1. **class Actor**

**StudentWorld\* getWorld() const;**

* **Functionality:** Returns a pointer to the world
* Defined in the base actor class because all derived classes may need to call a function from StudentWorld, and only this return function will allow so.

**bool getAlive() const;**

* **Functionality:** Returns the current state of an actor (alive/dead)
* Defined in the base actor class because all objects have an alive or dead state

**int getHP() const;**

* **Functionality:** Returns the health of an actor; if actor does not possess an hp value, -1 will be returned
* Defined in the base actor class because most objects have a health value

**void setDead();**

* **Functionality:** Sets the status of an actor to dead
* Defined in the base actor class because all actors must have a method to set their alive status to false

**void updateHP(int dmgAmt);**

* **Functionality:** Decreases the health value of a current actor by *amtDmg* and set actor’s state to dead if health falls below zero
* Defined in the base actor because almost all actors must have a method to show damage being done

**virtual bool checkSpawnSpacing(double& x, double& y);**

* **Functionality:** Checks if spacing between two spawning objects is valid; if not, generates a valid x and y coord
* Defined as virtual because the spacing requirements when spawning in an object may be different based on the actor; however, the default spacing requirement should be simply SPRITE\_WIDTH apart

**virtual bool checkDamageable() const = 0;**

* **Functionality:** Checks if actor is damageable or not (can be destroyed by projectiles or attacked by bacteria)
* Defined as pure virtual because there is about an even number of actors that can be damaged vs the ones that can’t be damage

**virtual bool checkBacteriaConsumable() const;**

* **Functionality:** Checks if actor can be consumed by bacteria
* Defined as virtual because all except one actor can’t be consumed by bacteria; having said this, the return value for this function in the base class would be false

**virtual bool checkBlockableActor() const;**

* **Functionality:** Checks if actor can block the movements of other actors
* Defined as virtual because all except one actor can’t block the path of another actor; having said this, the return value for this function in the base class would be false

**virtual int checkBacteriaType() const;**

* **Functionality:** Returns an integer based on the type of bacteria an actor is (0 being not a bacteria, 1 being RegSal, 2 being AggSal, and 3 being Ecoli)
* Defined as virtual because only the bacterias have a specific type; having said this the, the return value for this function in the base class would be false

**virtual void doSomething() = 0;**

* **Functionality:** The movement/actions of each actor during every tick
* Defined as pure virtual because each and every actor has their own unique set of actions performed during each tick

**Socrates : public Actor**

**int getSprayCharge() const;**

* **Functionality:** returns number of spray charges remaining
* Defined as public to retrieve info for the text display statistics

**int getFlameCharge() const;**

* **Functionality:** returns number of flame charges remaining
* Defined as public to retrieve info for the text display statistics

**void addFlameCharge();**

* **Functionality:** add 5 flame charges to the current number of flame charges
* Defined as public to allow flameCharge goodie to alter number of flame charges when consumed

**virtual bool checkDamageable() const;**

* **Functionality:** returns true because Socrates is a damageable actor

**virtual void doSomething();**

* **Functionality:** Checks for any inputs received by the player; valid inputs include left key (move Soc counterclockwise), right key (move Soc clockwise), space bar (shoots spray if spray count is above 0), and enter (shoots flames if flame count is above 0)

**Projectile : public Actor**

**virtual bool checkDamageable() const;**

* **Functionality:** returns false because projectiles can’t be damaged

**virtual void doSomething() = 0;**

* Defined as pure virtual because each projectile has their own unique set of actions performed during each tick

**int getDamageAmount() const;**

* **Functionality:** returns the damage amount of a projectile
* Defined as a public accessor to retrieve the common private member variable possessed by every projectile

**int gettDist() const;**

* **Functionality:** returns the max travel distance of a projectile
* Defined as a public accessor to retrieve the common private member variable possessed by every projectile

**void dectDist();**

* **Functionality:** decreases the travel distance by SPRITE\_WIDTH
* Defined as a public mutator to perform common actions needed by every projectile

**Flame : public Projectile**

**virtual void doSomething();**

* **Functionality:** moves flame projectile outwards a distance of 32 pixels from the center point and checks if projectile overlaps with any damageable actor. If so, damage that actor and remove the flame projectile from screen

