Going to Town

1 new class was implemented and 1 existing class was modified.

Vehicle class added

Vehicle class extends from Item. The purpose is this class is to move the actor from one location to the other.

Instance Variable

1. String name,
   - Name of the vehicle(It is not unique)
2. char displayChar
   - Char displayed on the map
3. Boolean portable ->False

Constructor

1. String name,
   - Name of the vehicle(It is not unique)
2. char displayChar
   - Char displayed on the map

Methods

1. public void addAction(Action action)
   It adds an action to this.allowableActions

2. public void addMoveAction(GameMap map,int x,int y,Actor actor,String direction)
   It checks whether the actor appears on the map or not. If actor is not in map, it moves actor to map based on the map, the x and y coordinate and prints the direction(String) of the player's movement.

Application class modified

A gameMap called ghostTown is added.
Two vehicle called car1 and car2 is added.

- Car1 is in charge of transporting player from map to the ghostTown. Car2 is in charge of transporting player from ghostTown to map.

New weapons: shotgun and sniper rifle

13 new classes were implemented and 2 existing classes were modified.

Ammunition
Classes added

1. Ammunition
   Abstract class for default implementation of ammunition (10 rounds for a pack of ammunition). If anybody wishes to add other default values to ammunition they can easily add on to this class in the future.
   - Instance Variable
     private int rounds;
     Acts as a counter

   - Constructor
     Inherits name and displayChar, each round is set to 0.

   - Methods
     public int getRounds()
     Return rounds
     public void shotFired()
     Decrement rounds

2. AmmunitionRifle
   - Constructor:
     Name and displayChar is set to "Shotgun Ammunition" and :

3. AmmunitionShotgun
   - Constructor:
     Name and displayChar is set to Rifle Ammunition ';'

Classes Modified

1. Human
   - Methods
     public boolean hasAmmunition(Class<?> ammunitionType)
     Checks if actor has ammunition in inventory.
     public Ammunition getAmmunition(Class<?> ammunitionType)
     Gets ammunition from inventory.

Addition of Shotgun and Sniper Rifle

Classes added

1. **RangedWeapon**

   Abstract class that extends from WeponItem. Used to set default values and implement similar functionalities that Shotgun and SniperRifle has so as to not repeat similar code in both classes.

   Constructor

   public RangedWeapon(String name, char displayChar)

   Calls super(name, displayChar, 10, “uses the “ + name + “ to “). Adds a RangedWeaponCapability.WHACK to its set of capabilities.

   Methods

   @Override

   public int damage()

   If it is shooting, default damage done is 15 points. Else, it whacks for 10 damage.

   @Override

   public String verb()

   If if is shooting, “shoot” is appended to the verb and returned, otherwise “whack” is appended.

2. **RangedWeaponCapability**

   Enum class to check if Weapon is currently being used to fire or as a melee weapon.

3. **SniperRifle**

   Constructor

   public SniperRifle(String name, char displayChar, int damage, String verb)

   Name is set to “SniperRifle”, displayChar is set to ‘R’, damage is set to 20 and verb is set to "wacks with SniperRifle")

4. **Shotgun**

   Constructor

   public Shotgun()

   Calls super(“shotgun”, ‘|’).

Classes Modified

1. **Application**

   -> Add shotgun and sniper rifle to the map

2. **Human**

   Methods

   public weapon getWeapon()

   Calls getHighestDamageWeapon(). If getHighestDamageWeapon returns null(meaning there’s no weapon in inventory), getIntrinsicWeapon() is returned

   private weapon getHighestDamageWeapon()

   Returns the weapon with the highest damage. If there is no weapon in inventory, return null

public boolean hasRangedWeapon(Class<?> RangedWeaponType)
Checks if there is an instance of RangedWeapon in the actor's inventory.
public RangedWeapon getRangedWeapon(Class<?> RangedWeaponType)
Returns the RangedWeapon of the type specified, else if there is none, then return null.

Firing shotgun
Classes added

1. **DisplayShotgunAction**
   Attributes
   private Menu submenu
   The submenu to be displayed
   private Display display
   The display the submenu should be on
   private Actions shootDirections
   Actions to be passed to the submenu
   private RangedWeapon shotgun
   The shotgun chosen for this action.
   private Ammunition ammo;
   the ammunition used to fire
   Constructor
   public DisplayShotgunAction(Display display, RangedWeapon shotgun,Ammunition ammo)
   It is passed the display and the shotgun object that is firing, and the ammunition.
   Methods
   public String execute(Actor actor, GameMap map)
   Puts a ShotgunFireAction for every exit of the actor in a submenu and displays that submenu. The action picked by the user, is then executed and returned.
   public String menuDescription(Actor actor)
   Returns a string saying actor fires the shotgun.

