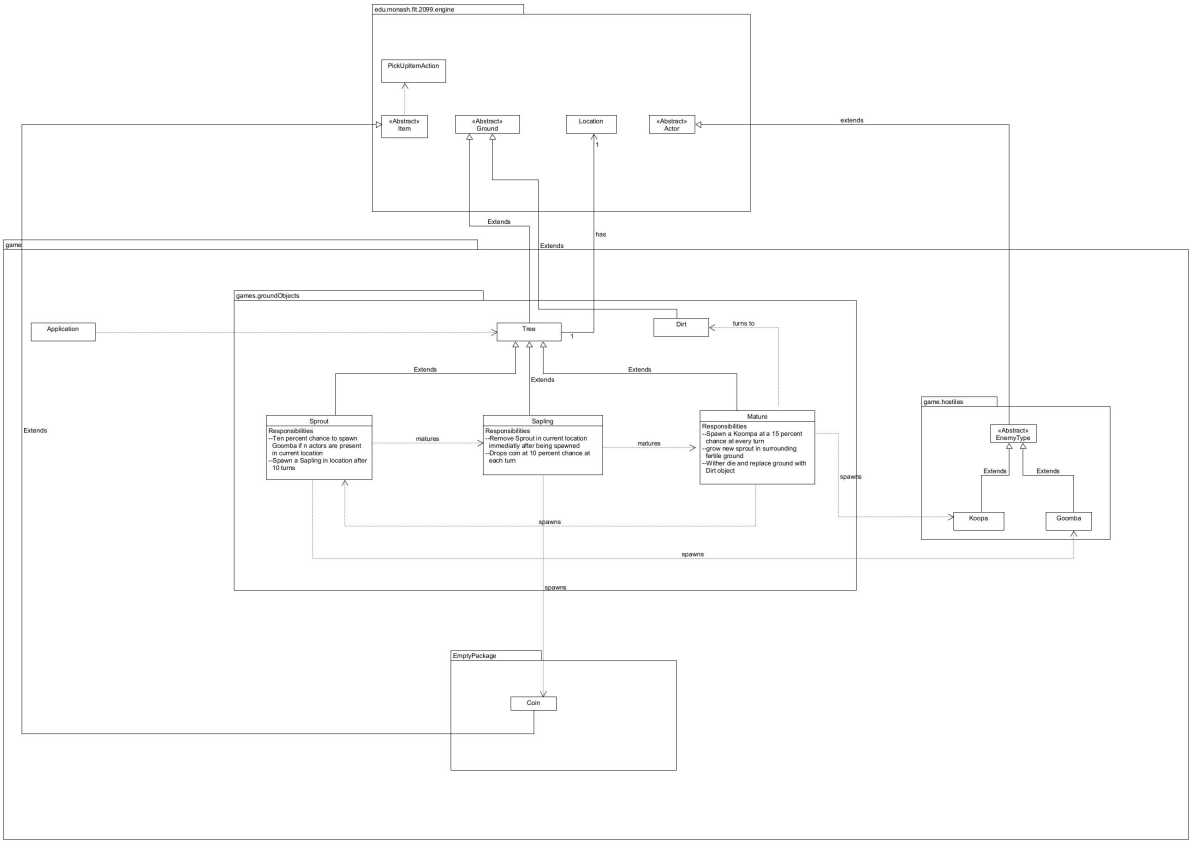
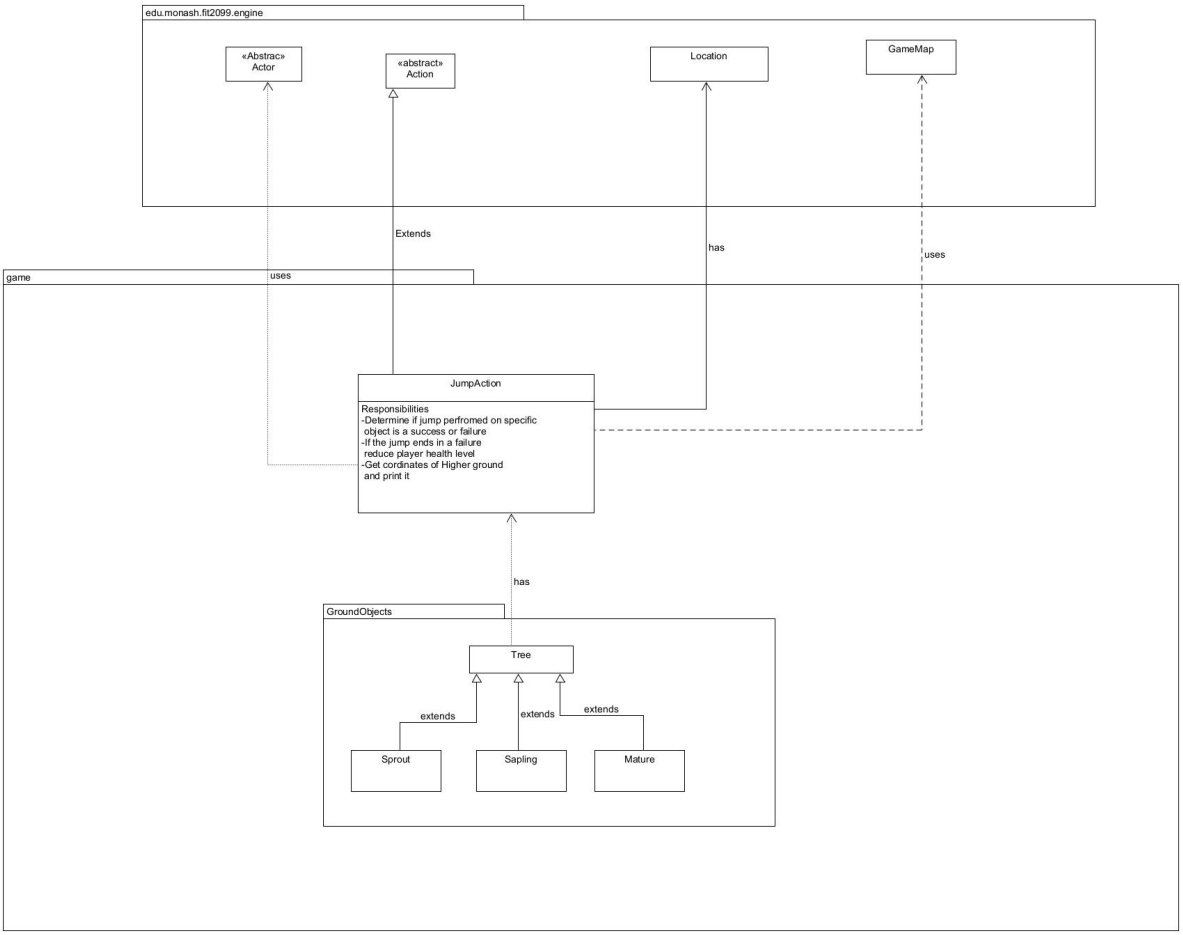
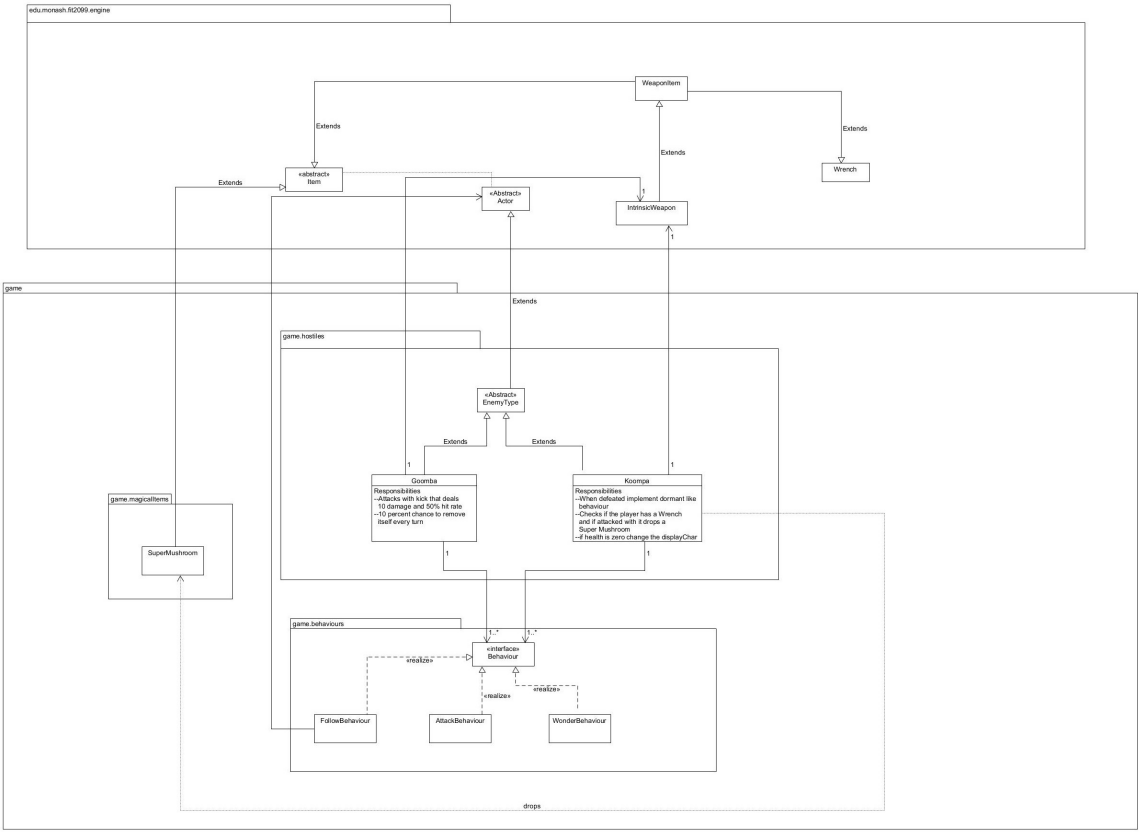
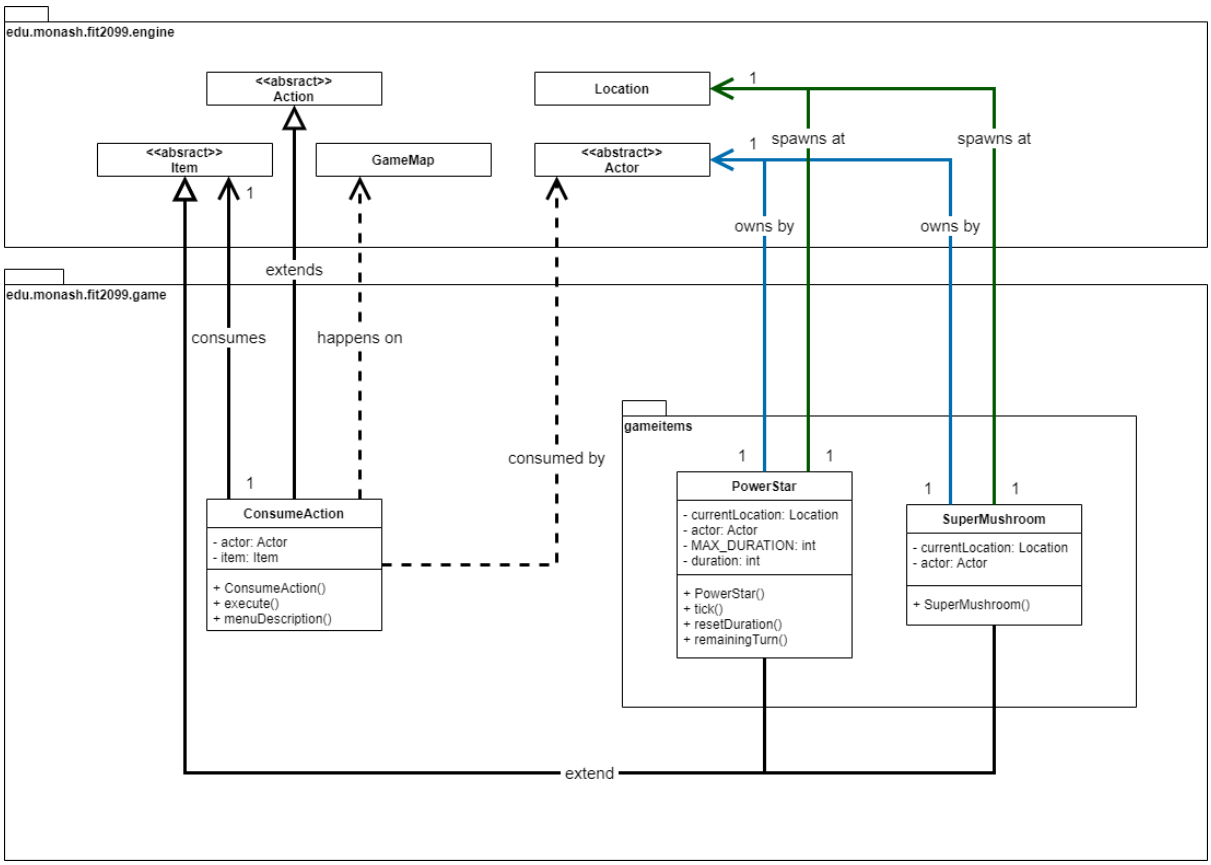
Java Rogue - Like Game

