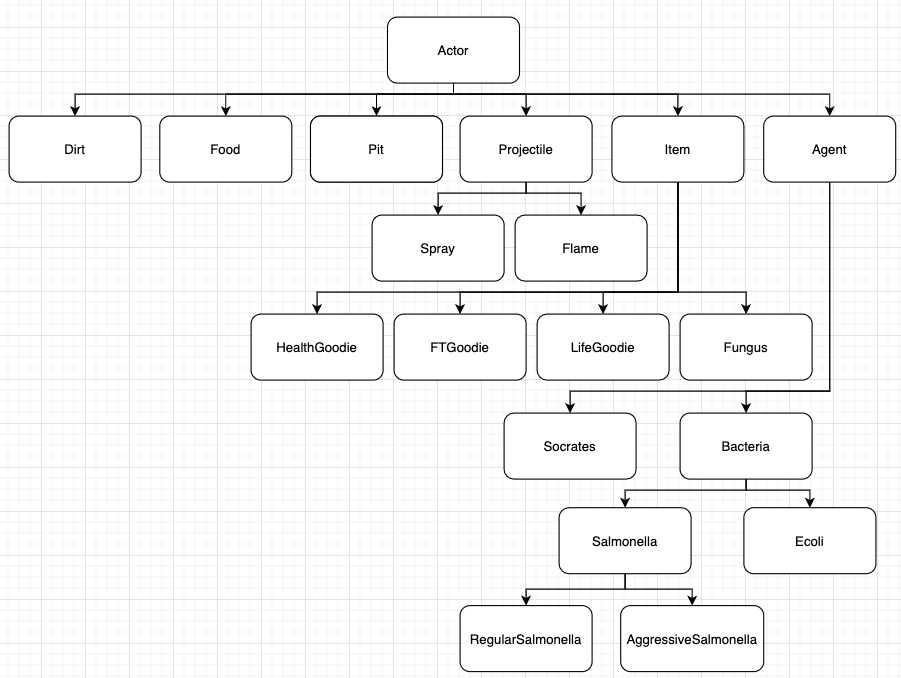
Neev Levy

Project 3 Report

**Class Hierarchy Diagram for Actor:**



\*\*All functions labeled const are not allowed to modify data members of the class\*\*

\*\*No constructor is virtual because they cannot be\*\*

**1. Classes and their Public Member Functions:**

StudentWorld

* StudentWorld(std::string assetPath)
  + Given by the spec.
* ~StudentWorld()
  + StudentWorld needs a destructor because it must delete all of the dynamically allocated data members it creates over the course of the game.
* **virtual** **int** init(), **virtual** **int** move(), **virtual** **void** cleanUp()
  + All three functions were given as virtual by the spec. Although this project had no derived classes coming from the StudentWorld, a user might like to make one in the future to create a sort of mini game with possibly different rules.
* **void** addActor(Actor\* newActor)
  + Actor interactions that introduce new actors into the game may call this function through their StudentWorld to do so.
* Socrates\* player() **const**
  + Actors that need access to the player (i.e. to inflict damage or to access the player’s current coordinates) can call this function through their StudentWorld to do so.
* **bool** isOverlap(Actor\* actor1, Actor\* actor2, **double** radius) **const**
  + Checks to see if two given actors are currently overlapping in the game, given a particular radius.
* **void** getOverlap(Actor\* actor, list<Actor\*>& actorsThatOverlap, **double** radius)
  + Allows actors to access all of the other actors that it currently overlaps with in the game, given a particular radius.
* **void** decreasePits()
  + StudentWorld must keep track of the number of pits currently in the game. When a pit deactivates itself, it calls this function through its StudentWorld to indicate that there is one less active pit in the current level.

Actor

* Actor(StudentWorld\* studentWorld, **int** objectType, **int** imageID, **double** startX, **double** startY, Direction dir = 0, **int** depth = 0, **double** size = 1.0)
  + Constructor for an actor; cannot be virtual.
* **virtual** ~Actor() {}
  + As a base class, Actor must have a virtual destructor, even if it does nothing.
* **virtual** **void** doSomething() = 0
  + This function is pure virtual because an Actor cannot be told to do something. It would seemingly do nothing, which is the same as not doing something. Having a pure virtual function also specifies Actor as an abstract class, meaning an object cannot be declared as merely an Actor. The Actor class is acting as only a base class.
* StudentWorld\* studentWorld() **const**
  + Every actor must be able to access the StudentWorld it is living in. This function is the same for every actor, so by choosing to not make it virtual, no actor can override the function.
* **int** objectType() **const**
  + Every actor is constructed with an integer that indicates its most specific class type. This function is not virtual because it is the same for every one of Actor’s derived classes.
* **bool** isActive() **const**
  + Every actor has an active or deactivated status. This function allows an actor to access what activity state it is in. It works the same for every derived class, so it is not virtual
* **void** deactivate()
  + Every actor is able to deactivate itself under certain circumstances. The function does the same thing for every one of Actor’s derived classes, so it is not virtual.
* **virtual** **void** takeDamage(**int** amount)
  + Every actor is able to take damage. However, certain actors (agents) have a number of hit points, whereas other actors will die after being damaged once. For the majority of actors, a call to takeDamage(int amount) will simply deactivate the actor. The function is defined as virtual so that agents are able to override the function, and specify that they may not always deactivate after taking damage.

