**Part I**

*Actor Class : Public GraphObject*

*bool isAlive() const:*

This returns whether an Actor is still alive. It was not defined virtual and was placed in the Base Class Actor because all derived classes would need access, but it would work the same for all.

*bool inBounds(double x, double y, int r) const:*

This returns whether a coordinate (x,y) is within a certain radius from the center of the map. It was not defined virtual and was placed in the Base Class Actor because multiple derived classes would need access, but it would act the same for all.

*void setDead():*

This changes an Actor’s member variable to denote that it is dead. It was not defined virtual and was placed in the Base Class Actor because all derived classes would need access, but it would work the same for all.

*bool isOverlap(Actor\* actor, int collideParam) const:*

This determines if the Actor calling it overlaps with the Actor passed in. It was not defined virtual and was placed in the Base Class Actor because all derived classes would need access, but it would work the same for all.

*bool isOverlap(double x, double y, Actor\* actor, int collideParam) const:*

This determines if an Actor passed in overlaps with the passed in coordinates. It was not defined virtual and was placed here to avoid confusion by placing the overload somewhere different and because the only use was in a separate file by an actor object so derived classes did not need to redefine it.

*int distance(double x1, double y1, double x2, double y2) const:*

This calculates the distance between two passed in coordinates. It was not defined virtual and was placed in the Base Class Actor because all derived classes would need access, but it would work the same for all.

*StudentWorld\* getWorld() const:*

This returns a pointer to the StudentWorld object that the Actor resides in. It was not defined virtual and was placed in the Base Class Actor because all derived classes would need access, but it would work the same for all.

*virtual bool isDamageable() const = 0:*

This returns whether an Actor is able to take damage in some form. It was defined virtual so that all derived classes of Actor could redefine it to denote whether they took damage.

*virtual void doSomething() = 0:*

This handles an Actor’s actions within a tick. It was defined as pure virtual because all objects created are derived from Actor and they each need to define what they will do in a tick.

*virtual void damage(int x) = 0:*

This handles dealing damage to an Actor. It was defined as pure virtual because many actors derived from Actor take damage in different ways.

*virtual bool isEdible() const:*

This returns whether an Actor can be eaten by certain other Actors. It was defined as virtual so that each derived class could define whether this behavior was true for it but is not pure virtual because not many classes need to define their own version and share a common default functionality.

*virtual bool isDirt() const:*

This returns whether an Actor is Dirt due to the unique properties of dirt. It was defined as virtual so that each derived class could define whether this behavior was true for it but is not pure virtual because not many classes need to define their own version and share a common default functionality.

*virtual bool isPit() const:*

This returns whether an Actor is a pit due to the unique properties of pits. It was defined as virtual so that each derived class could define whether this behavior was true for it but is not pure virtual because not many classes need to define their own version and share a common default functionality.

*virtual void doDamage(Actor\* target):*

This deals damage to an Actor passed in as a parameter. This was defined as virtual because there are Actors derived from the Actor class which deal different amounts of damage but is not pure virtual because not many classes need to define their own version and share a common default functionality.

*virtual int projectileType() const:*

This returns whether an Actor is a Projectile due to the unique properties of projectiles. It was defined as virtual so that each derived class could define whether this behavior was true for it but is not pure virtual because not many classes need to define their own version and share a common default functionality.

*virtual bool isBaddie() const:*

This returns whether an Actor is an enemy of Socrates. It was defined as virtual so that each derived class could define whether this behavior was true for it but is not pure virtual because not many classes need to define their own version and share a common default functionality.

*Socrates Class : Public Actor*

*virtual bool isDamageable() const:*

Returns that Socrates can take damage. Was defined here because it is defined as pure virtual in Actor and to denote that Socrates can be damaged.

*virtual void damage(int x):*

Subtracts passed in amount from Socrates hit points. Was defined here because it is defined as pure virtual in Actor and to define how Socrates takes damage/what he does when he takes damage.

*void heal():*

Resets Socrates’ hit points to full. Was defined here so that Socrates can heal when he interacts with other Actors. Not defined as virtual because he is the only class which needs to heal.

*void addFlame():*

Adds flame charges to Socrates’ current amount. Was defined here so that Socrates can get more charges when he interacts with other Actors. Not defined as virtual because he is the only class which needs this functionality.

*void doSomething():*

Allows Socrates to move, fire projectiles, detect when he is dead, and replenish charges of his projectiles. Was defined here so that Socrates could define what its objects do during a tick and because this is defined as pure virtual in Actor.