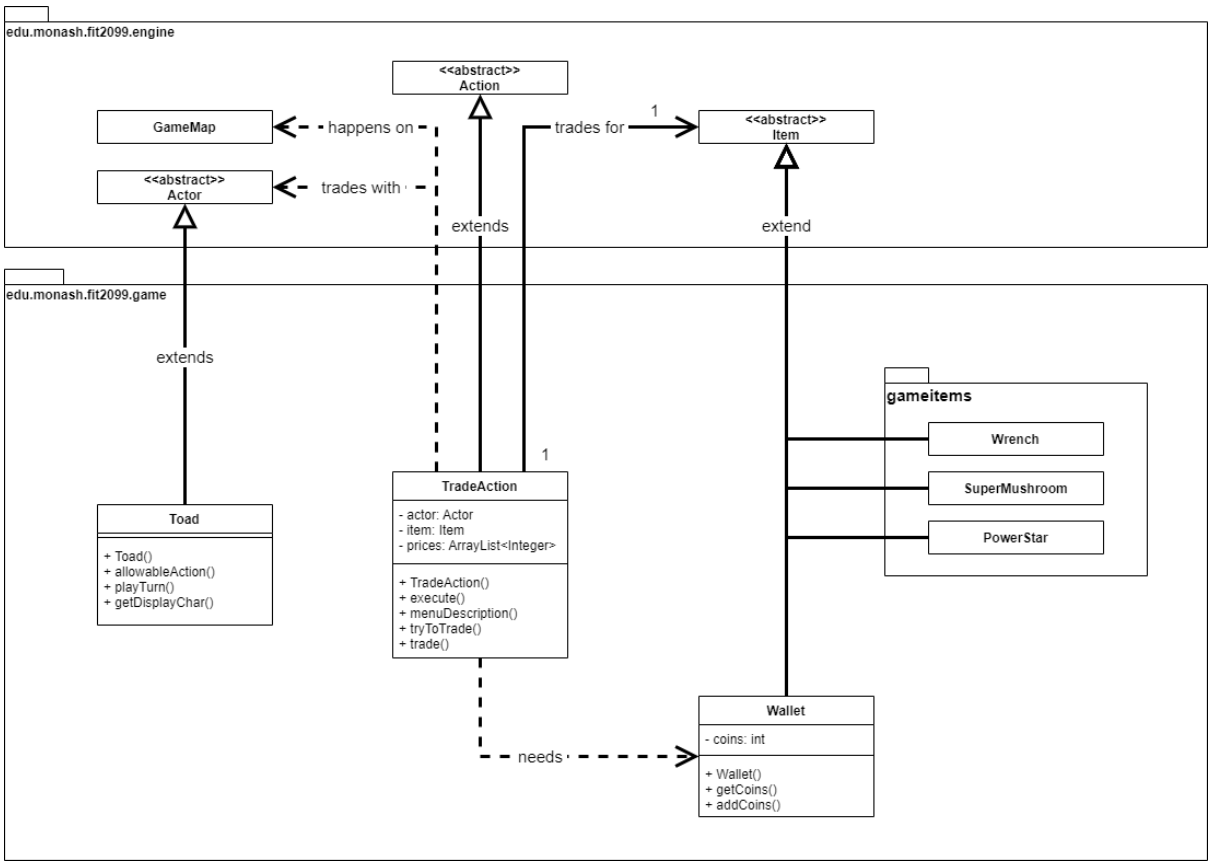
Contents

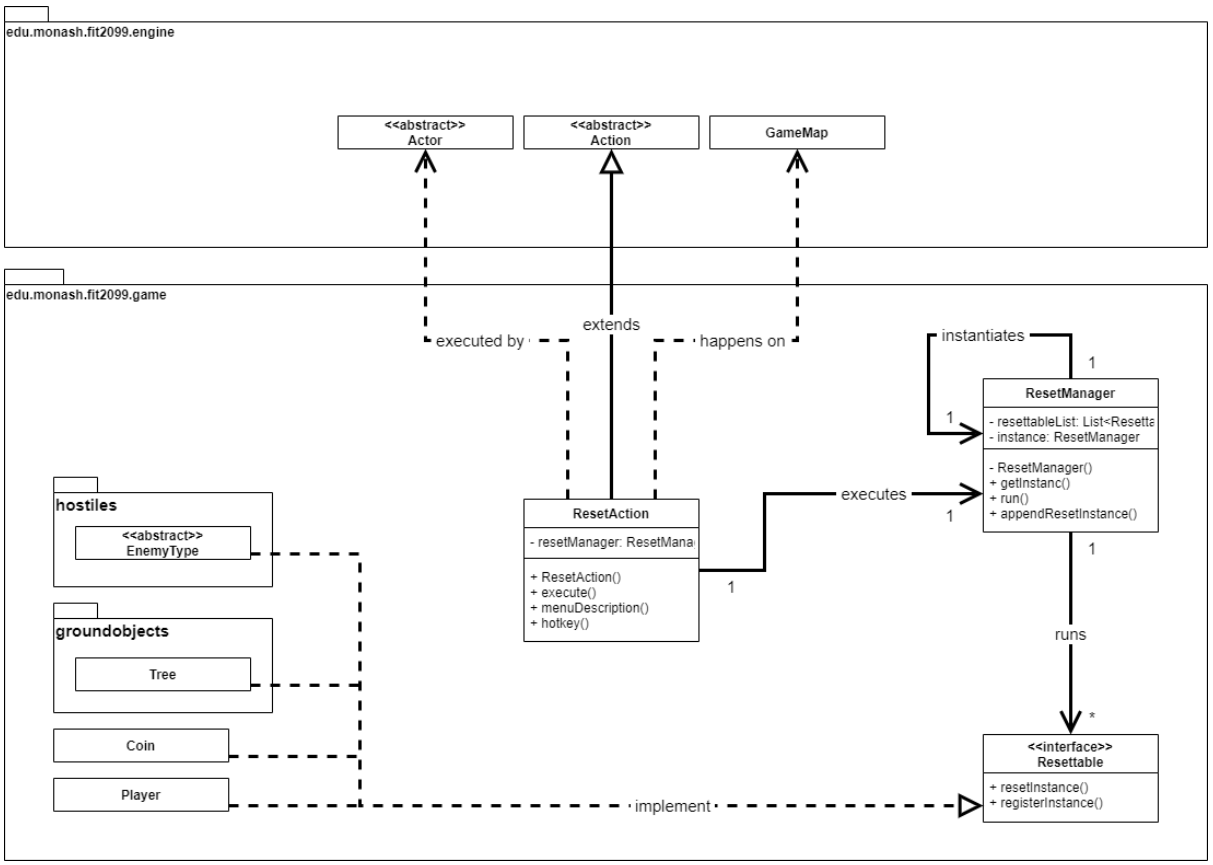

edu.monash.fit.2099.engine
PickUpItemAction
«Abstract»
Item
«Abstract»
Ground
Location
«Abstract»
Actor
extends
Extends
has
Extends
game
games.groundObjects
Application
Tree
1
Dirt
turns to
Extends
Extends
Extends
Extends
Sprout
Responsibilities
--Ten percent chance to spawn
Goomba if n actors are present
in current location
--Spawn a Sapling in location after
10 turns
matures
Sapling
Responsibilities
--Remove Sprout in current location
immediatly after being spawned
--Drops coin at 10 percent chance at
each turn
matures
Mature
Responsibilities
--Spawn a Koompa at a 15 percent
chance at every turn
-grow new sprout in surrounding
fertile ground
--Wither die and replace ground with
Dirt object
spawns
spawns
spawns
game.hostiles
«Abstract»
EnemyType
Extends
Extends
Koopa
Goomba
EmptyPackage
Coin


