in order until it finds a Behaviour that returns an action. LShapeWanderBehaviour will always return an action so it is the last resort.

BarkBehaviour is a new class that extends AttackTypeBehaviour. It has a range. When getAction is called, it creates a list of attackable actors in the range of the bark, and adds them to a list. It then creates a BarkAction from this list and returns it.

When BarkAction is executed, it calls a new method paralyse on all of the target actors. This new paralyse method sets a boolean paralysed to true. When a ZombieActor has its playTurn method called we first see if the actor is paralysed. If they are, we set their status to not paralysed and then force them to execute a ParalysedAction, essentially skipping their turn.

LShapeHuntBehvaiour stores the max number of moves it will search ahead. When getAction is called on it, we add the location the dog is standing to a queue and do BFS only moving in L shape moves. Every time we go to add a location to the queue we first check if there is an attackable actor adjacent to the location, and if there is, we return the first move the dog would have made to get to that location. We also keep track of the number of movies we are looking away from the dog, and if that distance exceeds the max, we return null as we couldn't find an actor.
(pseudocode included for this process below)

LShapeWanderBehaviour extends from WanderBehaviour and puts all possible L moves into a list. If there is no possible move, it returns a move to its current location. Then in WanderBahviour, we pick a random move.

Pseudo Code for LShapeHuntBehaviour's getBehaviour method (as using an interaction diagram would be too complicated):

```
// BFS style algorithm
dists <- empty map of locations to distances
preds <- empty map of locations to predecessor location
queue <- empty queue
dist <- 0

dists.add(source, 0)  // distance to source is zero
preds.add(source, NULL)  // source has no parent
queue.push(source)

WHILE queue is not empty AND dist is less than maxNoMoves
      current <- queue.serve()
      FOR reachableLocation reachable from current
            IF reachableLocation has not been explored yet
```

```
                IF reachableLocation adjacent to attackable actor
                      RETURN move action to root move to reachableLocation
                ENDIF
            newDist <- dists.get(current) + 1
            dist <- MAX(newDist, dist)
            dists.add(reachableLocation, newDist)
            preds.add(reachableLocation, current)
            queue.push(reachableLocation)
            ENDIF
      ENDFOR
ENDWHILE
RETURN NULL  // no attackable actors in range, or on map
```

Mambo Marie

Class: MamboMarie

Extension of ZombieActor. Dependent on ZombieSpawnAction and VanishAction. Associated with WanderBehaviour.

MamboMarie is a new actor who is responsible for the zombie pandemic. The class utilises two counters listed as attributes; chantCounter and vanishCounter, to simultaneously track the turns on the way to MamboMarie's main actions; chanting and vanishing.

A tick method is used every turn to increment these counters. Using if statements the actor will execute VanishAction when vanishCounter reaches 30. If the chantCounter reaches 10 a ZombieSpawnAction is executed. If neither of these occur, the actor wanders (creates new WanderBehaviour and getsAction).

Assumptions

- Mambo Marie's ZombieCapability is UNDEAD, ie zombies will not attack her
- Mambo Marie initialised at Main level, but appears at a random level
- On 30th turn MamboMarie only vanishes, doesn't also spawn and chant
- Mambo Marie has a 5% chance of appearing on every actor's turn (when she is vanished and not killed)

Class: ZombieSpawnAction

Extends Action. Dependent on Zombie. Can only be executed by Mambo Marie (checks if actor canVanish, an updated feature added to the ActorInterface to remove downcasting). Returns a chant string.

MamboMarie creates 5 instances of Zombie and randomly adds them on the GameMap MamboMarie is located.

Execute method selects a random zombie name from a list of strings, generates random x and y coordinates, checks if a zombie can be added at this random location, and if true adds a new Zombie. This is repeated using a while loop until 5 zombies have been created.
A chanting string is returned that also includes details of the zombies spawning.

When generating random locations across the map we utilised GameMap's getXRange().max() and getYRange().max(), allowing for extendability if future maps are created in different sizes.

When the ZombieSpawnAction is called in the MamboMarie class, the actor's chantCounter is reset to 0.

Class: VanishAction

Extends Action. Dependent on WorldWithEndings. Returns string.

VanishAction serves different roles for different actors. Will only discuss functionality for MamboMarie here.

If the actor is an instance of MamboMarie (uses method canVanish that returns true for MamboMarie), removes MamboMarie from the GameMap and calls a method from the WorldWithEndings class setMamboMarieVanished. This records that MamboMarie is currently vanished and thus can later appear.

Returns strings declaring MamboMarie vanished.

Class: WorldWithEndings

Extends World. Dependent on AppearMamboMarie. Only discussing the class's relationship with MamboMarie here.

