The following report describes what the functions of the classes I have written do, as well as how I may have tested them or made sure that they work as intended. The graphics for this project were given to me by my professor, Carey Nachenberg. Instead of including this information as comments I have written a report since there is so much of it.

A. StudentWorld (Derived from gameworld)
    1. The StudentWorld constructor just creates an object derived from GameWorld
    2. The StudentWorld Destructor calls the cleanup function to help destruct all Actors and components of the game when the game ends (either by quitting or a game over). To test this I just had to play the game till I died and make sure there was no undefined behaviours when cleaning up (ie deleting socrates twice).
    3. init() is virtual because it is also defined in the base class, gameworld. In init() I create the game's setup, where I put the round's dirt, food, and pit objects (amounts determined based on level).I also initialize socrates by creating a socrates object so the player can play the game. To test if this all worked correctly I started the game and clicked f immediately to pause. I counted the food items on multiple levels (level 1 had 5, 2 had 10 etc.) and Dirts objects(level one had 180, level 2 had 160 etc.). Each level had 1 pit as specified by the spec. Additionally, Socrates was functioning perfectly and a pointer to him was successfully stored as a member variable.
    4. initObj is a function I added to help init(). Essentially, it will make sure that it initializes all food piles without any overlap with other food, dirt or pits, pits without any overlap with other pits, food or dirt, and dirt with overlap with dirt but nothing else. It adds every actor it creates to the allActors vector. I tested this when testing init() and seeing if everything initializes correctly after a life is lost, a level is beaten, and at the very beginning.
    5. move() will check if a level has been completed, and if so will return that. It constantly updates the game board after calling every actor in the games doSomething function. Additionally if any actors or Socrates die, move() will delete them from the game, and if Socrates dies it'll return game over. Finally move() will call addnewActors() to determine if a fungus or goodie should be added, and if so it'll do that. The only way to really test this function is to play the game extensively and make sure all parts are working right every time. Since objects were deleting when they should and the game was working right I think my move() function was working just fine.
    6. cleanUp() will delete socrates at the end of the game if not already deleted, as well as all other actors. It will then proceed to erase all elements of the vector allActors. This is to ensure there are no memory leaks. I know this function works because there are no undefined behaviors for deleting twice at the end of the game and I put a print statement to tell when each actor was deleted and every single one was.

7. addToAllActors function takes in a pointer to an Actor and adds it to the game's Actors stored in allActors. This function helps when creating Actors in classes outside of StudentWorld (ie when a bacteria reproduces).
8. damageableObjectAt returns true if there is any damageable object within SPRITE_RADIUS*2 of the given coordinates. This function will also set the pointer that is passed in by reference = to the object there so that the caller knows what it is damaging and can damage it. This function works because when my sprays are supposed to hit the Bacteria they do and they deal 2 damage every time.
9. overlap returns true if the x and y coordinates passed in are within SPRITE_RADIUS*2 of the passed in actor's x and y coords. I tested this function by giving assert statements for different locations that should be within or outside of the range to count as overlapping.
10. overlapWithSocrates basically just returns if overlap is true when the actor passed into overlap is used with the Actor being socrates. The x and y passed in are directly passed into overlap as well. This function is very helpful when determining if a goodie is overlapping with socrates or a bacteria with socrates etc.
11. overlapWithFood goes through allActors and if an object's isFood returns true it will call overlap with that actor to see if the x and y given were overlapping with a food object. This function is used when Bacteria are seeing if they are overlapping with food to eat it. Since they successfully can eat all the food that's there I know this function works correctly.
12. overlapWithDirt does the same thing as the other overlapWith functions except checks isDirt and then checks if overlapping. This function is tested when the bacteria are moving around. When they should run into a dirt object, regular salmonella will change directions and since that works correctly this function must be returning correct values.
13. findNearestFood will take in an object's coordinates as parameters, a maximum distance that the food can be at and two double passed in by reference. If there is a food Actor with the distance the function sets the x and y passed in by reference to the food's x and y, then checks if any of the other foods are closer and if so will set the x and y to the closest one's, then returns true. Or else it returns false. This function is tested when Salmonella lock onto the nearest food and go towards it. Since they can do that correctly I know this function works.
14. distActors returns the distance between two actors as a double. Originally I tested this using various asserts that should return true or false based on the number returned.
15. gameStats sets a string at the top of the game with all the stats (lives, level, flames, score etc.). It is called repeatedly in move so that the stats can update accordingly in the top.
16. getSocrates just returns a pointer to socrates. This is helpful when needing to alter his health or something, based on what goodie or fungus he runs into.