edu.monash.fit2099.engine
«Abstrac»
Actor
«abstract»
Action
Location
GameMap
Extends
has
uses
game
uses
JumpAction
Responsibilities
-Determine if jump perfromed on specific
object is a success or failure
-If the jump ends in a failure
reduce player health level
-Get cordinates of Higher ground
and print it
has
GroundObjects
Tree
extends
extends
extends
Sprout
Sapling
Mature


edu.monash.fit2099.engine
WeaponItem
Extends
Extends
Extends
Extends
«abstract»
Item
Wrench
«Abstract»
Actor
1
IntrinsicWeapon
1
game
Extends
game.hostiles
«Abstract»
EnemyType
Extends
Extends
1
1
Goomba
Responsibilities
--Attacks with kick that deals
10 damage and 50% hit rate
--10 percent chance to remove
itself every turn
Koompa
Responsibilities
--When defeated implement dormant like
behaviour
--Checks if the player has a Wrench
and if attacked with it drops a
Super Mushroom
--if health is zero change the displayChar
1
1
game.magicalItems
SuperMushroom
game.behaviours
1..*
1..*
«interface»
Behaviour
«realize»
«realize»
«realize»
FollowBehaviour
AttackBehaviour
WonderBehaviour
drops


edu.monash.fit2099.engine
<<absract>>
Action
Location
1
<<absract>>
Item
GameMap
<<abstract>>
Actor
1
spawns at
spawns at
1
owns by
owns by
extends
edu.monash.fit2099.game
consumes
happens on
gameitems
consumed by
1
1
1
1
1
ConsumeAction
- actor: Actor
- item: Item
+ ConsumeAction()
+ execute()
+ menuDescription()
PowerStar
- currentLocation: Location
- actor: Actor
- MAX_DURATION: int
- duration: int
+ PowerStar()
+ tick()
+ resetDuration()
+ remainingTurn()
SuperMushroom
- currentLocation: Location
- actor: Actor
+ SuperMushroom()
extend

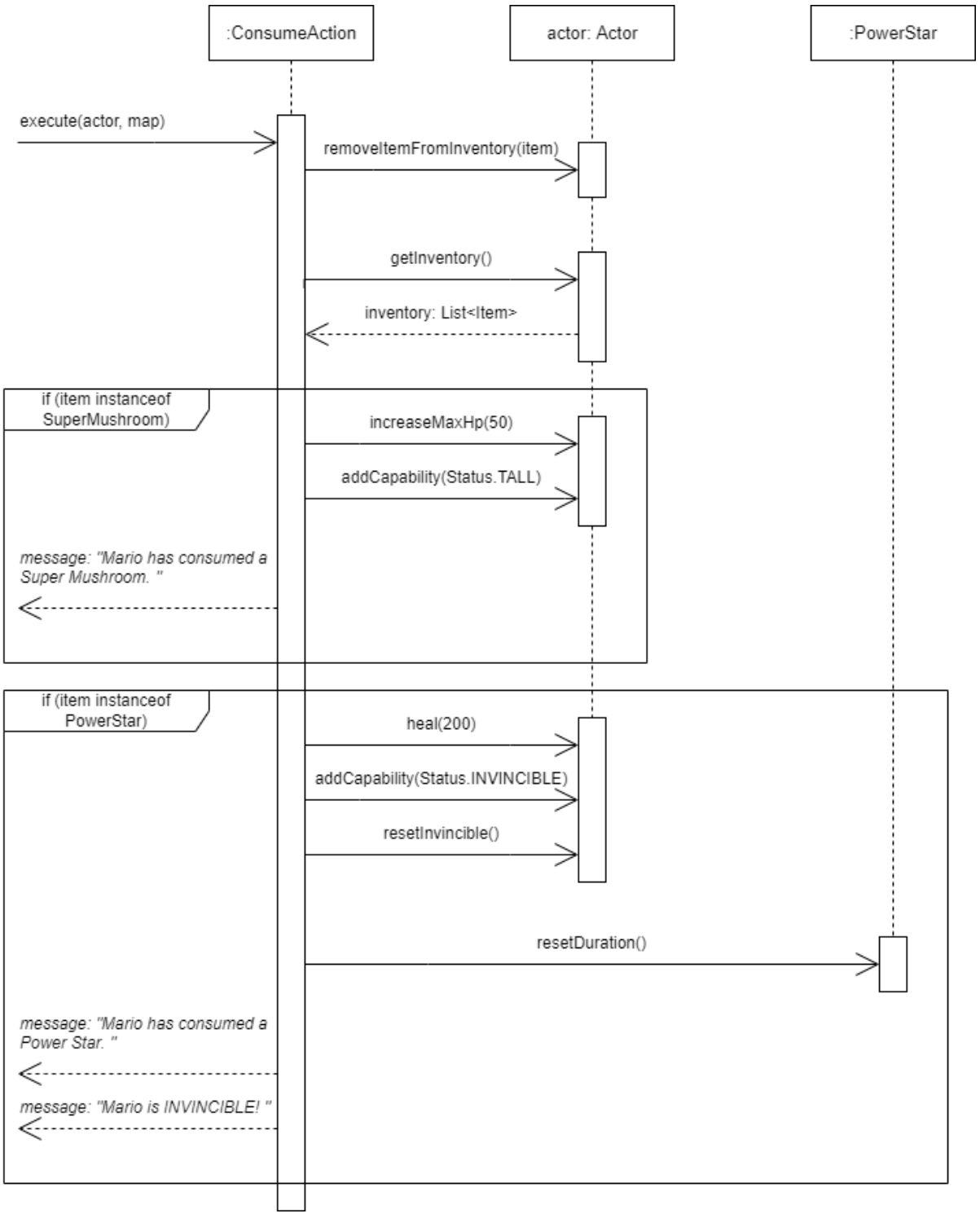

edu.monash.fit2099.engine
<<abstract>>
Action
GameMap
happens on
trades for
1
<<abstract>>
Item
<<abstract>>
Actor
trades with
extends
extend
edu.monash.fit2099.game
extends
gameitems
Wrench
SuperMushroom
PowerStar
1
TradeAction
- actor: Actor
- item: Item
- prices: ArrayList<Integer>
+ TradeAction()
+ execute()
+ menuDescription()
+ tryToTrade()
+ trade()
Toad
+ Toad()
+ allowableAction()
+ playTurn()
+ getDisplayChar()
Wallet
- coins: int
+ Wallet()
+ getCoins()
+ addCoins()
needs


edu.monash.fit2099.engine
<<abstract>>
Actor
<<abstract>>
Action
GameMap
edu.monash.fit2099.game
extends
executed by
happens on
instantiates
1
ResetManager
- resettableList: List<Resetta
- instance: ResetManager
- ResetManager()
+ getInstanc()
+ run()
+ appendResetInstance()
1
executes
1
ResetAction
- resetManager: ResetMana
+ ResetAction()
+ execute()
+ menuDescription()
+ hotkey()
1
1
runs
*
hostiles
<<abstract>>
EnemyType
groundobjects
Tree
Coin
Player
implement
<<interface>>
Resettable
+ resetInstance()
+ registerInstance()

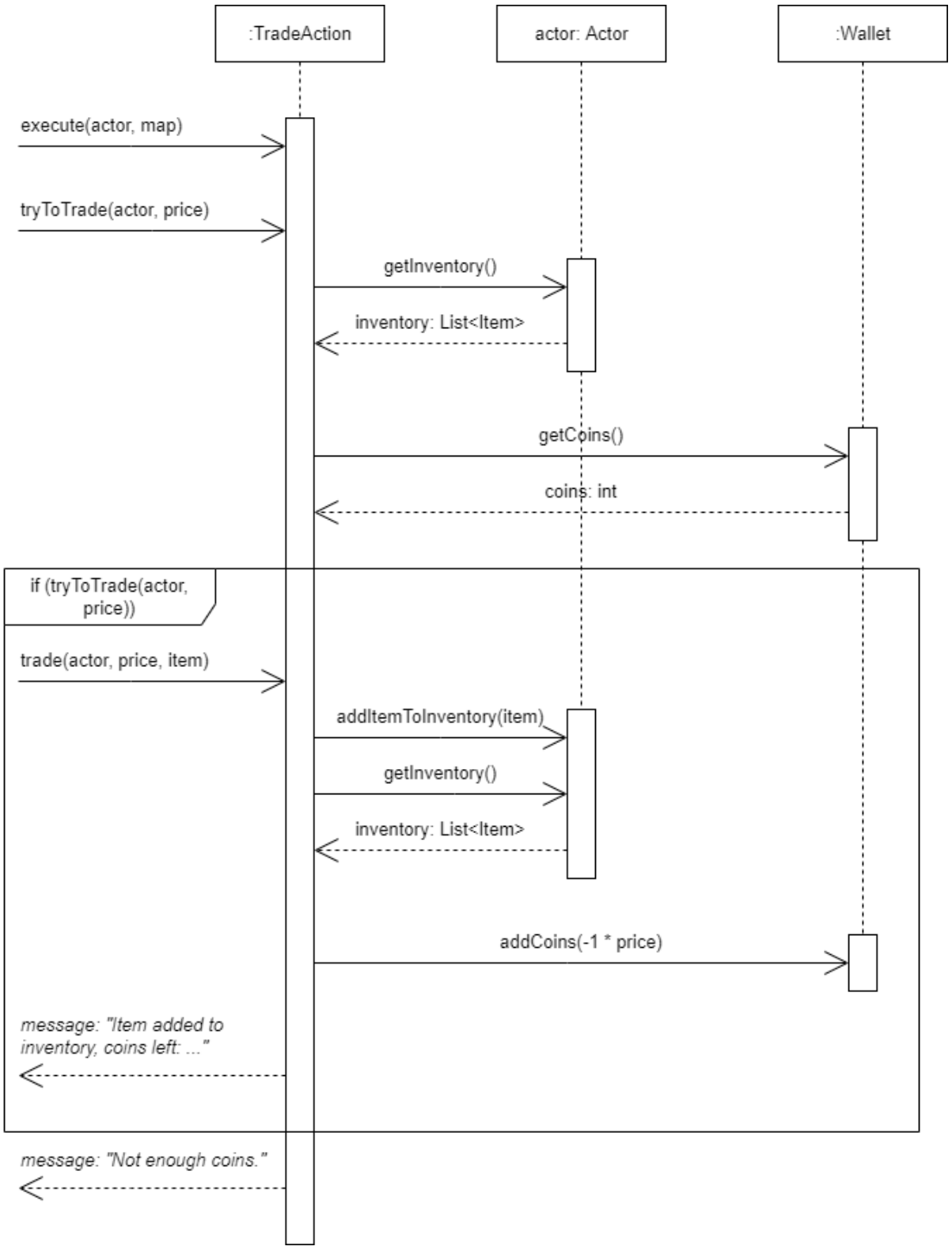
2.0 Interaction Diagrams

This part of the document will showcase the interaction diagrams for some of the complex methods in each requirement which involve at least three different classes.