Dirt

* Dirt(StudentWorld\* studentWorld, **double** startX, **double** startY)
  + Dirt has a constructor. It does nothing else specific to itself. Its interactions with other particular actors are managed by said actors.

Food

* Food(StudentWorld\* studentWorld, **double** startX, **double** startY)
  + Food has a constructor. It does nothing else specific to itself. Its interactions with other particular actors are managed by said actors.

Pit

* Pit(StudentWorld\* studentWorld, **double** startX, **double** startY)
  + Pit has a constructor that initializes its location in the petri dish.
* **bool** isEmpty() **const**
  + Checks to see if the pit has run out of bacteria. It is not virtual because Pit is not a base class.
* **void** doSomething()
  + The doSomething() call for pit needs to be overridden because it actually does things during each tick of the game.

Projectile

* Projectile(StudentWorld\* studentWorld, **int** objectType, **int** imageID, **double** startX, **double** startY, Direction dir, **int** maximumTravelDistance, **int** damage)
  + Each projectile is constructed with a unique location, direction, maximumTravelDistance, and damage.
* **virtual** ~Projectile() {}
  + As a base class, Projectile must have a virtual destructor, even if it does nothing.
* **void** doSomething()
  + Not a virtual function because every projectile performs the same basic actions during a tick. The only differences in the type of projectiles are the values for their data members, like their damage output and maximum travel distance, which are taken care of in their constructors.

Spray

* Spray(StudentWorld\* studentWorld, **double** startX, **double** startY, Direction dir)
  + Constructor for a Spray object; needs the location and direction. Damage and maximum travel distance are known fixed values for a Spray object.

Flame

* Flame(StudentWorld\* studentWorld, **double** startX, **double** startY, Direction dir)
  + Constructor for a Flame object; needs the location and direction. Damage and maximum travel distance are known fixed values for a Flame object.

Item

* Item(StudentWorld\* studentWorld, **int** objectType, **int** imageID, **double** startX, **double** startY, **int** ScoreChange, **bool** hasSound);
  + Constructor for an Item object.
* **virtual** ~Item() {}
  + As a base class, Item must have a virtual destructor, even if it does nothing.
* **void** doSomething()
  + The basic actions performed during any type of Item’s doSomething() function is the same. The only differences are in their interaction with the player, which are specified in their playerInteraction() function.
* **virtual** **void** playerInteraction() = 0;
  + All items have a unique interaction with the player. This function is declared as pure virtual so that each class derived from item must have its own playerInteraction() function. Also specifies Item as an abstract class so that it is only used as a base class.

HealthGoodie

* HealthGoodie(StudentWorld\* studentWorld, **double** startX, **double** startY)
  + Constructor for a HealthGoodie object; takes its location.
* **void** playerInteraction()
  + Specifies what will occur when a HealthGoodie interacts with the player. Will not be virtual because not class is derived from HealthGoodie.

FTGoodie

* FTGoodie(StudentWorld\* studentWorld, **double** startX, **double** startY)
  + Constructor for a FTGoodie object; takes its location.
* **void** playerInteraction()
  + Specifies what will occur when a FTGoodie interacts with the player. Will not be virtual because not class is derived from FTGoodie.

LifeGoodie

* LifeGoodie(StudentWorld\* studentWorld, **double** startX, **double** startY)
  + Constructor for a LifeGoodie object; takes its location.
* **void** playerInteraction()
  + Specifies what will occur when a LifeGoodie interacts with the player. Will not be virtual because not class is derived from LifeGoodie.

Fungus

* Fungus(StudentWorld\* studentWorld, **double** startX, **double** startY)
  + Constructor for a Fungus object; takes its location.
* **void** playerInteraction()
  + Specifies what will occur when a Fungus interacts with the player. Will not be virtual because not class is derived from Fungus.