2. **ShotgunFireAction**
   Attributes
   private Exit direction
   The direction the actor fired the shotgun in
   private RangedWeapon shotgun
   The shotgun being fired
   private Ammunition ammo;
   The ammunition used to fire
   static final String NORTH = "8"
   static final String NORTH_EAST = "9"
   static final String EAST = "6"
   static final String SOUTH_EAST = "3"
   static final String SOUTH = "2"
   static final String SOUTH_WEST = "1"
   static final String WEST = "4"

static final String NORTH_WEST = "7"
static and final hotkeys for all possible directions to reduce literals in the code so that people who see this code in the future can easily tell which direction has what hotkey and they won't accidentally mix up the numbers if modifying the class because it's easier to keep track by direction name and not a number.
<u>Constructor</u>
public ShotgunFireAction(Exit e, RangedWeapon shotgun,Ammunition ammo)
Initializes direction and shotgun. Adds RangedWeaponCapability.SHOOT to its set of capabilities so that the shotgun object knows that it is being used to shoot right now and can change the damage and verb accordingly.
<u>Methods</u>
public String execute(Actor actor, GameMap map)
Calls the private method getRange(map) to get the list of locations within range. Then iterates through the range to see if there are actors. The actors within range have a 75% chance of getting shot.
public String menuDescription(Actor actor)
Returns a string saying fire the shotgun in what direction.
public String hotKey()
Return the exit's preferred hotKey.
private ArrayList<Location> getRange(GameMap map)
Gets the locations in range depending on the direction fired.

Firing Rifle

1. **DisplayRifleAction**
   <u>Attributes</u>
   private Menu submenu
   The submenu to be displayed
   private Display display
   The display the submenu should be on
   private Actions shootTargets
   Actions to be passed to the submenu
   private RangedWeapon rifle
   The rifle chosen for this action.
   Private Ammunition ammo
   The ammunition of the weapon
   Private int maxRange
   Initialize to 100
   private HashSet<Location> visitedLocations
   A hashset of location
   private ArrayList<Location> getAllZombieLocation
   Location of all the zombie

   <u>Constructor</u>

public DisplayRifleAction(Display display, RangedWeapon shotgun,Ammunition ammo)

It is passed the display and the rifle object that is firing, and also the ammunition.

Methods

public String execute(Actor actor, GameMap map)

Puts DisplayRifleSpecialAction(actor,rifle,ammo) for every zombie in that range

public String menuDescription(Actor actor)

Shows that actor using the rifle

p**ublic ArrayList<Location> getLocation(Actor actor, Location loc)**

Gets the location of zombie

private boolean containsTarget(Location here)

Boolean that returns whether it is a target or not

private ArrayList<Location> search(ArrayList<ArrayList<Location>> layer)

Searches every layer

private ArrayList<ArrayList<Location>> getNextLayer(Actor actor, **ArrayList<ArrayList<Location>> layer)**

Get every layer

2. **DisplayRifleSpecialAction**

Attributes

private Menu submenu

The submenu to be displayed

private Display display

The display the submenu should be on

private Actions chooseAction

Actions to be passed to the submenu(Aim or Attack)

private RangedWeapon rifle

The rifle chosen for this action.

private Ammunition ammo;

The ammunition of the weapon

private Actor target;

The actor that is the target

Constructor

private RangedWeapon rifle

The rifle chosen for this action.

private Ammunition ammo;

The ammunition of the weapon

private Actor target;

The actor that is the target

Method

1. public String execute(Actor actor, GameMap map)

   Adds RiffleAimAction and RiffleAttackAction
2. public String menuDescription(Actor actor)
   Prints the target is chosen.

3. **RifleAimAction**
   Instance Variables
   private Actor target;
   The actor getting hurt
   private RangedWeapon rifle;
   The weapon used

   public String execute(Actor actor, GameMap map)
   Calls the aim method in rifle

   public String menuDescription(Actor actor)
   Returns the actor aiming at the target

4. **RifleAttackAction**
   Instance Variables
   private RangedWeapon rifle
   The weapon used
   private Random rand = new Random();
   The random generator
   private Ammunition ammo;
   The ammo
   Constructor
   super(target);
   The target
   this.rifle=rifle;
   The weapon used
   this.ammo = ammo;
   The ammo
   rifle.addCapability(RangedWeaponCapability.SHOOT);

   public String execute(Actor actor, GameMap map,Weapon weapon)
   If aim is 0, the chances are the same and the damage
   If aim is 1, the chances are .9 and the damage doubled
   If aim is 2, it is a insta kill.
   Check ammo. If ammo is 0, then removed ammo from inventory.