Included attributes

- Boolean mamboMarieVanished
- Boolean mamboMarieKilled: initialised as false (at beginning of game MamboMarie hasn't been killed)

These allow the world to track the current status of MamboMarie. The below methods are included to interact with the above attributes

- setMamboMarieVanished
- killMamboMarie

- getMamboMarieKilled

During the processActorTurn method, check if MamboMarie has been killed, and if so MambMarie will never be created again. If MamboMarie isn't killed and is currently on a GameMap, there is a 5% chance they'll execute AppearMamboMarie.

Including this in the processActorTurn method is the more efficient method, as opposed to including this sequence in every child class of Actor's playTurn method.

Class: AppearMamboMarie

Dependent on MamboMarie, WorldWithEndings. Returns string.

Create a new MamboMarie along the edge of a random map.

Find the range of the map's perimeter using GameMap's methods getXRange().max() and getYRange().max(). These coordinates are used to create random locations across the edges of the map. Using these GameMap methods allows for extendability if future maps are created in different sizes.

Once a location is found that MamboMarie can enter, the MamboMarie actor is added and a method from WorldWithEndings - setMamboMarieVanished is called. This records that MamboMarie is currently present on a GameMap and therefore will appear again (can only have one MamboMarie instance at a time).

A while loop with a counter is utilised to avoid an infinite loop if all locations along the edge of the map are occupied.

Assumptions
- MamboMarie always appears along edge of map
- MamboMarie appears at random map (initialised at Main level)

Class: AttackTypeAction

New dependency on WorldWithEndings

The method killTarget now checks if the target is an instance of an MamboMarie using canVanish. If so, using the killMamboMarie method from WorldWithEndings it is recorded that MamboMarie has been killed and thus cannot reappear.

Class: Application

New dependencies on MamboMarie and WorldWithEndings.

Class UML notes

- Applications dependencies on MamboMarie exist but are not shown in our diagram because it caused clutter and made it difficult to read

Application now initialises MamboMarie at random location on Main level and records in WorldWithEndings that she is currently present.

Going To Town

Enum Class : Levels

Introduced an enum class that defines all levels (maps). It holds the name of the level as well as a list of strings that will be used to create the GameMap. The Main Level and Town Level are defined in this class.

Utilised an enum class for extendability. It is now very easy to include additional maps. Appropriate attributes and methods are included in this class to access the names of the levels and the list of strings.

Class: GameMapExtension

Extends GameMap. Dependent on Levels.

Included a constructor that requires GroundFactory and Levels as parameters. This allows the creation of all the GameMaps defined in the Levels enum. This is achievable by inheriting from the GameMap class and utilising the FancyGroundFactory.

Class: Application

New dependencies: GameMapExtension and Vehicle. New associations with GameMap and Location.

New functionalities: Application now creates both GameMaps Main Level and Town Level. These two maps are stored as attributes. New methods are included to retrieve this information. This will be utilised in DriveAction.

Application also initialises a Vehicle on each map at random locations. The location of these Vehicles are stored as attributes, and methods have been included to retrieve these locations. This information will be utilised by the DriveAction.

Humans, Farmers, Zombies and ZombieDogs are initialised randomly across both maps. All items are randomly initialised across the Town Level (including Sniper and Shotgun).

Class: Vehicle

Extends Item. Dependent on DriveAction

Adds an allowable action for the Player - DriveAction. When standing on this non-portable item, the Player has the option to drive (move location) to another map. When the player selects this option, Vehicle creates a DriveAction.

Class: DriveAction

Extends Action. Dependent on Application and Levels. Can only be executed by Player (checks if actor canDrive, an updated feature added to the ActorInterface to remove downcasting). Returns string.

NOTE: Class UML

- DriveAction's dependencies on WorldWithEndings & Levels exist but are not shown in our diagram because it caused clutter and made it difficult to read

We defined the map the Player is travelling to as the destination. DriveAction retrieves the location of the vehicle on the destination level using a method from Application; getVehicleLocation. They are then moved to the location of said Vehicle.

We determine what the destination is by using methods from Application to retrieve the GameMaps and compare them to which map the Player is on.

Assumption

- Actor can only drive to and from location of vehicle on each map that are initialised at beginning of game

Infection (2/3 Bonus Marks)

Humans can become infected when zombies use a bite attack

- There is a 10% chance to infect humans. As a result the human has between 10-20 turns until they're turned into a zombie
- Player can use a vaccine to cure the infection
- Vaccine can be used when standing on the item or picked up to use later
- Player can vaccinate themself before they are infected to prevent any future infections or during infection to cure

Class: BiteAction

New functionality: There is now a 10% chance that when a Zombie executes a BiteAction on a Human, the Human will become infected.

BiteAction infects the Human if the following requirements are met: the target (Human) is still conscious after the bite, they're currently not infected and they're not vaccinated. Upon fulfilling these requirements the BiteAction calls a method infect() from the Human class.