Agent

* Agent(StudentWorld\* studentWorld, **int** objectType, **int** imageID, **double** startX, **double** startY, Direction dir, **int** hitPoints)
  + Constructor for an Agent object.
* **virtual** ~Agent() {}
  + As a base class, Agent must have a virtual destructor, even if it does nothing.
* **void** takeDamage(**int** amount)
  + Taking damage has the same effect for every agent; it subtracts its current hit points by the amount of damage inflicted. The function is not declared virtual so that no derived class can override it.
* **void** gainHitPoints(**int** amount)
  + Gaining hit points has the same effect for every agent; it adds the amount of additional hit points to the current number of hit points. The function is not declared virtual so that not derived class can override it.
* **int** hitPoints() **const**
  + Returns the number of hit points the agent currently has. This function is not virtual because it performs the same thing for every type of Agent, so it may not be overridden by any derived classes.
* **virtual** **void** playHurtSound() **const** = 0 {
  + Agent does not have a sound it plays when hurt. The sound that plays must be specified in its derived classes. This function is pure virtual so that every derived class of Agent must override the function. Also makes Agent an abstract class, meaning it can only be used as a base class and cannot be declared outright.
* **virtual** **void** playDeadSound() **const** = 0
  + Agent does not have a sound it plays when it dies. The sound that plays must be specified in its derived classes. This function is pure virtual so that every derived class of Agent must override the function. Also makes Agent an abstract class, meaning it can only be used as a base class and cannot be declared outright.

Socrates

* Socrates(StudentWorld\* studentWorld, **double** startX, **double** startY)
  + Constructor for a Socrates class; takes its initial location.
* **void** doSomething()
  + Socrates has its own actions to perform during every tick. There are no classes derived from Socrates, so there is no point in considering making this function virtual.
* **void** increaseFTCharges(**int** amount)
  + Increases the number of flames Socrates has left. Mainly used by the flame goodie in its player interaction.
* **void** playHurtSound() **const**
  + Specifies the sound to play when Socrates gets hurt.
* **void** playDeadSound() **const**
  + Specifies the sound to play when Socrates dies.
* **int** sprays() **const**
  + Returns the current number of sprays Socrates has left. Mainly used by the StudentWorld to update the game text.
* **int** ftCharges() **const**
  + Returns the current number of sprays Socrates has left. Mainly used by the flames to update the game text.

Bacteria

* Bacteria(StudentWorld\* studentWorld, **int** objectType, **int** imageID, **double** startX, **double** startY, **int** hitPoints, **int** damage, **bool** isAggressiveSalmonella, **bool** isSalmonella)
  + Constructor for a Bacteria object.
* **virtual** ~Bacteria() {}
  + As a base class, Bacteria must have a virtual destructor, even if it does nothing.
* **void** doSomething()
  + Each type of bacteria has similar actions performed during each tick, so the function is defined in Bacteria and not allowed to be overridden by any derived classes.
* **bool** aggressiveAction()
  + An action specific to aggressive bacteria, that may or may not be performed in the doSomething() function.
* **virtual** **void** finalAction() = 0
  + Every type of bacteria has its own unique final action performed during each tick. As such, the function is declared pure virtual in Bacteria so that every class derived from Bacteria must fill in its own final action.
* **double** distance(**double** x1, **double** y1, **double** x2, **double** y2) **const**
  + Calculates the distance between two given points.
* **void** resetMovementPlan()
  + Every bacteria has a movement plan distance is reset to 10 if needed. Derived classes from Bacteria are not allowed to override this function because it acts the same way for all types of Bacteria.
* **int** movementPlan()
  + Returns the current movement plan distance of the Bacteria. Acts the same for every type of Bacteria, so it is not declared virtual.
* **void** decreaseMovementPlan()
  + Decreases the current movement plan distance of the Bacteria. Acts the same for every type of Bacteria, so it is not declared virtual.

Salmonella

* Salmonella(StudentWorld\* studentWorld, **int** objectType, **double** startX, **double** startY, **int** hitPoints, **int** damage, **bool** isAggressiveSalmonella)
  + Constructor for a Salmonella object.
* **virtual** ~Salmonella() {}
  + As a base class, Bacteria must have a virtual destructor, even if it does nothing.
* **void** finalAction()
  + Must specify the final action a Salmonella object will take during its call to the doSomething() function of the Bacteria base class. This function is not virtual because it performs the same set of actions for every type of Salmonella and must not be overridden by any derived classes.
* **void** playHurtSound() **const**
  + Specifies what sound will play when a Salmonella object is hurt. All Salmonella objects play the same sound when hurt, so this function is not virtual so that derived classes cannot override it.
* **void** playDeadSound() **const**
  + Specifies what sound will play when a Salmonella object dies. All Salmonella objects play the same sound when they die, so this function is not virtual so that derived classes cannot override it.