2.1 Consume Action

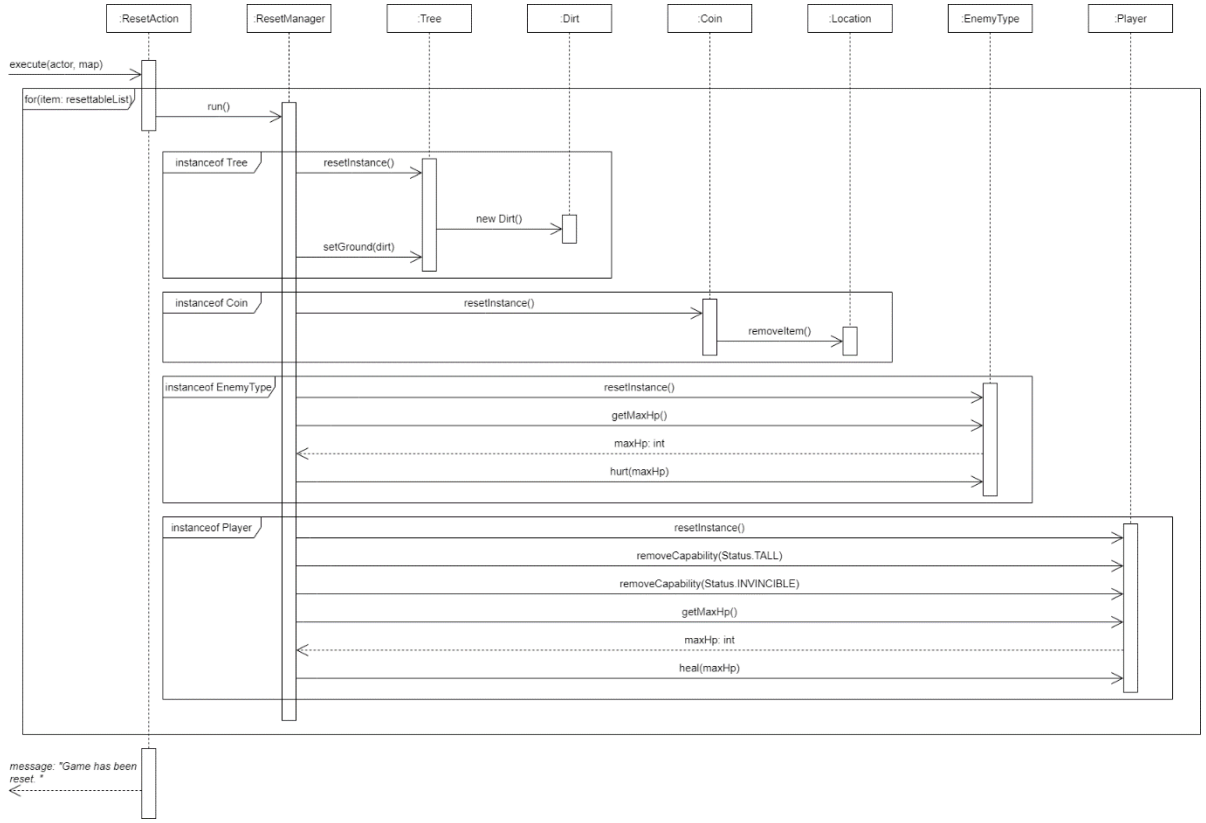


2.2 Trading Action



2.3 Reset Action

Note *: The image may appear small due to the constraints of the document layout, clarity and readability of the image may improve when zoomed in.



3.0 Design Rationale

This part of the document explores the reasoning behind why and how each requirement are tackled with object-oriented programming and the SOLID principles in mind

3.1 Let it Grow

Single Responsibility principle

Here It is stated that the tree has three states a sprout, a sapling and a mature tree and each stage has unique spawning abilities. Therefore in order to divide these unique responsibilities a Tree class

```
public Sprout() { super( displayChar: '+', treeType: "Sprout", jumpSuccessRate: 0.9, jumpFallDamage: 10); }

/**
 * Overrides the tick function of the ground object
 * This is responsible for replacing the current ground with the Sapling tree object when 10 turns are completed
 * This also spawns Goomba at the current location at a 10 percent chance
 * @param location The location of the Ground
 */
@Override
public void tick(Location location) {

    this.setLocation(location);
    //increment the tick counter and see if to spawn sprout
    if (incrementTick() == 10) {
        setGroundType(new Sapling());
    } else {
        //Spawning Goomba at 10% chance
        Random rand = new Random();
        if (rand.nextFloat() > 0.9) {
            this.addActorHere(new Goomba());
        }
    }
}
```

has been created three subclasses sprout, sapling and mature has been created so that we can prevent the tree class from having too many responsibilities. Here the Sprout subclass has been given the responsibility of spawning Goomba and the responsibility of growing into a sapling after 10 turns

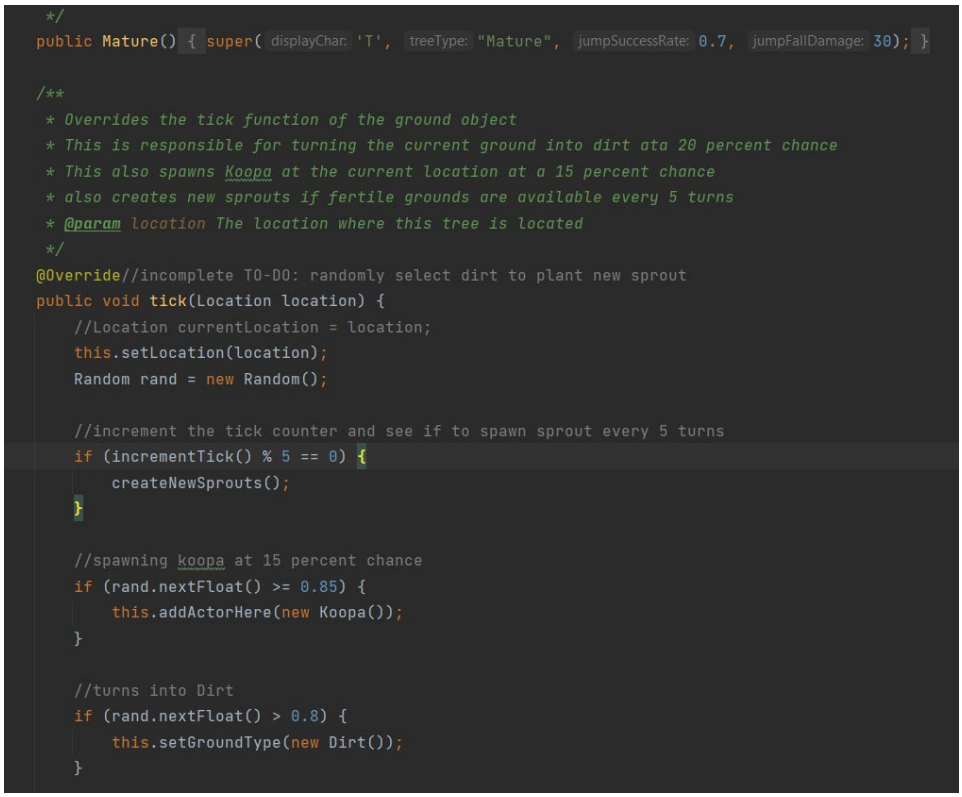
At the second stage of the life cycle of the tree which is the sapling I have provided it with the responsibility of spawning a coin and growing into a mature tree

```
public class Sapling extends Tree {

    /**
     * Constructor for the sapling object
     * creates tree object with display char:'t', name: ''sapling'', jumpSuccessRate: 80 percent, fallDamage: 20 hitpoints
     */
    public Sapling() { super( displayChar: 't', treeType: "Sapling", jumpSuccessRate: 0.8, jumpFallDamage: 20); }

    /**
     * Overrides the tick function of the ground object
     * This is responsible for replacing the current ground with the Mature tree object when 10 turns are completed
     * This also spawns a coin at the current location with a valuation of $20 every turn at a 10 percent chance
     * @param location The location of the Ground
     */
    @Override
    public void tick(Location location){
        this.setLocation(location);
        //increment the tick counter and see if to spawn Mature Tree at the 10th turn
        if (incrementTick() == 10) {
            this.setGroundType(new Mature());
        }else {
            //Spawning Coin at 10% chance
            Random rand = new Random();
            if (rand.nextFloat() > 0.9) {
                this.addItemHere(new Coin( value: 20, this.getLocation()));
            }
        }
    }
```

And finally the Mature class has been provided the responsibilities of spawning new sprouts around fertile ground, spawning Koopa disintegrating into dirt.



```
 */
public Mature() { super( displayChar: 'T',  treeType: "Mature",  jumpSuccessRate: 0.7,  jumpFallDamage: 30); }

/**
 * Overrides the tick function of the ground object
 * This is responsible for turning the current ground into dirt ata 20 percent chance
 * This also spawns Koopa at the current location at a 15 percent chance
 * also creates new sprouts if fertile grounds are available every 5 turns
 * @param location The location where this tree is located
 */
@Override//incomplete TO-DO: randomly select dirt to plant new sprout
public void tick(Location location) {
    //Location currentLocation = location;
    this.setLocation(location);
    Random rand = new Random();

    //increment the tick counter and see if to spawn sprout every 5 turns
    if (incrementTick() % 5 == 0) {
        createNewSprouts();
    }

    //spawning koopa at 15 percent chance
    if (rand.nextFloat() >= 0.85) {
        this.addActorHere(new Koopa());
    }

    //turns into Dirt
    if (rand.nextFloat() > 0.8) {
        this.setGroundType(new Dirt());
    }
```

Therefore it is clear that every tree sub class has a responsibility for a single part of the programs functionality as per the single responsibility principle.