- ➢ I tested all of my StudentWorld functions on their own seeing if the right values were returned or if the game was affected correctly. Each function contains a description of how it was tested. I follow this pattern for many functions in the rest of my project and each function has written how it was tested if testing was necessary. Additionally since I was able to play the game smoothly with no problems I think my StudentWorld class works just fine.

B. Actor (derived from GraphObject)
    1. The Actor constructor will call the GraphObject constructor using an initializer list. It also initializes all the member variables based on the parameters that were inputted. I tested my constructor using print statements in it for when Actors were constructed to see if they were constructed correctly.
    2. The Actor destructor is empty, however it is defined as virtual since every derived class from Actor has its own destructor.
    3. getPointerToStudentWorld() returns a pointer to the Actor's StudentWorld it is in. This is useful later on when calling functions that are defined in StudentWorld.
    4. getHealth() returns the Actor's health. This is mostly helpful for Socrates and the Bacteria since other objects don't really have health, just lives. I still used it for Actors who died instantly by setting their health to 1 so that when hit they will die immediately. I call this function repeatedly for every actor in move when I check at the end of the function if any Actors have died.
    5. setHealth(int health) takes in an integer and will set the Actor's new health to that number. Generally when I called it it was to decrease an Actor's health after being damaged so I would call it like this:
       damagedAct->setHealth(damagedAct->getHealth() - damage);
    6. getLives() returns the amount of lives an Actor is. Essentially this tells me if an Actor is alive or dead.
    7. setLives(int lives) sets the lives of an Actor (normally setting to 0 when an Actor dies.
    8. The virtual functions:
        a. virtual bool isDamageable()
        b. virtual bool isFood()
        c. virtual bool isPit()
        d. virtual bool isSalmonella()
        e. virtual bool isBacteria()
        f. virtual bool isDirt()

       In Actor each of these virtual functions is going to return false however for some derived classes they will be overridden to return true to help identify what actions need to be taken (ie if isDirt delete spray Actor that hit dirt and delete the dirt item).
    9. doSomething() is Actor does absolutely nothing but is virtual and gets overridden in many classes (such as Projectile) based on what action there is. I chose to not make it pure virtual so I wouldn't have to redefine the function for dirt, food, and

other Actors that actually do nothing, however since many Actors have a doSomething that should do something separate, it must be virtual.

- ➢ This class doesn't have much to test. The getters and setters are pretty simple and all Actor objects I tried to create were constructed correctly.

B. Socrates (derived from Actor)
    1. Socrates constructs Socrates according to how the spec says he should be (health 100, 20 sprays, 5 flames, etc.) and uses the parameter for StudentWorld in order to call the Actor constructor with an initializer list
    2. Socrates destructor is empty since there are only primitive data types stored as member variables
    3. doSomething() will get the key that the player clicked during the tick and based on that will either move socrates, shoot a spray, shoot flames, or give socrates a spray if no key was hit. If the spray or flame keys are hit, socrates will add flames or spray, whichever was hit, to the AllActors array in StudentWorld using the addtoAllActors function in StudentWorld. I tested his do something by first moving him around during part one, and then by making sure he could shoot sprays at dirt and making sure the flames (all 16) show up when playing the game.
    4. Socrates's isDamageable() returns true since he can be dealt damage by bacteria and fungus
    5. getSprays() returns the amount of sprays socrates has. Will be used when displaying stats board on top and when determining if Socrates can still shoot sprays or if he is out.
    6. useSpray() decreases the amount of sprays Socrates has by 1.
    7. Get Flames returns the amount of flames Socrates has. Will be used when displaying stats board on top and when determining if Socrates can still shoot flames or if he is out.
    8. useFlame() decreases the amount of flames Socrates has by 1.
    9. gainFlame() increases Socrates's flames by 5. This will be called when a flame goodie is found by Socrates
    - ➢ I first tested Socrates by making sure his doSomething worked by moving around and shooting sprays and flames. Since the flames and sprays all showed up on the screen they were constructed correctly on this end and any potential errors that would've come would've been courtesy of the projectile class or StudentWorld, but not here.