RegularSalmonella

* RegularSalmonella(StudentWorld\* studentWorld, **double** startX, **double** startY)
  + Constructor for a RegularSalmonella object; takes its initial location.

AggressiveSalmonella

* AggressiveSalmonella(StudentWorld\* studentWorld, **double** startX, **double** startY)
  + Constructor for a AggressiveSalmonella object; takes its initial location.

Ecoli

* Ecoli(StudentWorld\* studentWorld, **double** startX, **double** startY)
  + Constructor for an Ecoli object; takes its initial location.
* **void** finalAction()
  + Must specify the final action an Ecoli object will take during its call to the doSomething() function of the Bacteria base class. All Ecoli perform the same final action, so the function is not virtual.
* **void** playHurtSound() **const**
  + Specifies what sound will play when an Ecoli object is hurt.
* **void** playDeadSound() **const**
  + Specifies what sound will play when an Ecoli object dies.

**2. Missing Functionality:**

* There is no outright missing functionality in my program. All of the required game mechanics are inside my program’s game. However, there are differences between the model game we were given and the game I have programmed, mainly due to design decisions not made clear by the spec. These differences cause the model game and my game to behave differently in a few specific aspects. For one, the E. Coli in the model game move smoothly around the dirt piles they collide with, whereas the E. Coli in my game exhibit more friction-like behavior with the dirt piles they attempt to circumvent.

**3. Design Decisions:**

* For the Bacteria class’s doSomething() function, there were many instances where the spec said to make sure that if the Bacteria was to move 3 units in the given direction, it wouldn’t overlap with dirt along the way. This made it unclear whether or not the Bacteria would need to check the path towards the final location or just the final location. I decided to implement a loop that would first check 1 unit of movement, then 2, and finally 3. Had either 1 unit or 2 units been successful, but 3 units was not, I chose to have the Bacteria stay still.
* The spec did not mention where the lifetime of each item should be decreased. I decided to decrease each item’s lifetime by one at the end of the item’s doSomething() function.
* Some actors would have interactions with other actors (i.e. bacteria picking up food, Socrates damaged by fungus). These sorts of interactions were typically mentioned in the other circumstances section of each class description and left the programmer to decide how to manage these interactions. I decided to have only one actor in each actor-to-actor interaction manage it. The actor that would manage it would be the one that caused it. For example, spray moving into bacteria would be managed by spray, or bacteria moving into Socrates would be managed by Socrates.

\*\*Note: many of the test cases included adding ‘cout’ lines throughout each class's functions to make sure the proper effects were occuring. I would often times print out things that the game wouldn’t show (i.e. an item’s lifetime, a bacteria’s hit points)\*\*