5. **SniperRifle**
   Instance Variable
   Private int aiming=0

Private int damage = damage();
Private Actor actorTarget=null
Constructors
Name is set to SniperRifle and displayChar is }

Method
public String execute(Actor actor, GameMap map,Weapon weapon)
If there is actorTarget and the target is the same actorTarget,aiming increases.
Else, change the actorTarget to target and aiming is set to 1

public int getAim()
return aiming

public Actor getCurrentTarget()
return actorTarget

public int damage()
Overrides the previous damage.
If aiming is 0, return normal damage
If aiming is 1, return doubled damage
Else return 100.

Public void reset
Reset everything to normal(aiming=0,actorTarget=null, and remove shoot capability)

Mambo Marie

3 new classes were implemented.

VoodooPriestess

Inherits from ZombieActor because the priestess is not on Human's team (should not be assigned ZombieCapability.ALIVE) but is also not a Zombie.

ATTRIBUTES

1. **private int chantCounter**
   Counter to keep track of the number of times the VoodooPriestess has chanted.
2. **private int turnsOnMap**
   A counter for how many turns has it been since Mambo Marie has appeared on the map.
3. **private Behaviour[] behaviours**
   Only WanderBehaviour right now. But is stored in an array so that is easy to add other behaviours if we want to in the future.

METHODS

1. **VoodooPriestess(String name)**
   Constructor for a voodoo priestess. Calls super(name, '&', 200, ZombieCapability.UNDEAD). It has a parameter for the name (instead of fixing the name as 'Mambo Marie') in case we want to instantiate more Voodoo priestesses in the future. The maxHealthPoints is 200 because she should be hard to kill and the ZombieCapability is UNDEAD because she's on the same team as the Zombies.
2. **playTurn(Actions actions, Action lastAction, GameMap map, Display display)**
   Increments turnsOnMap by one. If she has spent 30 turns on the map, she will be removed from the map and turnsOnMap will be reset to 0. If turnsOnMap is divisible by 10 (every 10 turns she spends on the map), a new ChantAction is created. Otherwise, we will loop through the other behaviours she has to get an action.

ChantAction

Inherits from Action.

ATTRIBUTES

1. **private int chantCounter**
   The number of times the actor has invoked ChantAction. This is for naming the zombies.
2. **private Random rand**
   Random generator to generate random locations.

METHODS

1. **ChantAction(chantCounter)**
   Constructor for ChantAction the nth time the actor has chanted is passed as parameter.
2. **@Override**
   execute(Actor actor, GameMap map)
   Creates five new Zombie objects at random locations in the map. They would have names "Zombie Minion" + which chant it came from and what number zombie it was in that particular chant (e.g. Zombie Minion 3.2 indicates that this zombie was the

second zombie to rise from the dead from the actor's third chant). This naming convention allows the zombies to have unique names, the player to know how many times Mambo Marie has chanted, and allows zombies that have risen from the chant to be named more dynamically than picking out names from a fixed collection of names. Returns the menu description..

3. **@Override**
   menuDescription(Actor actor)
   Returns a string saying actor chants and 5 new zombies have risen from the dead.

ZombieWorld
See design rationale for this class under section 'Ending the game'.

Ending the game

2 new classes were implemented and 2 existing classes were changed.

Application
Change world to be a ZombieWorld object instead of World from the engine package. Pass world as a parameter to player to give the option to quit.

Player
ZombieWorld is now an instance variable and is passed through the constructor. QuitAction(zombieWorld) has been added to the list of actions in playTurn. To quit, it was either this or we had to put QuitAction in an overridden processActorTurn() in ZombieWorld. But we decided that if we put it in processActorTurn() we had to downcast to check that QuitAction was only added for the player's list of actions and also the rest of the method was the same as the super method so there would have been a lot of repeated code. And even though having an instance variable to store the ZombieWorld in player would give player access to its other methods, there is no way for the user to call these methods so nothing can be changed in ZombieWorld. Decidedly, this is the best way to implement quit.

QuitAction
Inherits from Action.
ATTRIBUTES
1. **private ZombieWorld zombieWorld**
   The world player wants to quit from.

METHODS
1. **QuitAction(zombieWorld)**
   Constructor for QuitAction. zombieWorld instance variable is initialised.
2. **@Override**
   execute(Actor actor, GameMap map)
   Calls zombieWorld.quit() to change its status so that it can stop running. Returns the menu description..
3. **@Override**
   menuDescription(Actor actor)
   Returns a string "Quit".
4. **@Override**
   hotkey()
   Returns a character "0" for the hotkey to quit.