*int numFlame() const:*

Returns the number of flame charges Socrates currently has left. Not defined as virtual because he is the only class which needs this functionality.

*int numSpray() const:*

Returns the number of spray charges Socrates currently has left. Not defined as virtual because he is the only class which needs this functionality.

*int getHP() const:*

Returns the number of hit points Socrates currently has left. Not defined as virtual because he is the only class which needs this functionality.

*Dirt Class : Public Actor*

*virtual bool isDamageable() const:*

Returns that Dirt can take damage. Was defined here because it is defined as pure virtual in Actor and to denote that Dirt can be damaged.

*virtual void damage(int x):*

Sets Dirt to dead. Was defined here because it is defined as pure virtual in Actor and to define what occurs when a Dirt object is damaged.

*virtual bool isDirt() const:*

Returns that it is dirt. Was defined here so other functions could tell when an object was dirt.

*void doSomething():*

Does nothing since dirt does nothing during a tick. Was defined here because it was defined as pure virtual in Actor.

*Food Class : Public Actor*

*virtual bool isDamageable() const:*

Returns that food is not able to take damage. Was defined here because it is defined as pure virtual in Actor and to denote that Food can’t be damaged.

*virtual void damage(int x):*

Does nothing since food can’t take damage. Was defined here because it is defined as pure virtual in Actor.

*virtual bool isEdible() const:*

Returns that food is edible. Was defined here so that objects of other classes can interact with food and know that it is able to be eaten.

*void doSomething():*

Does nothing since food does nothing during a tick. Was defined here because it was defined as pure virtual in Actor.

*Projectile Class : Public Actor*

*virtual bool isDamageable() const:*

Returns that Projectiles can not take damage. Was defined here because it is defined as pure virtual in Actor and to denote that Projectiles can’t be damaged.

*virtual void damage(int x):*

Does nothing since Projectiles can’t take damage. Was defined here because it is defined as pure virtual in Actor.

*virtual bool isProjectile() const:*

Returns that it is a Projectile. Was defined here so that objects of other classes can interact with Projectiles and know their specific properties.

*void doSomething():*

Controls movement of projectiles and destroys them when max range is reached. Defined here so that projectiles could move each tick after they are fired and be deleted after impact.

*Flame Class : Public Projectile*

*virtual void doDamage(Actor\* target):*

Defines how much damage flame projectiles do. Defined here so that flames are able to do their specific amount of damage to things they collide with.

*Spray Class : Public Projectile*

*virtual void doDamage(Actor\* target):*

Defines how much damage spray projectiles do. Defined here so that spray are able to do their specific amount of damage to things they collide with.

*Pit Class : Public Actor*

*virtual bool isDamageable() const:*

Returns that pits can’t be damaged. Was defined here because it is defined as pure virtual in Actor and to denote that pits can’t be damaged.

*virtual void damage(int x):*

Does nothing since Pits can’t be damaged. Defined here because it was defined as pure virtual in Actor.

*void doSomething():*

Handles the spawning of Bacteria and keeps track of how many are yet to spawn. Defined here so that there is a chance of bacteria spawning every tick and because it was defined as pure virtual in Actor.

*virtual bool isPit() const:*

Returns that it is a pit. Was defined here so that StudentWorld has a way to tell how many pits are left for proper traversal to the next level.

*Bacteria Class : Public Actor*

*virtual bool isDamageable() const:*

Returns that bacteria are damageable. Defined here so that bacteria can interact with other objects and take damage and because it was defined as pure virtual in Actor.

*virtual void damage(int x):*

Subtracts the passed in amount from the hit points of a bacteria. Defined here so that bacteria lose hit points when damaged and because it was defined as pure virtual in Actor.

*int getHP() const:*

Returns the amount of hit points a bacterium has left. Defined here so that all bacteria can know when they have no more hit points. Functionality is the same for all derived classes.

*bool checkMove():*

Checks whether a bacteria can move forward and if so moves that way. If not a random new direction is determined. Defined here so that all bacteria can avoid collisions with certain objects. Functionality is the same for all derived classes.

*int foodEaten() const:*

Returns the amount of food eaten since last divide. Defined here so that bacteria can divide after a certain amount of food is eaten. Functionality is the same for all derived classes.

*int getPlan() const:*

Returns the value of a bacteria’s plan which denotes whether it will move forward or do something else. Defined here so that all bacteria can decide where to move. Functionality is the same for all derived classes.

*void decPlan():*

Decreases the bacteria’s plan value by one. Defined here to aid in movement of bacteria. Functionality is the same for all derived classes.