**4. Testing Each Class:**

* StudentWorld
  + The main functions that I need to check in StudenWorld were init(), move(), and cleanup(). At the start of the game, I could tell whether or not init() was working properly by examining the number of dirt piles, pits, and food objects and their placement (i.e. whether or not objects that were not supposed to overlap). I could also see that Socrates spawned in the correct location.
  + To check the move() function I would have to make sure that every actor was doing something during each tick of the game. The pits were periodically spawning bacteria. The bacteria were moving throughout the petri dish in their own unique moving patterns. Socrates could move clockwise, counterclockwise, or fire a projectile. Projectile that collide with damageable objects or reach their max distance would disappear. The game text at the top of the screen was constantly being updated. New goodies and fungi would spawn periodically on the rim of the petri dish.
  + To test the cleanup() function, I would either finish the current level or let Socrates die. Once the petri dish reset, I would know that the actors of the previous playthrough were removed. I’d also watch out for any memory leak warnings at the end of the game.
* Dirt
  + To test dirt I had to make sure that it wouldn’t do anything unless it interacts with another actor. When left alone, the dirt piles remained in their position. When hit by a projectile, the dirt pile would disappear. Bacteria would not be able to travel over the dirt piles.
* Food
  + Food would not do anything during a tick unless it interacted with a particular actor. If bacteria overlapped with a food object, the food object would disappear.
* Pit
  + To test pits, I would count the number of bacteria that it spawned throughout its lifetime and make sure it matched its initial inventory. Once all the pit depleted its inventory of bacteria, it would disappear. The pit would not interact with other actors. Bacteria and projectiles could travel over any pits they encountered. Also, I made sure bacteria born sounds were being played when necessary.
* Spray
  + I moved Socrates all around the petri dish and made sure that at each spot he could possibly fire, the spray projectile was firing in the correct direction. I would fire at bacteria and track their hit points to make sure the spray projectiles were inflicting the correct number of damage. I would fire at dirt projectiles to make sure the sprays would destroy dirt piles. I would fire at pits and food objects to make sure the sprays would travel across them. When the projectile would not collide with any objects, it would disappear at the correct travel distance. I monitored the spray count as Socrates fired or stopped firing. Also, I checked to make sure the spray sound was playing when fired.
* Flame
  + I moved Socrates all around the petri dish and made sure the flame attack would properly fire all around him. I checked to make sure that flames were damaging the actors they should and inflicted the right amount of damage. Flames would immediately destroy dirt piles, goodies, and fungi, when coming into contact with them. Flames would kill E Coli and regular salmonella in one hit, while killing aggressive salmonella in two hits. Flame would play the correct sound whenever fired. Flames would not interact with food objects or pits.
* HealthGoodie
  + Health goodies were spawning periodically and randomly around the petri dish. When Socrates came into contact with a health goodie, he would gain the proper amount of hitpoints, but not exceed 100. A goodie acquired sound would play when necessary. Health goodies would be destroyed whenever coming into contact with a projectile. The user score was changed correctly.
* FTGoodie
  + Flame goodies were spawning periodically and randomly around the petri dish. When Socrates came into contact with a flame goodie, he would gain the 5 flame charges as shown in the game text. A goodie acquired sound would play when necessary. Flame goodies would be destroyed whenever coming into contact with a projectile. The user score was changed correctly.
* LifeGoodie
  + Life goodies were spawning periodically and randomly around the petri dish. When Socrates came into contact with a life goodie, his number of lives would increase by one. A goodie acquired sound would play when necessary. Life goodies would be destroyed whenever coming into contact with a projectile. The user score was changed correctly.
* Fungus
  + Fungus were spawning periodically and randomly around the petri dish, but more often than any individual type of goodie. When Socrates came into contact with a fungus, he would lose the proper amount of hit points and the fungus would disappear. No sound was played during this interaction. Fungus would be destroyed immediately whenever coming into contact with a projectile. The user score was changed correctly.
* Socrates
  + I had to make sure that all of the correct user inputs were working. The left key would cause Socrates to move 5 degrees around the petri dish in the counterclockwise direction. The right key would cause Socrates to move 5 degrees around the petri dish in the clockwise direction. The space key would fire a single spray projectile in the direction Socrates was currently facing. The enter key would fire flame projectiles all around Socrates. When Socrates came into contact with bacteria, he would lose the correct number of hit points and play the sound indicating he was hurt. When his hit points reached zero, he would play the sound indicating that he had died and his number of lives would decrease by one. Whenever Socrates came into contact with a goodie, the specific goodie interaction would occur and the goodie would disappear. Whenever Socrates came into contact with a fungus, he would lose the correct number of hitpoints.
* RegularSalmonella
  + In order to test the regular salmonella I removed the pit’s ability to spawn aggressive salmonella, since both types of salmonella have the same graphic image. I made sure that the damage the regular salmonella inflicted on Socrates was the correct amount. It would either move in its current direction or attempt to move towards any nearby food objects. The regular salmonella would not travel over a dirt pile and would always attempt to move in a new direction when it encountered one. When examining the regular salmonella’s actions that were difficult to see just by monitoring the game, I would comment out certain other functions and only let one regular salmonella spawn.
* AggressiveSalmonella
  + In order to test the aggressive salmonella I removed the pit’s ability to spawn regular salmonella, since both types of salmonella have the same graphic image. I made sure that the damage the aggressive salmonella inflicted on Socrates was the correct amount. The aggressive salmonella would try to move towards Socrates If Socrates was not nearby, it would either move in its current direction or attempt to move towards any nearby food objects. The regular salmonella would not travel over a dirt pile and would always attempt to move in a new direction when it encountered one. When examining the aggressive salmonella’s actions that were difficult to see just by monitoring the game, I would comment out certain other functions and only let one aggressive salmonella spawn.
* Ecoli
  + Testing the E coli mainly consisted of making sure that they would follow around Socrates when they were close enough. I would check the distance from the E coli to Socrates by having the E coli print out the distance during every tick. I made sure that the damage the E coli inflicted on Socrates was the correct amount. The E coli would not be able to travel through any dirt piles and instead it would attempt to circumvent any dirt piles it encountered. When hit by a projectile, the E coli would sustain the correct amount of damage. When examining the E coli’s actions that were difficult to see just by monitoring the game, I would comment out certain other functions and only let one E coli spawn.