C. Dirt (derived from Actor)
    1. Dirt's constructor takes in the location that Dirt will be created at and feeds that as well as a pointer to StudentWorld to the Actor Constructor initializer list
    2. Dirt's isDamageable returns true since dirt can be hit by sprays and flames
    3. isDirt will return true here since Dirt is a Dirt object. This will help when Salmonella check if overlapping with dirt when moving forward
    4. Dirt destructor is empty since there are no member vars stored by Dirt
    - ➢ There is quite literally nothing to test here, Dirt does nothing.

D. Projectile (derived from Actor)

1. The Projectile constructor takes in parameters for damage it does and pixels it will travel (32 for flame, 112 for spray) as well as location and direction to use in initializer list for Actor.
2. Projectile destructor is empty since there are only primitive data types stored as member variables
3. doSomething() for all projectiles will move the projectile forward 3 pixels as long as the number of pixels traveled is less than their maximums, or else they will setLives to 0. If a projectile comes into contact with another object that is damageable it should deal it damage based on how much damage it is supposed to (ie flame does 5, spray does 2), then it should set its lives to 0 and disappear. Additionally if the object is a bacteria and it dies then if a random int between 0 and 1 is 0 then it will create a food object at the location of the killed bacteria. I tested all of this functionality once I had bacteria up and running, but for moving and destroying damageable objects I just kept shooting at dirt objects right after creating Dirt.

➢ I tested this class by shooting both kinds of projectiles, sprays and flames, and making sure they both traveled the right distance and dealt the right damage (sprays do 2 and flames do 5). I also had to make sure that the projectiles, if they killed an Actor, made the right calls to StudentWorld in order to delete the object that was killed.

E. Sprays and Flames (derived from Projectile)
    1. Both constructors take in location to pass to initializer list of projectile, however flames also pass damage dealt as 5 while sprays only do 2
    2. Both destructors are empty since there are no member variables stored

    ➢ All functionality is implemented in Projectile, nothing to test

F. Food (derived from Actor)
    1. The Food constructor will take in the location of the food to pass to the Actor constructor using the initializer list.
    2. Food destructor is empty since there are no member vars stored by Food
    3. isFood will return true for Food. this will help later when checking if a Bacteria is overlapping with a food item so it can eat the food

    ➢ There is quite literally nothing to test here, Food does nothing.

G. droppedItem (derived from Actor)
    1. The droppedItem constructor creates a droppedItem given the x, y, and type of Item in the constructor and passes them to the Actor initializer list to create the droppedItem. It also initializes the lifetime of the item based on the parameter passed in.
    2. droppedItem destructor is empty since there are only primitive data types stored as member variables
    3. doSomething() is virtual and empty since every droppedItem's doSomething() will be different

4. lifeloss() decreases the lifetime by 1. Is normally called at the end of each tick to show a tick has passed and to see if the droppedItem is still alive or out of lifetime.
5. getLifetime() just returns how many ticks of life the droppedItem has left

➢ I tested my dropped Item class by first making sure my constructor was constructing the objects correctly with all the right characteristics by printing what should be happening and if when put on the game screen the droppedItem is in the right spot etc. There wasn't much else about the class to test since it only has a getter for m_lifetime and a decrementer for it.

H. Fungus (derived from droppedItem)
    1. The fungus constructor passes all its parameter to droppedItem through an initializer list for construction
    2. The Fungus destructor is empty since there are no member vars stored by Fungus
    3. Fungus's doSomething() will check if it is overlapping with socrates, and if so it will do him 20 damage and reduce the score by 50. Or else it checks it's lifetime left after reducing it by 1 to see if it needs to die.