*void setFood(int x):*

Sets a bacteria’s food since last divide to passed value. Functionality is the same for all derived classes so it is defined here.

*virtual void divide() = 0:*

Creates a new copy of whatever type of bacteria is dividing. Defined here as virtual void because each derived class splits into itself so the function will be slightly different.

*void doSomething():*

Does nothing since each derived class will define this for itself. Must be defined because it is defined as pure virtual in Actor.

*void resetPlan():*

Sets a bacteria’s plan value to the default of 10. Defined here so that a bacteria can move and because functionality is the same for all derived classes.

*virtual bool isBaddie() const:*

Returns that a bacteria is an enemy. Defined here so that bacteria can act in unique ways with other objects. Functionality is the same for all derived classes.

*int findSocrates(Actor\* hero, int r):*

Sets a bacteria’s direction to where Socrates is. Defined here so that certain types of bacteria can find and attack Socrates. Functionality is the same for all derived classes.

*Salmo Class : Public Bacteria*

*virtual void damage(int x):*

Does the passed in amount of damage and plays a sound specific to the class. Defined here because each type of bacteria plays its own sound.

*void doSomething():*

Checks if the Salmo is still alive, directs movement toward closest food, detects collisions and overlap, controls divisions, and does damage. Defined here because bacteria behavior is unique.

*virtual void divide():*

Creates a new Salmo. Defined here so that specific type of bacteria is created.

*AgroSalmo Class : Public Bacteria*

*virtual void damage(int x):*

Does the passed in amount of damage and plays a sound specific to the class. Defined here because each type of bacteria plays its own sound.

*void doSomething():*

Checks if the AgroSalmo is still alive, directs movement toward closest food unless Socrates is within a distance, detects collisions and overlap, controls divisions, and does damage. Defined here because bacteria behavior is unique.

*virtual void divide():*

Creates a new AgroSalmo. Defined here so that specific type of bacteria is created.

*Ecoli Class : Public Bacteria*

*virtual void damage(int x):*

Does the passed in amount of damage and plays a sound specific to the class. Defined here because each type of bacteria plays its own sound.

*void doSomething():*

Checks if the Ecoli is still alive, directs movement toward socrates, detects collisions and overlap, controls divisions, and does damage. Defined here because bacteria behavior is unique.

*virtual void divide():*

Creates a new Ecoli. Defined here so that specific type of bacteria is created.

*Goodie Class : Public Actor*

*virtual bool isDamageable() const:*

Returns that Goodies are not able to take damage. Defined here because functionality is the same for all derived classes.

*virtual void damage(int x):*

Does nothing since goodies are unable to take damage. Defined here because it was defined as a pure virtual function in Actor.

*int getLifetime():*

Returns the lifetime of a Goodie. Defined here because functionality is the same for all derived classes.

*void decLifetime():*

Decreases the lifetime of a Goodie by one. Defined here because functionality is the same for all derived classes.

*void doSomething():*

Does nothing because Goodie functionality is unique, but must be defined here because it was defined as pure virtual in Actor.

*healthGoodie Class : Public Goodie*

*void doSomething():*

When certain objects collide with this, their hit points are restored to full. Defined here because its function is different from all other goodies.

*flameGoodie Class : Public Goodie*

*void doSomething():*

When certain objects collide with this, their flame charges are increased by 5. Defined here because its function is different from all other goodies.

*lifeGoodie Class : Public Goodie*

*void doSomething():*

When certain objects collide with this, their life count is increased by one. Defined here because its function is different from all other goodies.

*fungus Class : Public Goodie*

*void doSomething():*

When certain objects collide with this, their hit points are lowered. Defined here because its function is different from all other goodies.

*virtual bool isDamageable() const:*

Returns true since fungus is able to take damage. Defined here since fungus behaves differently from other types of goodies.

*virtual void damage(int x):*

Sets fungus to dead when damage is taken. Defined here since fungus behaves differently from other types of goodies.

*StudentWorld Class : Public GameWorld*

*virtual int init():*

Creates initial game objects (Dirt, Socrates, food, pits) and ensures game gets harder as levels progress. Defined here because it is a pure virtual function of GameWorld and because the central game board needs to be initialized. Functionality is unique.

*virtual int move():*

Controls everything that occurs during a tick. Calls all living actors doSomething functions, updates the scoreboard, and deletes dead objects. Defined here because it is a pure virtual function of GameWorld and because without it actors will do nothing. Functionality is unique.

*virtual void cleanUp():*

Deletes all actors created to this point. Defined because if is a pure virtual function of GameWorld and so that the destructor and other classes can get rid of all actors when necessary. Functionality is unique.