Class: Human

Humans are zombified 10-20 turns after becoming infected. To achieve this we've included 2 new attributes; infectionAge and zombifyAge. When a Human becomes infected as the result of a successful BiteAction, the infect() method is utilised. This sets the zombifyAge to number between 10-20 and starts the infectionAge at 0.

The infect method records that the Human is infected in an attribute infect. Methods are also included to access and interact with this attribute.

A tick method is included to increment the infectionAge every turn that the actor is infected. This is executed in ZombieActor's playTurn.

Class: ZombieActor

New dependency on VanishAction.

Utilise a method in Human tickInfection. This increments the infectionAge of the Actor every turn they're infected. When the infectionAge reaches the randomly set zombifyAge; VanishAction is executed.

Class: VanishAction

New dependency: Zombie. Returns string. VanishAction serves different roles for different actors. Will only discuss functionality for Humans here.

VanishAction's role is to remove the actor without creating a Corpse. If the actor is a Human, we will create a new Zombie with the same name at the same location (ie Human turns into a Zombie). If the actor is a Player, there is no need to create a Zombie because the Player has died and the game ends.

VanishAction also ensures to drop all items from the Actor's inventory at their location before they are removed from the GameMap.

Class: Vaccine

Extends PortableItem. Dependent on VaccinateAction.

Adds an allowable action for the Player - VaccinateAction. When standing on this portable item, the Player has the option to vaccinate and thus cure their infection/prevent future infection. When the Player selects this option, Vaccine creates a VaccinateAction.

The Player can also add the Vaccine to their inventory and vaccinate at a later time.

Class: VaccinateAction

Extends Action. Only executable by Player (uses method canVaccinate from ActorInterface).

VaccinateAction prevents the Player from becoming infected in the future and if currently infected cures their infection.

VaccinateAction calls a method from the Player class vaccinate(). This sets the Player's infection status to false and records that they have been vaccinated. Now when a Zombie executes a BiteAction against the Player, there is no option to infect.

VaccinateAction also ensures to remove the Vaccine from the ground/inventory.

If the Player has already been vaccinated, a string is returned explaining that they've already been vaccinated.

Class: Player

New functionality: vaccinate method. Called in the VaccinateAction. Sets vaccinated to true and infected to false.

Class: Application

New dependency on Vaccine.

Application initialises an instance of Vaccine on both Main and Town Level.

Class Responsibilities

AmmoBox
Responsible for storing the ammo in the box, and managing actor box interactions such as looting ammo.

AmmoBoxFactory
Responsible for creating ammo boxes with some randomness with the amount of ammo in each box.

AmmoPouch
Responsible for storing the ammo that an actor has, and managing weapon / actor to ammo pouch interactions such as depositing ammo and taking ammo.

AmmoType
Responsible for storing the different types of ammo and what the names and weights are.

Direction
Responsible for storing the 4 main cardinal directions, and their respective unit vectors.

LShapeHuntBehaviour
Responsible for creating MoveActorActions towards the closest attackable actor, using L shape movement patterns.

LShapeWanderBehaviour
Responsible for creating a MoveActorAction in a random direction in an L shape pattern.

LShapeMover
Responsible for generating the L shape moves an actor can make from a specific location.

ParalysedAction
Responsible for removing paralysis from actors if they were paralysed.

QuitGameAction
Responsible for setting an actor's quit status to true.

LootableBox
Responsible for providing an interface that enforces that an object can be looted. (i.e has loot method)

LootAction
Responsible for looting a lootable box.

AimWeaponAction
Responsible for aiming a scoped weapon.

DropRangedWeaponAction
Responsible for dropping a ranged weapon by disconnecting the ammo pouch from the weapon.

PickUpRangedWeaponAction
Responsible for picking up ranged weapons by connecting the ammo pouch to the weapon.

RangedWeapon
Responsible for providing an interface for ranged weapons as well as providing common functionality for all ranged weapons.

ScopedWeapon
Responsible for providing an interface for scoped weapons as well as providing common functionality for all scoped weapons.

ShootConeAction
Responsible for shooting a bullet in a cone in a certain direction and range, that damages all actors in range.

ShootScopedWeaponAction
Responsible for shooting a scoped weapon at an enemy.

ShootStrategy

Responsible for providing an interface for shooting strategies and providing common functionality for all shooting strategies.

Shotgun
Responsible for describing how it is used (i.e. implementing its useWeapon method), and providing its statistics (i.e. range, damage, etc.)

Sniper
Responsible for describing how it is used (i.e. implementing its useWeapon method), and providing its statistics (i.e. range, damage, etc.)

UseWeaponAction
Responsible for using given weapons when executed (i.e. calling their useWeapon method.)

BarkAction
Responsible for paralysing all given targets.