    ➢ I tested my Fungus by having it drop down and making sure it died when it ran out of ticks. I also made sure that when Socrates passed the fungus Socrates lost 20 health and player lost 50 points.

I. ExtraLife (derived from droppedItem)
    1. The ExtraLife constructor passes all its parameter to droppedItem through an initializer list for construction
    2. The ExtraLife destructor is empty since there are no member vars stored by ExtraLife
    3. ExtraLife's doSomething() will check if it is overlapping with socrates, and if so it will grant him an extra life and increase the score by 500. Or else it checks it's lifetime left after reducing it by 1 to see if it needs to die.

    ➢ I tested my ExtraLife goodie by having it drop down and making sure it died when it ran out of ticks. I also made sure that when Socrates passed the goodie Socrates gained a life and the player gained 500 points.

J. FlameCharge (derived from droppedItem)
    1. The FlameCharge constructor passes all its parameter to droppedItem through an initializer list for construction
    2. The FlameCharge destructor is empty since there are no member vars stored by FlameCharge
    3. FlameCharge's doSomething() will check if it is overlapping with socrates, and if so it will grant him 5 extra flames and increase the score by 300. Or else it checks it's lifetime left after reducing it by 1 to see if it needs to die.

    ➢ I tested my FlameCharge goodie by having it drop down and making sure it died when it ran out of ticks. I also made sure that when Socrates passed the goodie Socrates gained 5 flames and the player gained 300 points.

K. RestoreHealth (derived from droppedItem)

1. The RestoreHealth constructor passes all its parameter to droppedItem through an initializer list for construction
2. The RestoreHealth destructor is empty since there are no member vars stored by RestoreHealth
3. RestoreHealth's doSomething() will check if it is overlapping with socrates, and if so it will restore his health to 100 and increase the score by 250. Or else it checks it's lifetime left after reducing it by 1 to see if it needs to die.

➢ I tested my RestoreHealth goodie by having it drop down and making sure it died when it ran out of ticks. I also made sure that when Socrates passed the goodie socrates he gained health till 100and player gained 250 points.

L. Bacteria (derived from Actor)
1. The Bacteria constructor takes in parameters for the location, IID and health and feeds them to the Actor initializer list. It sets the move plan distance to 0, food eaten to 0, and plays the sound of a Bacteria being born.
2. The Bacteria destructor is empty since there are only primitive data types stored as member variables
3. Bacteria's doSomething() is empty since each Bacteria has distinct doSomething()s
4. isDamageable and isBacteria return true. These two functions will help when looking at things like overlappingWithDamageableObject later on
5. resetMovement() will reset the movement plan to 10, and will reset the direction to a random direction
6. canMoveForward will check if it can move forward a unit without colliding with a dirt object, and if so it will. It does this for the amount of units specified when calling the function (ie if we want our salmonella to move forward 3 units each time we pass in 3 and it runs the loop that moves the salmonella forward in its direction 3 times, however E Coli we might just do two pixels instead since it's slower). I tested this function by placing a single regular salmonella, a single aggressive salmonella, and a single Ecoli, and watching them move around and into dirt objects. Each of the three should move smoothly, and follow their rules (ie salmonella change move plan after ten ticks). It is easy to observe this if you pause the game and play it tick by tick.
7. reproduce(int case) will be called when three foods have been eaten and the bacteria reproduces. The case is representative of which bacteria needs to be produced (1 for regSal, 2 for aggroSal, 3 for Ecoli). Basically it finds the right location to create the Bacteria and then creates it there and adds it to allActors using the function in StudentWorld. I tested my reproduce() by placing a single regular salmonella, then a single aggressive salmonella, then a single Ecoli and making sure all three ate three pizzas and reproduced into the right type of Bacteria near the location the third pizza was eaten.
8. salMoveAlgorithm() is the algorithm for how a regular Salmonella and aggressive Salmonella should move if not locked onto Socrates. Basically it moves the Bacteria in the direction of its current move plan 3 pixels, and if there's no move if

there is a food within 128 units it goes in that direction. If there isn't then it resets the move plan to 10 and direction to a random direction from 0-359. I tested it by observing and making sure salmonella would move towards food when 128 units away, counting how many ticks it takes till each move plan resets(making sure it is ten unless disturbed by a dirt in the way), and making sure the reset of the move plan was working after those ten ticks.