**Spray : public Projectile**

**virtual void doSomething();**

* **Functionality:** Moves spray projectile in the 112 in the angle of Soc and checks if projectile overlaps with any damageable actor. If so, damage that actor and remove the spray projectile from screen

**class Consumable : public Actor**

**virtual bool checkDamageable() const;**

* **Functionality:** returns true because consumables can be damaged

**int getLifetime() const;**

* **Functionality:** Returns the lifetime of a consumable
* Defined as a public accessor to retrieve the common private member variable possessed by every consumable

**void decLifetime();**

* **Functionality:** Decreases the life time by 1
* Defined as a public mutator to perform common actions needed by every consumable

**virtual void doSomething() = 0;**

* Defined as pure virtual because each consumable has their own unique set of actions performed during each tick

**class RestoreHealthGoodie : public Consumable**

**virtual void doSomething();**

* **Functionality:** Checks to see if goodie overlaps Socrates; if so, Soc’s health is restored and 250 points are added. Lifetime decreases by 1 per tick and goodie disappears when lifetime equals 0

**class FlamethrowerGoodie : public Consumable**

**virtual void doSomething();**

* **Functionality:** Checks to see if goodie overlaps Socrates; if so, Soc’s flame charges are increased by 5 and 250 points are added. Lifetime decreases by 1 per tick and goodie disappears when lifetime equals 0

**class ExtraLifeGoodie : public Consumable**

**virtual void doSomething();**

* **Functionality:** Checks to see if goodie overlaps Socrates; if so, add an extra life and 500 points are added. Lifetime decreases by 1 per tick and goodie disappears when lifetime equals 0

**class Fungus : public Consumable**

**virtual void doSomething();**

* **Functionality:** Checks to see if fungus overlaps Socrates; if so, decrease Soc’s health by 20 and 50 points are subtracted. Lifetime decreases by 1 per tick and fungus disappears when lifetime equals 0

**class Bacteria : public Actor**

**virtual bool checkDamageable() const;**

* **Functionality:** returns true because bacteria can be damaged

**virtual int checkBacteriaType() const;**

* **Functionality:** returns 1 because ⅔ types of bacteria are salmonella
* Declared as virtual because the other type of bacteria is e coli

**int getFoodEaten() const;**

* **Functionality:** Returns the amount of food eaten by a bacteria
* Defined as a public accessor to retrieve the common private member variable possessed by every bacteria

**int getMovementPlan() const;**

* **Functionality:** Returns the movement plan of a bacter
* Defined as a public accessor to retrieve the common private member variable possessed by every bacteria

**void incFoodEaten();**

* **Functionality:** Increases the food eaten by 1
* Defined as a public mutator to perform common actions needed by every bacteria

**void decMovementPlan();**

* **Functionality:** Decreases movement plan by 1
* Defined as a public mutator to perform common actions needed by every bacteria

**void resetFoodEaten();**

* **Functionality:** Resets food eaten to zero
* Defined as a public mutator to perform common actions needed by every bacteria

**void setMovementPlan(int amt);**

* **Functionality:** Sets movement plan to the value of *amt*
* Defined as a public mutator to perform common actions needed by every bacteria

**void generateDividePoint(double& x, double& y);**

* **Functionality:** Computes the value of x and y reflected across the vertical and horizontal diameter of the arena
* Defined as a public function because all bacterias utilize this function when dividing into 2

**void procedure(int dmgAmt, int type);**

* **Functionality:** Checks to see if bacteria is overlapping Socrates (damage Soc), number of food eaten equals 3 (duplicate), or bacteria is overlapping food (consume food)
* Defined as a public function to perform a common set of task in each Bacteria’s *doSomething()*

**virtual void doSomething() = 0;**

* Defined as pure virtual because each bacteria has their own unique set of actions performed during each tick

**class RegularSalmonella : public Bacteria**

**virtual void doSomething();**

* **Functionality:** Perform procedure, if movement plan is greater than 0, check if moving forward 3 pixels is a valid path; if so, move forward 3 pixels; otherwise, change to a random direction. If the movement plan is 0, find the closest food and move towards the food. If there are no nearby food, reset movement plan and get a random direction