Open-closed Principle

The open design states that classes should be open for extension but closed for modification.

In achieving this principle with requirement one it is seen that both the sprout class and the mature class have responsibilities of spawning two different types of actors such that the sprout has to spawn Goomba and the mature tree has to spawn Koopa. Instead of changing the code we can use the function addActorHere() present within the tree class to pass in both the Koopa and Gooba Objects

Liskov Substitution principle

Here all three subclasses can be replaced with each other as all these classes share all and characteristics and functions among them. Replacing one class with another class will not equal unknown behavior as the only function which is implemented by each of the three tree subclasses is the tick method() and it also uses methods already present within the abstract tree class as is evident by the images of the code placed above .

Requirement 1 How requirements have been satisfied.

1/ A tree having three lifecycle stages.

In satisfying this requirement a tree class has been created and is divided into three subclasses sapling, sprout and mature.

```java
 */
public abstract class Tree extends Ground implements Resettable {
    /**
     * the location that the tree is in
     */
    private Location location;
```

```java
public class Sapling extends Tree {

    /**
     * Constructor for the sapling object
     * creates tree object with display char:'t', name: ''sapling'', jumpSuccessRate: 80 percent, fallDamage: 20 hitpoints
     */
    public Sapling() { super( displayChar: 't', treeType: "Sapling", jumpSuccessRate: 0.8, jumpFallDamage: 20); }
```

```java
 */
public class Mature extends Tree {

    /**
     * Constructor for the Mature tree object
     * creates tree object with display char:'T', name: ''Mature'', jumpSuccessRate: 70 percent, fallDamage: 30 hitpoints
     */
    public Mature() { super( displayChar: 'T', treeType: "Mature", jumpSuccessRate: 0.7, jumpFallDamage: 30); }
```

```java
public class Sprout extends Tree {

    /**
     * Constructor for the Sprout tree object
     * creates tree object with display char:'+', name: ''Sprout'', jumpSuccessRate: 90 percent, fallDamage: 10 hitpoints
     */
    public Sprout() { super( displayChar: '+', treeType: "Sprout", jumpSuccessRate: 0.9, jumpFallDamage: 10); }
```

The sprout, sapling and mature objects are to contain display characters ’+’ , ‘t’ , and ‘T’ respectively.

AS the tree is extending the ground class the ground classes constructor contains a parameter that accepts a display char so the display char is passes in through the super constructer of the tree subclasses then the tree passes it to its ground class constructor as seen below

```java
    */
    public Tree(char displayChar, String treeType, double jumpSuccessRate, int jumpFallDamage) {
        super(displayChar);
        this.treeType = treeType;
        this.jumpFallDamage = jumpFallDamage;
        this.jumpSuccessRate = jumpSuccessRate;

        // register as a resettable instancee
        registerInstance();
    }
```

Requirements for sprout

1. **It has a 10% chance to spawn Goomba on its position in every turn. If any actor stands on it, it cannot spawn Goomba.**

The above is accomplished by first instantiating the location attribute of the tree class. by the setLocation function which is run when the tick function of the sprout class is called. This location attribute is initialized first at the first instance of the game after the player makes his first move.

```
 */
public abstract class Tree extends Ground implements Resettable {
    /**
     * the location that the tree is in
     */
    private Location location;

    /**
     * the tick counter which indicates how many times the tick method has been called
     */
    private int tickCounter = 0;
```

```
    public void setLocation(Location location) {

        this.location = location;
        this.isGroundSet = true;
    }
```

```
private boolean isGroundSet = false
```

Once the process is complete then when at every turn the tick method is called

```
@Override
public void tick(Location location) {

    this.setLocation(location);
    //increment the tick counter and see if to spawn sprout
    if (incrementTick() == 10) {
        setGroundType(new Sapling());
    } else {
        //Spawning Goomba at 10% chance
        Random rand = new Random();
        if (rand.nextFloat() > 0.9) {
            this.addActorHere(new Goomba());
        }
    }
}
```

Then at each turn using a random object I will get a random float value between 0.0 and 1.0. if the float value is greater than 0.9 which is likely to happen at a 10 percent chance at every turn I will then pass in the Goomba object to the addActorHere() function present in the tree class.

```
 */
public void addActorHere(Actor actor) {
    boolean containsActor = this.location.containsAnActor();
    if (!containsActor) {
        this.location.addActor(actor);
    }

}
```

This function will then check whether or not an actor is already present in the current location using the containsAnActor() function that is already present within the locations class and if an actor is not present it will again use a method within the location class which will place an actor onto the given location. In this case which will be Goomba.

2. It takes 10 turns to grow into a small tree/Sapling (t)

For this step I have placed a tickCounter attribute with the initial value of zero in the tree class. There is also another method that will increment this counter and return the current value of the tick counter of the tree class

```
 */
public int incrementTick(){
    tickCounter++;
    return tickCounter;
}
```

```
    private int tickCounter = 0;

    /**
     * the name of the growth stage of the tree
     */
```

```
@Override
public void tick(Location location) {

    this.setLocation(location);
    //increment the tick counter and see if to spawn sprout
    if (incrementTick() == 10) {
        setGroundType(new Sapling());
    } else {
        //Spawning Goomba at 10% chance
        Random rand = new Random();
        if (rand.nextFloat() > 0.9) {
            this.addActorHere(new Goomba());
        }
    }
}
```

In the tick method of the spout I keep on incrementing the tickcCounter of the tree class and then when the value of the tickCounter is 10. I will then set the ground type to the spaling object which the next stage of the tree life cycle

```
/**
 * Sets the ground type of the location of the tree object     *
 * @param groundType the next ground type that should set in the current location
 */
public void setGroundType(Ground groundType) { this.location.setGround(groundType); }
```

Here within the tree class once the required ground type is passed it then uses its location attribute and uses a method withinit called setGround() which will then set the ground type to the required type which is the sapling ground type.

Requirements for Sapling

1. **It has a 10% chance to produce/spawn a coin ($20) on its location at every turn. (EDIT: replace 'drop' with 'produce/spawn')**

For this requirement as coin needs to be spawned at every turn at a 10 percent chance I will again make use of the tick function present within the sapling object.

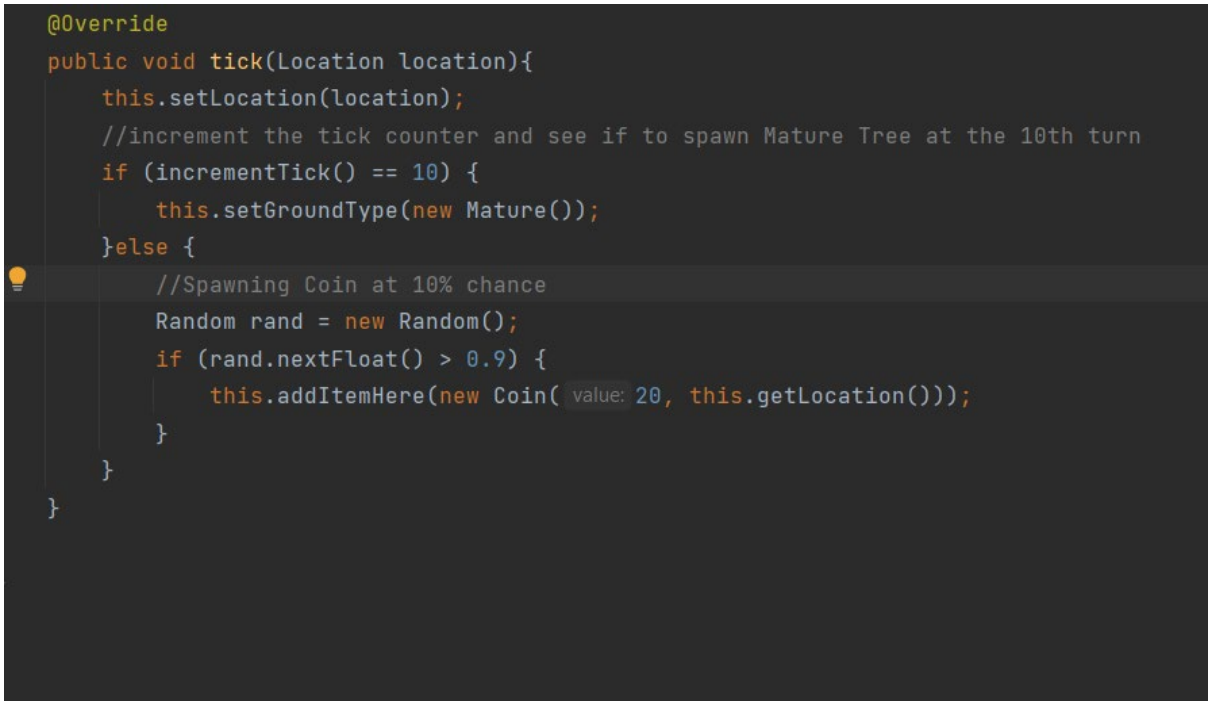
```
@Override
public void tick(Location location){
    this.setLocation(location);
    //increment the tick counter and see if to spawn Mature Tree at the 10th turn
    if (incrementTick() == 10) {
        this.setGroundType(new Mature());
    }else {
        //Spawning Coin at 10% chance
        Random rand = new Random();
        if (rand.nextFloat() > 0.9) {
            this.addItemHere(new Coin( value: 20, this.getLocation()));
        }
    }
}
```

Here first I will again cream a random object use it method to get a random float value between 0.0 and 1.0. if the value received is greater than 0.9 which happens at a 10 percent chance. I will then spawn spawn a coin within the current location using the addItemHere() method which is present within the tree class.

```
    */
public void ddItemHere(Item item){this.location.addItem(item);}
```

This method will then use the location attribute of the tree class and use a method called addItem() within it to add a coin to the current location.

2. **It takes another 10 turns to grow into a tall tree/Mature(T)**



Here again similar to the method described in the sprout class. I will use the incrementTick() method in the tree class to increment the tickCounter attribute of the tree class and get the returned value. If the returned value if equal to 10 then using the setGroundYype() method present within the Tree class I will set the current ground to a Mature tree type.