*bool isCollision(Actor\* a, int collideParam) const:*

Checks if there is a collision between a passed in actor and any other actors. Defined here because only StudentWorld has access to all actors that have been created but this information is needed for use elsewhere.

*bool isDirtCollision(double x, double y, Actor\* a, int collideParam) const:*

Checks if a perspective move to coordinate x,y by passed in actor will cause a collision with a dirt object. Defined here because only StudentWorld has access to all dirt objects that have been created but this information is needed for use elsewhere.

*bool onCollision(Projectile\* a, int collideParam):*

Defines what happens when a projectile collides with something and checks if a projectile collides with any other actors. Defined here because only StudentWorld has access to all actors that have been created so it must be checked here if a projectile collides with any of them so that specific interactions can take place with that object.

*bool onCollision(Bacteria\* a, int collideParam):*

Defines what happens when a bacteria collides with food and checks if a projectile collides with any food objects. Defined here because only StudentWorld has access to all food objects that have been created so it must be checked here so that specific interactions can take place with that food.

*void addToActors(Actor\* actor):*

Adds the passed in actor to the vector of actors maintained by StudentWorld. Defined here so that actors can be created outside of StudentWorld but still be managed by StudentWorld.

*void noMoreDormant():*

Tells StudentWorld there are no more dormant bacteria. Defined here so that other classes can communicate to StudentWorld when a specific event has occurred.

*Socrates\* getHero() const:*

Returns a pointer to Socrates. Defined here so that other classes such as those derived from actor can interact with Socrates.

*int findClosestFood(Actor\* a, int r):*

Returns the direction of the closest food object within radius r to a passed in Actor. Defined here because only StudentWorld has access to all food objects so to find the closest this calculation must be done here.

**Part II**

Unfinished Functionality: None

Bugs:

* Very rarely it appears some aggressive salmonella get stuck on the edge of the circle
* Aggressive Salmonella sometimes change to random directions when stuck on dirt

**Part III**

Assumptions:

* It wasn’t completely clear whether the 3 pixels ahead of bacteria were to be checked one at a time or 3 at a time so I checked only once if they could move 3 pixels forward
* The spacing for the scoreboard wasn’t given explicitly so I did my best to emulate what it looked like in the sample game.

**Part IV**

**StudentWorld Class**

The testing for this class in particular was somewhat challenging because most of the functions it contains are used so that other classes can interact with each other through the go-between of StudentWorld. Because of this, a lot of the testing done on this class was through close examination of the behavior of other objects derived from the Actor class. Especially with the collision functions, it was useful to use the tick by tick step through to make sure collisions were registering properly. When possible, I also used the debugger to examine specific values and double check that they were what was expected. This was difficult to do for all functions however because for some interactions it would take many ticks before they even had the chance to occur. In situations where the chance of a desired interaction could be increased, I would do so.

Specifically, for the core functions of init, move, and cleanup. I used the debugger to check that items were being initialized properly. For move, I similarly used the debugger to see that all actors were having their doSomething functions called appropriately. As for cleanup, it appears there is no undefined behavior as I have yet to run into any unexpected results that could not be explained by something else and my program has not crashed even after many diverse tests.

**Actor Class**

All of Actor’s functions act only on derived classes so for testing I looked to the behavior of these classes and paid particular attention to Actor’s functions. For many of the functions which discern whether something is or isn’t a type of object, testing was done elsewhere as with pure virtual functions. For functions like isOverlap, the pause functionality was extremely useful in seeing that when objects overlapped the desired outcome was achieved. Specifically, I used the goodies to test this function because they used the isOverlap function without using any StudentWorld collision functions. For inBounds, it was a matter of observing particular cases such as when bacteria moved near the edge of the circle. Lastly, for the function setDead, it was confirmed to work since all items were deleted as they should have been.

**Socrates Class**

To test the Socrates class, I first tested movement by simply playing the game and making sure nothing unexpected occurred with movement, even after long periods of time. To test the ability to shoot projectiles I similarly played the game and compared the results to that of the sample that was given to us. I also made sure that when sprays were depleted the proper amount was returned each tick using the debugger to confirm my play testing. As for the accessor functions, their functionality was tested through the proper behavior of other functions that utilized them.

**Dirt Class**

To test dirt, I used the debugger to make sure the proper number of objects were initialized. I play tested the game to make sure dirt wasn’t doing anything during a tick. To test that dirt disappeared when damaged, I similarly played the game, clearing all dirt to make sure. I also made sure there were no overlaps with other stationary objects which ensured its initialization was done properly. Lastly, later on the isDirt function was confirmed to work because other functions which relied on it were also working as expected.