ZombieWorld
Inherits from World.
PRIVATE ENUM CLASS
GameStatus
Tells if the game has been won, lost, the player has quit, or the game is running. Declared as a private class within ZombieWorld because keeping track of the outcome of the gameplay should be encapsulated within this class itself.
ATTRIBUTES
1. **private GameStatus currentStatus**

To keep track of the currentStatus of the game.

2. **private VoodooPriestess mamboMarie**
   Mambo Marie is initialized in the world itself so that she can easily be removed and put on map while still processing her turn.
3. **private Random rand()**
   Random generator for chances of mamboMarie appearing that turn.

METHODS

1. **ZombieWorld(Display display)**
   Constructor. It passes display to super constructor.
2. **@Override**
   run()
   Almost the same as the super method, except that for each turn, Mambo Marie has a 5% chance of appearing on the map the player is currently on. She will appear at the coordinates (0,0) unless there is something blocking there, then we will go through the locations in the top edge, and try to add her. We implemented the appearing of Mambo Marie this way, so that she can appear easily on any map in the world and she would appear on the map where the player is currently on.
3. **quit()**
   Sets the currentStatus to GameStatus.QUIT. This method was implemented so that currentStatus wouldn't have to be declared as public and so other classes won't be able to change it. The only time another class can change the status, is to quit.
4. **@Override**
   stillRunning()
   Checks if game should still be running and if it should, then return true. If not, then change the status if it needs to be done and then return false. To check if the game has been won or lost, we iterate through actorLocations and check how many zombies and humans are left. If there are no more humans, the game is lost. If there are no more zombies and Mambo Marie has been defeated, the game is won.
5. **@Override**
   endGameMessage()
   Prints a string depending on the outcome of the gameplay. It will throw an error if the game has ended and this method has been called, but the status is not GameStatus.LOST, GameStatus.WON or GameStatus.QUIT.

ASSIGNMENT 2 MODIFICATIONS

Zombie Attack

1. Create a new behavior class call ZombieAttackBehavior
   - Returns a new ZombieAttackAction instead of an attackaction
2. Create a new attackaction called ZombieAttackAction class
   - Since zombieAttackAction is only for zombie, the if-else statement that was used to check the class for actor and target is removed from the original AttackAction's execute method. Therefore ZombieAttackAction only check if the weapon is a bite or punched and whether the target is dead or not.
3. Modification of zombie class
   - zombie class's behavior list replace AttackBehaviour with ZombieAttackBehaviour

4. Create a new attackaction called HumanAttackAction.
   - Since this guarantees that the target is a zombie, it will check whether the zombie loses any legs or hands and will add the limbs to the map.
5. Create a new behavior class call HumanAttackBehaviour
   - Returns a new HumanAttackAction
6. Modify zombie class by overriding the getAllowableAction by replacing HumanAttackAction instead of AttackAction.
   - This is because getAllowableAction is all the action that is allowed to act on the current actor(zombie) by other actor(human).

7. The old version of attackAction replaces the current AttackAction so that if a new class that has no special features was added, they will be using attackAction, with the exception of the addition of human corpse(from assignment 2 corpse) in the method isDead().
8. A new method called isDead is added to check whether the target is Dead or not.
   - If the target is human and the target is dead, it will turn into a human corpse and will turn into a zombie.
9. The execute method was being overload multiple times. But doing this, we reduce repeated code and it is more accessible

i) public String execute(Actor actor, GameMap map,Weapon weapon,Double chances)

Shows that you can now set the percentage of success.

ii) public String execute(Actor actor,GameMap map,Weapon weapon,int damage)

Shows that you can now choose the weapon and set the damage you want the zombie to take.

iii)public String execute(Actor actor, GameMap map, Weapon weapon)

Shows that you can set the weapon

All this modifications are made to make the code more readable and extendable. The system now do not need to check if the actor and target is a zombie or not. Since the part where the target,actor and weapon that will be the one changing depending on with attackaction it has, and the part where the target is checked whether it is conscious or not, isDead reduces the repeated code in all AttackAction classes. It makes the code more extendable and readable.

Farmer Behaviours

HarvestBehaviour, FertilizeBehaviour and SowBehaviour are combined into one behaviour called FarmingBehaviour. This is because there are some overlap in functionality such as when harvest and fertilizing, the Farmer looks at the location where he is currently standing and Farmer checks exits when harvesting and also sowing crops. So in order to reduce the dependencies and repeated code, I've decided to combine them into one behaviour which then calls the different farming actions.