Requirements for Mature

1. It has a 15% chance to spawn Koopa in every turn. If an actor stands on it, it cannot spawn Koopa.

```
 */
@Override//incomplete TO-DO: randomly select dirt to plant new sprout
public void tick(Location location) {
    //Location currentLocation = location;
    this.setLocation(location);
    Random rand = new Random();

    //increment the tick counter and see if to spawn sprout every 5 turns
    if (incrementTick() % 5 == 0) {
        createNewSprouts();
    }

    //spawning koopa at 15 percent chance
    if (rand.nextFloat() >= 0.85) {
        this.addActorHere(new Koopa());
    }

    //turns into Dirt
    if (rand.nextFloat() > 0.8) {
        this.setGroundType(new Dirt());
    }


}
```

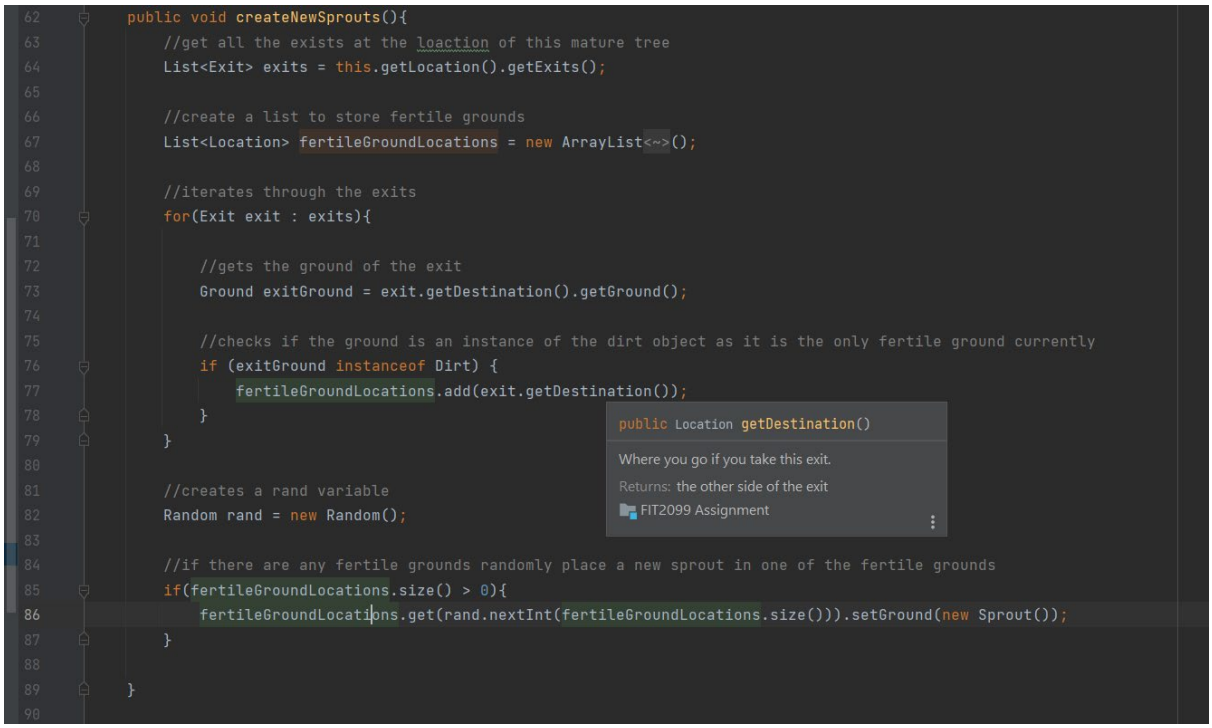
For the first requirement of spawning a Koopa at the current location. Again I will utilize the tick function that is run at each turn. Like the previous two classes I will first create a random object then using its next float value which will produce an output in the range of 0.0 and 1.0 it will then compare this and see if this value is greater than 0.85 which will happen at a 15 percent chance. If the condition is true it will then use the addActorHere() method in the tree class and spawn the Koopa at the location.

2. It has 20% to wither and die (becomes Dirt) in every turn.

To accomplish this requirement I have used the random object that has already been create to generate a float value. Then I will check if the produced float value created than 0.8 which will occur at a 20 percent chance. If this has occurred I will use the setGroundType() method of the tree class and set the current ground type of the location to the dirt type

3. For every 5 turns, It can grow a new sprout (+) in one of the surrounding fertile squares, randomly. If there is no available fertile square, it will stop growing sprouts. At the moment, the only fertile ground is Dirt. (Edit: remove 'spawn' keyword in this bullet point)

For this requirement I have created a function called createNewSprouts()

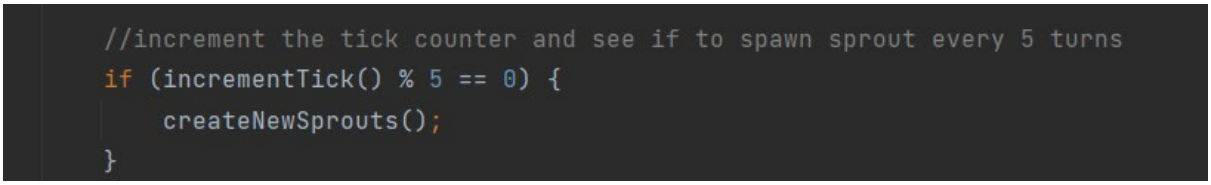


Here initially I create a list of exists which I get from the current location attribute of the tree class.

Then I proceed to initialize another list which will hold a list of fertile ground objects.

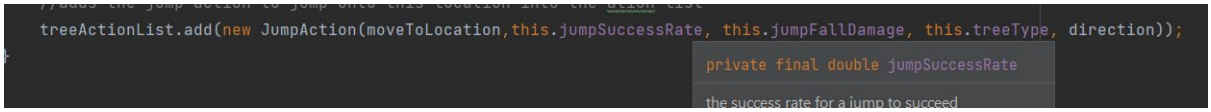
Then I get the list of exits and iterate through it and for each exit I get the ground. If the ground is of instance Dirt which currently is the only fertile land and then add it to the list of fertile grounds. Once the process is complete I then randomly select a fertile ground and then place a new sprout on it.

As it says to repeat this process for every 5 turns I first get the tickcounter and check if it is divisible by 5. If the condition is true I then call the createNewSprouts() method.



Requirement 2 Design

Here it is clear that the dry principle has been followed as the jump action can be reused for multiple high ground objects such as Walls, Sprouts, Saplings and Mature Trees as all the necessary variables required without the need to repeat code or change . As evident by the code examples below



Regarding solid principles the design of the jump action follows the Single Responsibility principle well. As the only responsibility of the jump action class is to determine whether or not a jump is successful or not and is encapsulating this specific part of the functionality. Because of this it has resulted in high cohesion of the data that is within the jump action class

Considering the design in terms of the Open Close principle as well the design is open to extension and can be used by more classes in the future as it does not take in instances of different classes to determine the success rate but only takes in concreate integer values to determine the success or failure of the jump. There no code needs to be modified for more classes to use the jumpAction class.

Requirement 2 Implementation

- **Wall: 80% success rate, 20 fall damage**
- **Tree:**
    - **Sprout(+): 90% success rate, 10 fall damage**
    - **Sapling(t): 80% success rate, 20 fall damage**

- **Mature(T): 70% success rate, 30 fall damage**

```
 *
 * @author Ravindu Santhush Ratnayake
 */

public class JumpAction extends Action {

    /**
     * the success rate of the jump
     */
    private double successRate;

    /**
     * the fall damage to be given to the actor when the jump fails
     */
    private int fallDamage;

    /**
     * the location to move the actor to if the jump fails
     */
    private Location moveToLocation;

    /**
     * the name of the jump object
     */
    private String jumpObject;

    /**
     * the direction of the location to jump from the actor
     */
    private String direction;
```

In order to implement the jump action in a very modular way. Using the DRY principles and in such a way that this can be reused and expanded efferently. All necessary attributes such as successRate, fallDamage, moveToLocation, jumpObject and the direction have been created.

Using the constructor to the class all these attributes will be initialized.

After that when executing this function I will determine whether a Super Mushroom has been consumed. IF its has been consumed then I will immediately move the actor to the location.

```
Random rand = new Random();
// if actor is TALL, it means it has consumed a Super Mushroom
// therefore it'll have a 100% success rate at jumping
if (actor.hasCapability(Status.INVINCIBLE)) {
    // can walk normally
    map.moveActor(actor, moveToLocation);

    // destroy and replace higher ground with dirt
```

Else if no mushroom has been consumed I will generate a random integer value and compare to the success rate to determine if the jump is a success or failure.

```
    } else {

        //compares success rate to randomly generated float value
        if (rand.nextFloat() < successRate) {

            //if the if-condition is passed then the actor is moved to the required destination
            map.moveActor(actor, moveToLocation);
            return actor + " successfully jumped on " + jumpObject + "(" + moveToLocation.x() + "," + moveToLocation.y() + ")";

        } else {
            //if the jump failed actors health is reduced
            actor.hurt(fallDamage);
            return "Jump failed " + actor + " lost " + fallDamage + " in health";
        }
    }
}
```

If it a success based on the success rate I will move the player onto the location else I will reduce the players health by the fall damage attribute as given by the requirements.