**class AggressiveSalmonella : public Bacteria**

**virtual void doSomething();**

* **Functionality:** If Socrates is within 72 pixel radius, find direction towards Socrates and move 3 pixels towards Soc (if AggSal gets stuck on a dirt pile it shall remain there). However, if Soc is not with proximty, perform procedure, if movement plan is greater than 0, check if moving forward 3 pixels is a valid path; if so, move forward 3 pixels; otherwise, change to a random direction. If the movement plan is 0, find the closest food and move towards the food. If there are no nearby food, reset movement plan and get a random direction

**class eColi : public Bacteria**

**virtual int checkBacteriaType() const;**

* **Functionality:** Return 2 because the bacteria type is E Coli

**virtual void doSomething();**

* **Functionality:** Perform procedure. If Socrates is within 255 pixel radius, find direction towards Socrates and move 2 pixels towards Soc. If e coli happens to get stuck on a rock increase direction incrementally by 10, 10 times. If still stuck, it shall remain there.

**class initialSpawnedActor**

**virtual bool checkDamageable() const;**

* **Functionality:** Return false because a majority of InitialSpawnedActors are undamageable

**virtual bool checkBacteriaConsumable() const;**

* **Functionality:** Return false because a majority of InitialSpawnedActors can’t be consumed by bacteria
* This is redefined in this class because the derived class food can be consumed by bacteria

**virtual bool checkBlockableActor() const;**

* **Functionality:** Return false because a majority of InitialSpawnedActors are not blockable actors
* This is redefined in this class because the derived class dirt is a blockable Actor

**virtual void doSomething();**

* **Functionality:** Do nothing because a majority of InitialSpawnedActors have nothing in their doSomething method

**class Dirt : public initialSpawnedActor**

**virtual bool checkDamageable() const;**

* **Functionality:** Return true because a dirt can be damage

**virtual bool checkBlockableActor() const;**

* **Functionality:** Return true because a dirt is a blockable actor

**class Food : public initialSpawnedActor**

**virtual bool checkBacteriaConsumable() const;**

* **Functionality:** Return true because food can be consumed by bacteria

**class Pits : public initialSpawnedActor**

**virtual void doSomething()**

* **Functionality:** Releases bacterias randomly through a random number generator. Each pit contains 5 regular salmonella, 3 aggressive salmonella, and 2 e coli. Once all bacteria is released the pit will set itself to dead and disappear from the petri dish.

**class StudentWorld : public GameWorld**

**Socrates\* getSoc() const;**

* **Functionality:** Return a pointer to socrates
* Defined as a public accessor, so each Actor will be able to recognize Socrates and call functions with Soc (ex. Bacteria can update Soc’s hp if it overlaps him)

**void insertProjectile(double x, double y, Direction dir, int flameOrSpray)**;

* **Functionality:** Inserts a projectile (flame or spray) to the world at the given x and y coordinates and the given direction, dir. Projectile will be pushed to the end of our vector containing all actors.
* Defined as a public function to allow Socrates to spawn a projectile when either “space” or “enter” key are pressed.

**void insertBacteria(double x, double y, int type);**

* **Functionality:** Inserts a bacteria (regSal, aggSal, or E Coli) to the world at the given x and y coordinates. Type of bacteria will be specified by the argument passed into type (0, 1, or 2), and that bacteria will be pushed to the end of our vector containing all actors.
* Defined as a public function to allow the pit to spawn bacterias into the world.

**bool projectileDamage(double x, double y, int dmgAmt);**

* **Functionality:** Loop through the container of actors, if the actor is within proximity of projectile and can be damaged, that actor will be damaged by the amount specified in dmgAmt. If this actor is a bacteria, extra commands will be called (ex. play sound and increase score). If projectile hits an object, return true; otherwise, return false.
* Defined as a public function to give projectiles the ability to damage objects in the world

**bool overlapsSoc(double x, double y);**

* **Functionality:** If passed in x and y coordinates are within SPRITE\_WIDTH of Socrates, return true; otherwise, return false.
* Defined as a public function to give actors the ability to check if they are within close proximity of Socrates

**bool foodConsumed(double x, double y);**

* **Functionality:** If passed in x and y coordinates are within SPRITE\_WIDTH of a food actor, return true and set food object to dead; otherwise, return false.
* Defined as a public member function to check if bacteria is overlapping any food

**bool validBacteriaPath(double x, double y, double dist, Direction dir);**

* **Functionality:** Check if moving bacteria *dist* forward will be within SPRITE\_WIDTH / 2 of a blockable actor or moving it forward will move the bacteria off the petri dish, if so, return false; otherwise, return true.
* Defined as a public member function so bacteria can check if moving some distance forward is a valid path.