9. getMovePlan() and getFoodEaten() are just getters for the member variables which help make it easier to get the moveplan and foodEaten values.
10. setMovePlan and setFoodEaten help change a value to the parameter when a food is eaten or the move plan has changed.

➢ I tested all of my bacteria functions on their own by running them once I created the salmonella and ecoli and having values print out that need to be (ie amount of food eaten or distance returned, etc.) and making sure the numbers make sense.

M. regSalmonella (derived from Bacteria)
    1. The regSalmonella constructor takes in coords to pass into the Bacteria constructor along with its own characteristics (hp = 4, IID of Salmonella, etc.).
    2. The regSalmonella destructor is empty since there are no member vars stored by regSalmonella
    3. regSalmonella's doSomething() will check if still alive. If so if it is overlapping with socrates it deals Socrates 1 damage. Else if it is overlapping with a food object, eat it and if 3 foods have been eaten then reproduce. Finally call the move algorithm for Salmonella

    ➢ I tested my regular salmonella class by placing one at the location 120, 120 and placing food all over the place. Since it reproduced when eating 3, follows the nearest food, and changes the move plan every 10 ticks by slowing it down and running the game frame by frame.

N. aggroSalmonella (derived from Bacteria)
    1. The aggroSalmonella constructor takes in coords to pass into the Bacteria constructor along with its own characteristics (hp = 10, IID of Salmonella, etc.).
    2. The aggroSalmonella destructor is empty since there are no member vars stored by aggroSalmonella
    3. aggroSalmonella's doSomething() will check if it's still alive. If it is within 72 units of Socrates then the aggroSalmonella will set its move plan to move straight toward him. If it is running into a dirt object, the aggroSalmonella will get stuck in its spot. If it is overlapping with Socrates it deals Socrates 2 damage. Else if it is overlapping with a food object, eat it and if 3 foods have been eaten then reproduce. If it is not locked onto Socrates, the aggroSalmonella will call the Salmonella move algorithm passed down through Bacteria.

    ➢ I tested my aggroSalmonella by making sure it does everything regular salmonela does in all scenarios except when 72 pixels away from socrates. In which case it should follow and get stuck if a dirt is in between. I moved around and it followed me.

O. Ecoli (derived from Bacteria)

1. The Ecoli constructor takes in coords to pass into the Bacteria constructor along with its own characteristics (hp = 5, IID of Ecoli, etc.).
2. The Ecoli destructor is empty since there are no member vars stored by Ecoli
3. Ecoli's doSomething() will check if it's still alive. If it is overlapping with Socrates it deals Socrates 4 damage. Else if it is overlapping with a food object, eat it and if 3 foods have been eaten then reproduce. Finally if the Ecoli is within 256 pixels of Socrates, he moves directly in his direction. If it can't move in that direction it tries to move 10 degrees to the left ten times. If it fails to do that it gets stuck in place.

➢ I tested my Ecoli by spawning a single ecoli, having it eat three pizzas and making sure it reproduces. I also made sure it follows Socrates by moving around and made sure it gets stuck in dirt when appropriate by spawning it in such places where it should.

P. Pit (derived from Actor)

1. Pit's constructor will take in coords to pass to the Actor initializer list. It sets numRegSal to 5, numEcoli to 2, and numAggroSal to 3.
2. The Pit destructor is empty since there are only primitive data types stored as member variables
3. isPit() returns true, helps identify when checking the game is over if no pits or bacteria remain.
4. doSomething() will randomly choose a number from 0-49. If 0, it will spawn one of the Bacteria unless it runs out of them (hence the member variables). Each has an equal chance at spawning. If the pit is dead it will do nothing. If all ten Bacteria have spawned the pit will die.

➢ I tested my pit by playing the game and seeing if it spawn the right amount of each bacteria, then self destructs. I also made sure sprays and flames can't hit pits.