Also no other actor will be able to enter to the high grounds as in the tree class and wall objects as the canActorEnter() method is overridden and always returns false

```
/**
 * returns false so that actors can only jump to the location but not walk here
 * @param actor unused actor object
 * @return returns false value always
 */
@Override
public boolean canActorEnter(Actor actor) { return false; }

/**
```

Also only the player can jump as the tree and wall objects check if the actor requesting the allowable action has the capability to jump(CAN_JUMP)

3.2 Jump Up, Super Star! **Single Responsibility Principle**

Here as per the single responsibility principle two enemyType subclass objects have been created due to them having specific requires such as the goomba needing to be remove itself from the map to prevent clutter within the system and the koopa needing to check if the player is attacking it with a wrench and to implement a dormant like state once it has been defeated.

```
@Override
public Action playTurn(ActionList actions, Action lastAction, GameMap map, Display display) {

    //self destruct
    Random rand = new Random();

    if(rand.nextInt( bound: 10) > 8){
        this.hurt(this.getMaxHp());
        return new DoNothingAction();
    }else{
        return this.returnCurrentTurnAction( actor: this, map);
    }
```

```
 */
@Override
public void hurt(int points) {
    //checks if koopa is not dormant
    if (!isDormant) {
        super.hurt(points);
        // if the koopa health is below zero sets display character to D
        if (!super.isConscious()) {
            super.setDisplayChar('D');
            isDormant = true;
            //clears all behaviours from koopa behaviours so that he does nothing
            this.getBehaviours().clear();
        }
        //if koopa is currenly dormant and if attaacked then it dies
    } else if (isDormant) {
        isDead = true;

    }

}
```

Liskov Substitution principle

In following the liskov substitution principle the Koopa and Goomba objects can be substituted with each other and the enemyType class as no destructive behaviour will arise as all methods and functions as all functions return and take in identical inputs and outputs

For example here

```
 */
@Override
protected IntrinsicWeapon getIntrinsicWeapon() { return new IntrinsicWeapon( damage: 30,  verb: "punches");
```

```
@Override
protected IntrinsicWeapon getIntrinsicWeapon() { return new IntrinsicWeapon( damage: 10,  verb: "kicks"); }

/**
```

If we were to substitute each other and call these functions both will return an intrinsic item and unknown behavior will not occur.

Interface Segregation Principle

Here all behaviors are divided into are split into smaller parts. Such as the Wander Behavior, Follow Behavior, and Attack behavior and these behaviors are kept inside a hash map as shown below

```
/**
 * hashmap used to contain behaviours with priority key values
 */
private final Map<Integer, Behaviour> behaviours = new HashMap<>();
```

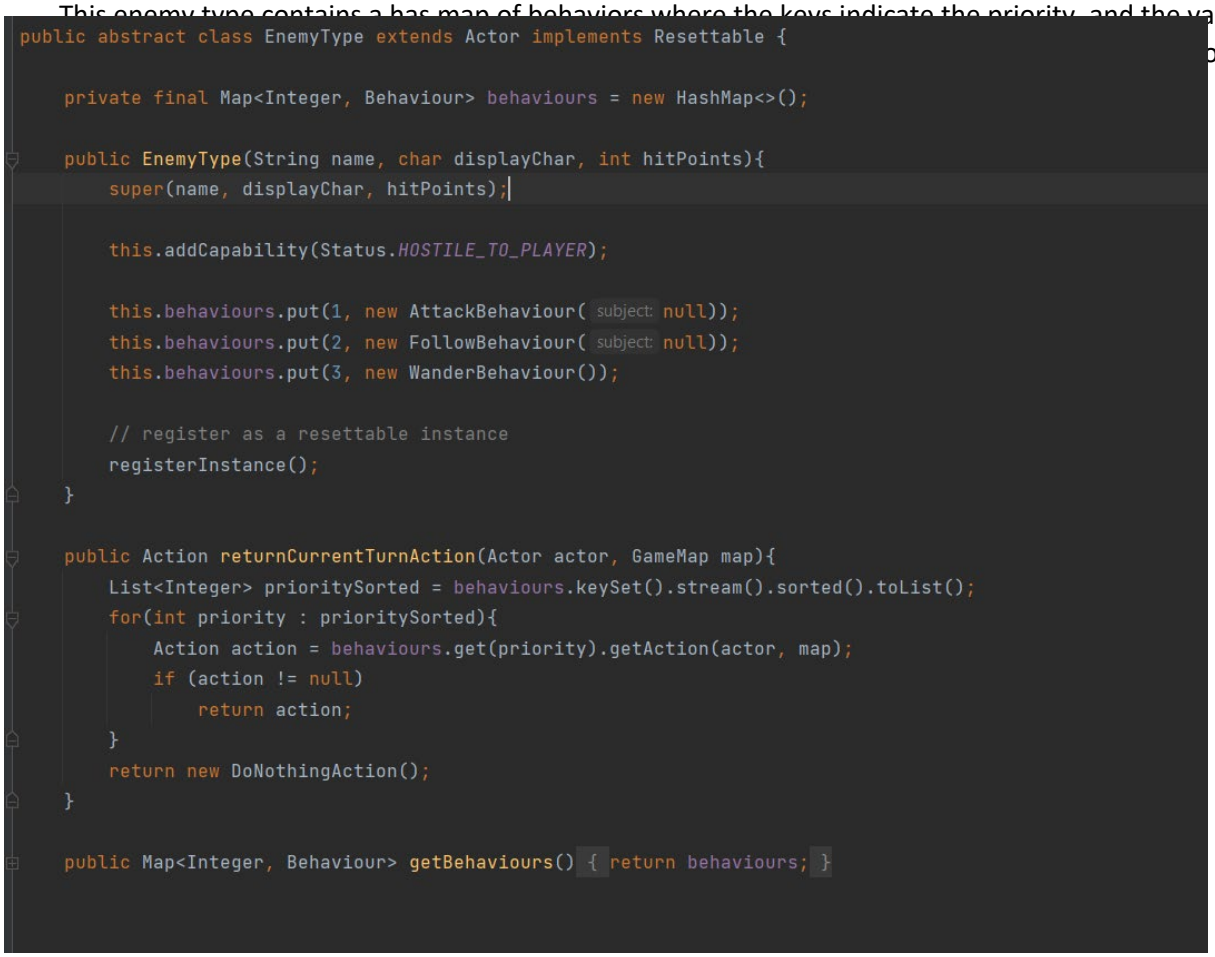
Therefore in the future as these behaviors are separate we can make sure that these enemy classes only implement the interfaces and behaviors that it necessary to be used my the specific class and that other behaviors may be discarde**d.**

3.3 Enemies

Requirement 3 Implementation

Here I have created an abstract class called Enemy type. With a capability called HOSITLE_TO_PLAYER

This enemy type contains a has map of behaviors where the keys indicate the priority and the values



```
public abstract class EnemyType extends Actor implements Resettable {

    private final Map<Integer, Behaviour> behaviours = new HashMap<>();

    public EnemyType(String name, char displayChar, int hitPoints){
        super(name, displayChar, hitPoints);

        this.addCapability(Status.HOSTILE_TO_PLAYER);

        this.behaviours.put(1, new AttackBehaviour( subject: null));
        this.behaviours.put(2, new FollowBehaviour( subject: null));
        this.behaviours.put(3, new WanderBehaviour());

        // register as a resettable instance
        registerInstance();
    }

    public Action returnCurrentTurnAction(Actor actor, GameMap map){
        List<Integer> prioritySorted = behaviours.keySet().stream().sorted().toList();
        for(int priority : prioritySorted){
            Action action = behaviours.get(priority).getAction(actor, map);
            if (action != null)
                return action;
        }
        return new DoNothingAction();
    }

    public Map<Integer, Behaviour> getBehaviours() { return behaviours; }
```

be the Attack behavior indicated with a priority of one followed by the follow behavior and finally with the behavior with the least priority which is the wander behavior.

This enemy abstract class will contain a function called returnCurrentTurnAction(which take in parameters for an actor and a game map.). In this function it will first create a list of keys based on the priority of the behaviors in ascending order. Then I will loop through these keys and get the behavior according to the priority. One a behavior is fetched I will use it to obtain the specific action. If the action returned is null the program will then iterate to the next behavior until an action is returned. If none of the behaviors return an action then a DoNothingAction will be returned.

This enemy abstract class also overrides the allowableActions() of the Actor class. It will first check if the actor requesting the actions contains a status which indicates that it is hostile to the enemy. If this is the case then it will return an action that will allow the actor to attack the enemy object. It will also replace the hash map behaviors with new behaviors with the difference being that these behaviors

have the hostile actor passed into them such that when the turn of the enemyType arrives it can target this hostile actor

```java
public Map<Integer, Behaviour> getBehaviours() { return behaviours; }



@Override
public ActionList allowableActions(Actor otherActor, String direction, GameMap map) {
    ActionList actions = new ActionList();
    // it can be attacked only by the HOSTILE opponent, and this action will not attack the HOSTILE enemy back.
    if(otherActor.hasCapability(Status.HOSTILE_TO_ENEMY)) {
        actions.add(new AttackAction( target: this,direction));
        this.behaviours.put(1, new AttackBehaviour(otherActor));
        this.behaviours.put(2, new FollowBehaviour(otherActor));

    }
    return actions;

}

@Override
public void resetInstance() {
    int maxHp = this.getMaxHp();
    this.hurt(maxHp);
}

@Override
public void registerInstance() { Resettable.super.registerInstance(); }
```

The Attack Behavior

```java
public class AttackBehaviour implements Behaviour {

    /**
     * the target actor in which the attack should be directed against
     */
    private final Actor target;

    /**
     * constructor
     * @param subject the target actor in which the attack should be directed against
     */
    public AttackBehaviour(Actor subject) {this.target = subject; }

    /**     *
     * @param actor the Actor acting
     * @param map the GameMap containing the Actor
     * @return
     */
    @Override
    public Action getAction(Actor actor, GameMap map) {
        //ArrayList<Action> actions = new ArrayList<Action>();

        for (Exit exit : map.locationOf(actor).getExits()) {
            Location location = exit.getDestination();
            if (location.containsAnActor()) {
                if(location.getActor().hasCapability(Status.HOSTILE_TO_ENEMY)){
                    return new AttackAction(target, exit.getName());
                }
            }
        }
        return null;
```

The attack behavior is a class which implements the behavior. It has an attribute called other actor where the attack is directed against. Once setup. The attack behavior

get the exists from the current location provided. It then loops through these and checks if the player is in one of these exits. IF the player is found then an attack action is returned else only a null value is returned.