**double stuckOnDirt(double x, double y, double dist, Direction dir);**

* **Functionality:** Check if moving bacteria *dist* forward will be within SPRITE\_WIDTH / 2 of a blockable actor, if so, Bacteria will return a distance value that will put the bacteria right next to the dirt pile and nothing more. If the path ahead is valid, *dist* will be returned
* Defined as a public member function so aggSal and eColi can get stuck on a dirt pile

**double findClosestFood(double x, double y, Direction& dir);**

* **Functionality:** Find the closest food within 128 pixels of the passed in x and y coordinates. If there is a food within this proximity, return the distance of that food and change the value of dir to the angle towards that food.
* Defined as a public member function so all Bacteria actors can locate the closest food

**double findSocratesDist(double x, double y, int& dir);**

* **Functionality:** Find the distance of Socrates from the passed in x and y coordinates. Return that distance and change the value of dir to the angle towards that food.
* Defined as a public member function so all actors can locate their distance from Socrates

**void setBacteriaReleased();**

* **Functionality:** Sets allBacteriaReleased to true to notify the world that all bacteria have been released from the pits
* Defined as a public member function so pits can tell the world that they released their inventory of bacterias

**virtual int init();**

* **Functionality:** Spawns all pits, food, and dirts at the start of every level by pushing these actors in the container of actors. Also spawns Socrates and set all BacteriaReleased to false.

**virtual int move();**

* **Functionality:** Have each actor do something, remove any dead actors, add goodies to the world, and update the statistics visual on the top of the screen.

**virtual void cleanUp();**

* **Functionality:** At the end of every round, remove any remaining dynamically allocated variables

1. As of recent testing this bug seems to have gone away; however, there are moments when the aggressive salmonella will jitter when overlapping Socrates.
2. I was unsure whether the flame projectile is supposed to move 32 pixels from the center or 40 because the demo game seems to have it move out 40, yet the game specs say 32
3. **class Actor : public GraphObject**

* Being the base class of all the actors, the Actor class was tested to make sure all derived classes had access to the public member functions inside this class. I tested to make sure all virtual functions can be overridden by the derived and any pure virtual function had to be mandatorily redefined. In addition, because Actor has pure virtual functions, it is an abstract base class. This means constructing an object of type Actor would result in an error. When checking if this is the case, an error was in fact presented.

**class Socrates : public Actor**

* Socrates, the player controlled character was primarily tested for its ability to read inputs from the user and execute them properly. For example, hitting the left or right key should move Socrates 5 degrees around the petri the dish. I tested if my command were true by checking if 72 clicks would bring Socrates back to its original position. When testing the spray and flames, I checked to make sure exactly 16 flames were spawned and that these inputs would only run if charges remain.

**class Projectile : public Actor**

* Projectiles were not as complex as the other classes, but yet still a very crucial component of the game. In constructing a projectile, each object must be initialized with both a damage amount and travel distance. The doSomething function in the Projecticle class should be pure virtual because each derived class has their own unique set of commands. Projectiles can’t be damaged and mutators and accessors were provided so the base class has access to both the damage amount and travel distance.

**class Flame : public Projectile**

* The flame projectile was tested for its correct damage output and travel distance. I made sure the damage output was exactly 5 by testing it on an aggressive salmonella which it should not kill in one projectile and made sure the flames traveled exactly 32 pixels.

**class Spray : public Projectile**

* The spray projectile was tested for its correct damage output and travel distance. I made sure the damage output was exactly 2 by testing it on an aggressive salmonella which it should not kill in one projectile and made sure the sprays traveled exactly 112 pixels.

**class Consumable : public Actor**

* All consumables have a lifetime, are consumed by Socrates if overlapped, and can be destroyed. Lifetime was tested in game by letting the consumable sit until they disappeared. On the other hand, being consumable and destroyable were tested by interacting Socrates with it. The doSomething function in the Consumable class should be pure virtual because each derived class has their own unique set of commands. Finally, mutators and accessors were provided so the base class has access to lifetime.