BarkBehaviour
Responsible for finding all attackable targets in a range, and then returning a BarkAction with the targets, or null of their are no targets.

ZombieDog
Responsible for describing how its turn is played (i.e. What behaviours it has, and in what order are they executed).

MamboMarie
MamboMarie is a new actor who is responsible for the zombie pandemic.

ZombieSpawnAction
creates 5 instances of Zombie and randomly adds them on the GameMap MamboMarie is located.

VanishAction
Removes the Actor from GameMap without creating a corpse. Also creates a new zombie instance when human zombifies.

WorldWithEndings
Records and sets if MamboMarie has vanished and h.as been killed. Executes AppearMamboMarie when applicable.

AppearMamboMarie
Create a new MamboMarie along the edge of a random map.

AttackTypeAction
Calls method from WorldWithEndings to record when MamboMarie has been killed.

Levels
Enum class that defines all levels (maps).

GameMapExtension
Constructs maps using the enum class Levels.

Vehicle
Item that allows a player to drive to another level.

DriveAction
Action that moves a Player to another level.

Application
New functionality: initialise all new items and actors and maps

BiteAction
New functionality: 10% chance when zombie bites a human they'll become infected

Vaccine
Item that can cure infection

VaccinateAction
Action that cures/vaccinates Player

Human
New functionality: attributes to track infection and methods to interact

Game Engine Changes

Weapon's Damage Method

Adding an actor parameter of target to the damage() method would allow some weapons / attacks to do extra damage to certain targets. For example, we could add a method to actors, hasBrittleBones, then in a mace item's damage method we could check if the actor hasBrittleBones(), and if they do, do double damage to that actor. This solves one issue in the ranged weapon's shooting mechanics where the damage it does depends on the target. A disadvantage of implementing this could be an extra dependency between some weapons and actor (in the previous example, the mace would be dependent on the actor class) which is disadvantageous as we aim to ReD.

Actor’s Hurt Method

Changing the return type from void to string would allow us to add interesting strings to the execution of some attacks. For example, say a zombie is hurt and their arm falls off. We could return a message of “ARRGGHH my arm fell off.” We could check every time the zombie is damaged if they have lost a limb, but this would lead to lots of repeated code, and it should be the responsibility of the zombies class to generate these messages. If an actor did not say anything when they are hurt, we could null. This is a potential solution to allowing zombies to return messages saying their arm had fallen off. Some disadvantages of this would be having to check if the hurt method returned a string, or just returned null as the actor had nothing to say. This would increase the complexity of some classes / methods, which would decrease readability of the code.

Display Class

This class has no internal state and it would be illogical to have multiple instances of this object, however requires an instance to call its methods (i.e. methods aren’t static). As a solution you could use the singleton design pattern as it controls access to a shared resource, the display, it is feasible that this instance will be used by multiple areas of the program, and there will only ever be one display. This would reduce long message chains of strings that would be printed on the display anyway, and would reduce the number of parameters for some method calls (e.g. playTurn for actor). Some disadvantages of using the singleton would be increased difficulty in unit testing as making a mock display class would be more difficult, and using a singleton encourages adding extra dependencies to some classes.

All Actors Have To Do All Actions / Behaviours

It seems strange that all actors need to be able to do all actions. We could just return null if an action is performed by an illegal actor but this seems like a workaround to the LSP and would require lots of type checking of objects which isn’t ideal. Our proposed solution is to have actors' playturn method execute these actions. Now instead of actions requiring the actor parameter for execution, we could swap this out for a different interface. For example, the BarkBehaviours getAction method instead of being getAction(Actor actor, GameMap map) could be (Barker actor, GameMap) and then ZombieDog could implement the interface of Barker, meaning that only the Zombie dog could use that behaviour. Similarly for BarkAction execute would change from execute(Actor actor, GameMap map) to (Barker actor, GameMap map). This new Barker interface could have an abstract method barkAt(Actor actor) which could have different logic for different bites. Doing this would remove world’s dependency on actions as well.

Zombie Behaviour Dependency Injection

Although we could have changed this ourselves, it was not required and we are only marked on the features we add, but Zombies would have lots of dependencies removed if dependency injection was used when constructing the Zombies behaviours. When we got the code, Zombie was dependent on AttackBehaviour, HuntBehaviour, and WanderBehaviour all unnecessarily.

Instead we could use constructor injection and add a parameter that is a list of behaviours that then the instance variable gets set too. Not only does this reduce dependencies, but would also allow us to randomise the Zombies behaviours (e.g. some zombies have higher range on their hunt behaviour, some zombies could even have new Behaviours). This conforms to ReD and the DIP as zombies would now only be dependent on Behaviour and the other Behaviours have to implement the Behaviour interface. Some disadvantages would be added complexity when creating Zombies or other actors but you could hide the creation logic in another class.