Requirements For Goomba

- **It starts with 20 HP**

Here the Goomba is a class which extends the EnemyType class which itself extends the Actor class. Therefore in the constructor I pass in the hitpoints parameter of 20

```
*/
public Goomba() {
    super( name: "Goomba",  displayChar: 'g',  hitPoints: 20);
    // register as a resettable instance
}
```

- **It attacks with a kick that deals 10 damage with 50% hit rate.**

  For this requirement I have overridden the intrinsic weapon method which will return an intrinsic weapon with 10 damage with the er ‘kick’. The hitrate of the Intrinsic weapon is

```
@Override
protected IntrinsicWeapon getIntrinsicWeapon() { return new IntrinsicWeapon( damage: 10,  verb: "kicks"); }

/**
```

  always set to 50 percent

- **In every turn, it has a 10% chance to be removed from the map (suicide). The main purpose is to clean up the map.**

  To achive the goomba suicide I have created a random num in the playTurn method. It first creates a random object. Then using this object it produces a number between 0 and 9(inclusive). I then check if this number is greater than 8. If it true then the maximum health value of the goomba is retrieved and then substrateted from its total health which will result in the goomba dying.

```
@Override
public Action playTurn(ActionList actions, Action lastAction, GameMap map, Display display) {

    //self destruct
    Random rand = new Random();

    if(rand.nextInt( bound: 10) > 8){
        this.hurt(this.getMaxHp());
        return new DoNothingAction();
    }else{
        return this.returnCurrentTurnAction( actor: this, map);
    }
```

Requirements for Koopa

- **It starts with 100 HP**

Here similar to Goomba the Koopa Class also extends the EnemyType class which itself extends the Actor class. Therefore in the constructor I pass in the hitpoints parameter of 100hp

```
 */
public Koopa() {
    super( name: "Koopa",  displayChar: 'K',  hitPoints: 100);
    // register as a resettable instance
}
```

- **<u>EDIT: It attacks with a punch that deals 30 damage with a 50% hit rate.</u>**

  For this requirement I have overridden the intrinsic weapon method which will return an intrinsic weapon with 30 damage with the verb 'punch'. The hitrate of the Intrinsic weapon is

```
 */
@Override
protected IntrinsicWeapon getIntrinsicWeapon() { return new IntrinsicWeapon( damage: 30,  verb: "punches");
```

- **When defeated, it will not be removed from the map. Instead, it will go to a dormant state (D) and stay on the ground (cannot attack nor move).**

  For this I have two attributes in Koopa one to check if Dead and the other to check if it is in a dormant state

```
    /**
     * variable that checks if the Koopa is in an dormant state
     */
    boolean isDormant = false;

    /**
     * variable that checks if the koopa is dead
     */
    boolean isDead = false;

    /**
     * Constructor.
     */
```

Every time the Koopa is hit I will call the super function of isConcious in the actor class to check if koopas health is less that zero. If true then the flag isDormant will be set to true. And the display character will be set to 'D' then all the behaviors of Koopa will be cleared so that will not do anything.

```
 */
@Override
public void hurt(int points) {
    //checks if koopa is not dormant
    if (!isDormant) {
        super.hurt(points);
        // if the koopa health is below zero sets display character to D
        if (!super.isConscious()) {
            super.setDisplayChar('D');
            isDormant = true;
            //clears all behaviours from koopa behaviours so that he does nothing
            this.getBehaviours().clear();
        }
        //if koopa is currenly dormant and if attaacked then it dies
    } else if (isDormant) {
        isDead = true;

    }

}
```

- **Mario needs a Wrench (80% hit rate and 50 damage), the only weapon to destroy Koopa's shell**

For this requirement in the overridden allowableActions function. I will first use the isDormant flag to check if the Koopa is dormant then if it dormant I will again check if the player has the capability HAS_WRENCH which indicates that the player has a wrench with him. Id any of these conditions are true then not allowable action will be returned. If conditions are true then an attack action will be returned.

```
@Override
public ActionList allowableActions(Actor otherActor, String direction, GameMap map) {
    ActionList actions = new ActionList();
    // it can be attacked only by the HOSTILE opponent, and this action will not attack the HOSTILE enemy back.
    if (otherActor.hasCapability(Status.HOSTILE_TO_ENEMY)) {
        //makes sure koopa is not dormant
        if (!isDormant) {
            actions.add(new AttackAction( target: this, direction));
            this.getBehaviours().put(1, new AttackBehaviour(otherActor));
            this.getBehaviours().put(2, new FollowBehaviour(otherActor));
            // if the koopa is dormant and the player has a wrench return attack action
        } else if (isDormant & otherActor.hasCapability(Status.HAS_WRENCH)) {
            actions.add(new AttackAction( target: this, direction));
        }
    }
    return actions;
}
```

Then in the hurt function it will deal the final blow and kill Goomba and set the isDead flag to true which will be checked in the overridden isConcious function that check if the Koopa is still alive

```
 */
@Override
public void hurt(int points) {
    //checks if koopa is not dormant
    if (!isDormant) {
        super.hurt(points);
        // if the koopa health is below zero sets display character to D
        if (!super.isConscious()) {
            super.setDisplayChar('D');
            isDormant = true;
            //clears all behaviours from koopa behaviours so that he does nothing
            this.getBehaviours().clear();
        }
        //if koopa is currenly dormant and if attaacked then it dies
    } else if (isDormant) {
        isDead = true;

    }

}
```

```
    /**
     * checks if Koopa is conscious
     * @return
     */
    @Override
    public boolean isConscious() {
        //checks if health is not zero
        if (super.isConscious()) {
            return true;
        } else { //if health is zero checks if shell is broken
            if (isDead) {
                //if shell is broken retutn true
                return false;
            }
            return true;
        }
    }
```

- **Destroying its shell will drop a Super Mushroom.**

  Finally once the shell is destroyed I will use the gameMap class to get the location of the current Koopa object and spawn a super mushroom in its location

3.4 Magical Items

Requirement 4 states that there should be two magical items that the player can consume for a temporary power up. The magical items, Super Mushroom and Power Star can be picked up, dropped, traded, as well as consumed.

Two respective classes (*SuperMushroom* and *PowerStar*) will be constructed for each magical item, as recommended in the SOLID design principles, this will ensure that each of the two classes will have only **single responsibility**. That is to say that each class will only be responsible for the methods that may apply to instances of its class, and changes in one class will not affect the other. As both of the items have a pretty large overlap with the existing *Item* class, the two classes will therefore extend the *Item* class. In accordance with **Liskov's substitution principle**, the two classes will also be able to call the methods for picking up and dropping items defined in the *Item* class.

Both magical items are associated with the *Actor* class, and the *Location* class. This stores the metadata of which instance(s) of the *Actor* class has a hold of the magical items, or which *Location* the items may spawn at.

This leaves only the consume action, and the trading action (which will be discussed in the next requirement) to be implemented. In order to implement the consume action, the *ConsumeAction* class is constructed, extending the existing *Action* class. This class acts as an abstraction to avoid the two magical items' classes (*SuperMushroom* and *PowerStar*) being directly dependent on the *Action* class. Thus, taking clues from the **dependency inversion principle**, and avoiding having to alter the *Action* class to perform these actions.

Executing the *ConsumeAction* would bring about different results depending on what the item consumed was. If the item consumed is an instance of the *SuperMushroom* class, the player will be given the capability "*TALL*" in the *Status* enumeration, as well as having their maximum hit points increased by 50. The *TALL* capability will allow the player to be able to jump to higher grounds with a 100% success rate, as well as having the display character be upper case. Using the enumeration to implement the *TALL* capability allows the code to follow the **open-closed principle**, where the results of *JumpAction* and display character could be changed without directly altering the methods.

3.5 Trading

Requirement 5 enables the player to trade for some of the in-game items, which include the Wrench, Super Mushroom, and the Power Star, which are separate classes (*Wrench*, *SuperMushroom*, and *PowerStar*), each extending the abstact *Item* class while implementing the methods in *Item* class as recommended in **Liskov's substitution principle**. These items are to be traded with the non-player character (NPC) Toad.

In order as to not breach the **single responsibility principle**, a separate class, namely the *Toad* class is created, extending the *Actor* class. This class is given its unique display character 'O' to represent the NPC as in the game and can implement the methods defined in the parent class. When an instance of the *Player* class (essentially, this is the character the user will be controlling) is in proximity of an instance of the *Toad* class, the option to trade for the above-mentioned items will appear in the menu.

This segues into the trade action that was briefly discussed in Requirement 4, similarly to the *ConsumeAction* class created there, a *TradeAction* class is created to facilitate the trading between the player and Toad. This applies the same SOLID design principle: the **dependency inversion principle** in order to bridge the actors and items involved from the *Action* class.

A *TradeAction* instance accepts two parameters, namely the actor making the trade, and the item that is being sought after. When the action is executed, the price of the item is determined based on whether it is an instance of *Wrench*, *SuperMushroom*, or *PowerStar*. Thereafter, the method will check for the validity of the trade, ensuring that the player has enough coins to make the purchase by retrieving the player's wallet from their inventory.

When coins are picked up by the player, rather than storing the *Coin* instances into the player's inventory, a *Wallet* class instance keeps tab of the player's wealth instead. This once again implements the **dependency inversion principle** by introducing a *Wallet* class as a centralized entity between the *Coin* and *TradeAction* classes. Of course, the combined value of coins picked up will be a private attribute to the *Wallet* instance and can only be retrieved by using its public accessor to ensure abstraction.

Once the validity of the trade is ensured, the item traded for will be added to the actor's inventory, meanwhile the player's wealth stored in the *Wallet* instance will be deducted based on the item's price.

3.6 Reset Game

Requirement 7 introduces a way to make the game more manageable and playable some time after it has been running to clear up the clutter by getting rid of some of the progress that has happened as the game was running.

As there are multiple things that will be affected once the game is reset, it is easier to implement this requirement using an interface *Resettable*. This is also done in a way that complies with the **interface segregation principle** where all of the classes that implements the *Resettable* interface make use of all of the methods. Each class that implements the interface will also implement the two methods:

```
public void resetInstance()

public void registerInstance()
```

The `registerInstance()` method registers the classes to a list managed by the *ResetManager* class. On the other hand, the `resetInstance()` method defines what actually happens when the class instance gets reset. For example, an instance of the *Player* class will be healed up their maximum hit points, while having any effects from all magical items removed. Whereas the classes that extends the abstract *EnemyType* class will be killed by inflicting a damage that is equal to the individual maximum hit points.

With all that said, the actual implementation of resetting the game won't be complete without having a way to execute it, which is where the *ResetAction* class comes in. It carries a similar role to the *ConsumeAction* in Requirement 4, and the *TradeAction* in Requirement 5, where it follows the **dependency inversion principle** in creating an abstraction between the *Resettable* instances and the *Action* class.

The *ResetAction* can only be executed once in the entire run of the game, however, so it differs slightly from *ConsumeAction* and *TradeAction* in this sense. This is ensured by introducing a Boolean attribute in the *Player* instance to check if the action has been executed and will not provide the player the option to reset the game after it has been reset once. This Boolean value is updated when the `resetInstance()` method is called.