**class RestoreHealthGoodie : public Consumable**

* The restore health goodie restores Socrates health and awards the player with 250 points. I had Socrates overlap this goodie to make sure these bonuses were given when consumed.

**class FlamethrowerGoodie : public Consumable**

* The flamethrower goodie increases Socrates flamethrower count by 5 and awards the player with 250 points. I had Socrates overlap this goodie to make sure these bonuses were given when consumed.

**class ExtraLifeGoodie : public Consumable**

* The extra life goodie increases the player’s life by 1 and awards the player with 500 points. I had Socrates overlap this goodie to make sure these bonuses were given when consumed.

**class Fungus : public Consumable**

* Fungus is the only “bad” consumable out of the four. Instead of providing bonuses, it decreases Socrates health by 20 and subtracts 50 points from the player. I had Socrates overlap this goodie to make sure these penalties were given when consumed.

**class Bacteria : public Actor**

* All bacteria have a movement plan, a food eaten count, and the ability to damage socrates. The movement plan describes a specific pattern each bacteria would take if no special circumstances are present. I tested this by removing all food and Socrates from the petri dish and seeing how the bacteria would react. When testing the food eaten count, I made sure that the bacteria would divide into two if 3 food actors are consumed. To check for its ability to damage Soc, I spawned a bacteria right on top of Socrates to observe if Socrate was being damaged. I Pizza spawning after the death of a bacteria was also tested by having Socrates kill bacteria.

**class RegularSalmonella : public Bacteria**

* The regular salmonella was quite a simple movement pattern. It moves forward until blocked by a dirt pile. If this is the case, it searches for the closest food within 128 pixels and moves toward that food. If no food is within this radius it will move in a random direction. I observed this movement by placing it next to several dirt piles and checking if it changes direction every time it approaches one of these objects. I tested the food criteria by putting a piece of food in the stated proximity and watched if the sal would move towards that piece of food.

**class AggressiveSalmonella : public Bacteria**

* The aggressive salmonella followed the exact same pattern as the regular salmonella except there is a catch: if Socrates is within a 72 radius of agg sal, it will redirect it’s direction towards Socrates and can get stuck on a dirt pile while doing so. I tested this by simply placing Socrates within the 72 radius and checking if the aggressive salmonella will direct itself towards Soc.

**class eColi : public Bacteria**

* The eColi has the most diverse movement pattern out of the three. An eColi bacteria follows Socrates at all times. Its action can be described as an aggressive salmonella with a proximity of 255 pixels. Similarly, it can also get stuck on dirt piles when trying to locate Soc. When testing my game, I just made sure eColi followed my player at all times.

**class initialSpawnedActor : public Actor**

* Initially spawned actors have the least functionality out of all actors. They are pretty much constructed as just decorations with small features. Not many tests were run on this base class; instead; they were run on each derived class. The doSomething function was not declared pure virtual because a majority of the derived classes doSomething method do nothing.

**class Dirt : public initialSpawnedActor**

* Dirt piles are in the game to block the movements of bacterias and projectiles. I tested my dirt pile class by making sure it properly obstructs the movement of a bacteria and can be destroyed by a projectile.

**class Food : public initialSpawnedActor**

* Food is in the game for the purpose of bacteria duplication. I tested my food class by making sure it can be consumed by bacteria and would result in bacteria duplication when 3 are consumed.

**class Pits : public initialSpawnedActor**

* Pits are in the game to spawn bacteria. I tested my pits by making sure exactly 5 regular salmonella, 3 aggressive salmonella, and 2 e coli spawned out of every pit. After each pit has released all its bacteria, it would then disappear.

**class StudentWorld : public GameWorld**

* The student world class controls the overall flow of the game. When testing this class, I had to make sure all my actors were present visually when added to the game and deleted when they are set to dead. I did this by simply replaying my game several times and changing values like the number of pits, food, etc. that would spawn at the start of the game. After several plays, it gave me a good idea of how well my class maintained the flow of the game. In addition, I also tested the statistics visual located at the top of screen to make sure it was constantly being updated as bacteria die, goodies consumed, etc.