**Food Class**

To test food, was very similar to testing dirt. I once again used the debugger to make sure the proper number of objects were initialized. I play tested the game to make sure food wasn’t doing anything during a tick. To test that food didn’t disappear when damaged, I similarly played the game, shooting over pieces of food. I also made sure there were no overlaps with other stationary objects which ensured its initialization was done properly. Lastly, later on the isEdible function was confirmed to work because other functions which relied on it were also working as expected.

**Projectile Class**

Testing for the projectile class was done through testing of its derived classes. By seeing that these classes worked, I was able to tell that all functions in the base class were also functioning correctly.

**Flame Class**

To test the flame class, I played the game and compared the behavior to the sample given to us. To test that the flames were properly damaging objects I made sure they had the desired effect when coming into contact with a variety of items. These included bacteria, dirt, pits, food, and goodies. I also used the frame by frame to check that the flame movement looked correct.

**Spray Class**

To test the spray class, I played the game and compared the behavior to the sample given to us. To test that the sprays were properly damaging objects I made sure they had the desired effect when coming into contact with a variety of items. These included bacteria, dirt, pits, food, and goodies. I also used the frame by frame to check that the spray movement looked correct.

**Pit Class**

To test pit functionality, I made sure that the proper number of each bacteria spawned each level through play testing. I also made sure that the level ended when there were no more dormant or live bacteria meaning pit had properly communicated that it was out of bacteria. To do this I would reduce the number of bacteria in each one to increase efficiency of testing. I also made sure that they could not be damaged by projectiles.

**Bacteria Class**

When it came to testing bacteria, the most important thing was their movement. To test this, I spent a lot of time going tick by tick with the pause functionality to see where each bacteria was going each time. I paid especially close attention to when they got stuck and how their behavior changed when they were near certain other objects. I also tested their damaging ability through play as well as their ability to take damage. Similarly, I used the pause functionality to check that they never went through objects (dirt) that they were not supposed to.

**Salmo Class**

With salmonella, I was most concerned with how they acted in proximity to food. To test this, I used pause functionality to go tick by tick and check how they behaved around food. To help with this, I reduced the number of foods to only one so It was clear when they were going for it. To test divides, I placed three pieces of food, and one salmo object to see that it properly duplicated.

**AgroSalmo Class**

With the aggressive salmonella, testing involved some similar things to regular salmonella but with slight alterations. To test proximity to food while Socrates was not in range, I used pause functionality to go tick by tick and check how they behaved around food. I then repeated this process for when Socrates was in range and when Socrates was in range but food was not. I also made sure to shoot them with two flames to test their unique health. To test divides, I placed three pieces of food, and one agroSalmo object to see that it properly duplicated.

**Ecoli Class**

With Ecoli, testing was once again quite similar to the other bacteria. This time I used pause functionality to go tick by tick and check how they behaved in relation to socrates’ movement. This involved letting Socrates sit still, constant motion, and a mix. I also observed when they were being blocked by dirt to make sure it was correct. To test divides, I placed three pieces of food, and one Ecoli object to see that it properly duplicated.

**Goodie Class**

To test the Goodie’s doSomething, I compared the spawn rate to that of the example to make sure they were the same. I then increased the spawn rate to feel out whether the distribution and lifespans seemed correct as well. All other esting for the Goodie class was done through testing of its derived classes. By showing that these classes work, it is ensured that the base class functions in use are also working properly.

**healthGoodie Class**

To test that healthGoodie was working properly, I first tested that they were being picked up. To do this I simply increased the spawn rate of goodies and made them all of this type. By playing the game and picking them up I was able to see that they had the desired effect and had the correct interactions.

**flameGoodie Class**

To test that flameGoodie was working properly, I first tested that they were being picked up. To do this I simply increased the spawn rate of goodies and made them all of this type. By playing the game and picking them up I was able to see that they had the desired effect and had the correct interactions.

**lifeGoodie Class**

To test that lifeGoodie was working properly, I first tested that they were being picked up. To do this I simply increased the spawn rate of goodies and made them all of this type. By playing the game and picking them up I was able to see that they had the desired effect and had the correct interactions.

**fungus Class**

To test that fungus was working properly, I first tested that they were damaging upon collision. To do this I simply increased the spawn rate of fungus. By playing the game and picking them up I was able to see that they had the desired effect and had the correct interactions. I also made sure they were destructible by using the flames nearby.