Sarah Mauricio

Prof. Nachenberg

CS32

27 Feb. 2020

**Project 3 Report**

1. A high-level description of each of your public member functions in each of your classes, and why you chose to define each member function in its host class; also explain why (or why not) you decided to make each function virtual or pure virtual. For example, “I chose to define a pure virtual version of the sneeze() function in my base Actor class because all actors in Kontagion are able to sneeze, and each type of actor sneezes in a different way.”
   1. StudentWorld
      1. StudentWorld(std::string assetPath);
      2. ~StudentWorld(); : This function deletes all of the dynamically allocated variables from the actor vector. It just calls cleanup to do this. This is virtual because destructors should always be virtual.
      3. virtual int init(); : This function adds all of the pits, food, dirt, and socrates into the game. It populates it with the correct amount and make sure none of the objects overlap, besides dirt with dirt. It is virtual because it redefines this method from GameWorld.
      4. virtual int move(); : This function calls the doSomthing() function for each one of the actors in the actor vector and socrates. It makes sure that if socrates dies the game ends, and if an actor dies during that tick, it gets deleted. It will also add the fungus and goodies to the game if the random number generated is 0. Lastly, it is responsible for the text on the top of the screen. It is virtual because it redefines this method from GameWorld.
      5. virtual void cleanUp(); : This function deletes all of the dynamically allocated actors. It is virtual because it redefines this method from GameWorld.
      6. void addActor(Actor\* a); : This function takes in a pointer to an actor and then adds this actor to the vector of actors. It is not virtual because other functions cannot redefine it.
      7. Socrates\* getSocrates(); : This function returns a pointer to the socrates object that is made in this class. It is not virtual because other functions cannot redefine it.
      8. void setCompleted(); : This function sets the level to complete. It is not virtual because other functions cannot redefine it.
      9. bool isComplete(); : This function returns whether or not the current level is complete (has no bacteria left in it). It makes sure the number of dead pits equals the current level number and the number of dead bacteria equals the amount of bacteria that were in the game at one point. It is not virtual because other functions cannot redefine it.
      10. bool pitEmpty(); : This function returns whether or not a pit is empty. It is not virtual because other functions cannot redefine it.
      11. void setPitEmpty(); : This function sets a pits status to empty, meaning it has no bacteria left to emit. It is not virtual because other functions cannot redefine it.
      12. bool overlap(int x1, int y1, int x2, int y2); : This function returns whether or not one object with coordinates ‘x1’ and ‘y1’ overlaps with another actor with coordinates ‘x2’ and ‘y2’. It is not virtual because other functions cannot redefine it.
      13. bool overlapsWithDirt(int x, int y); : This function returns whether or not the actor with coordinates ‘x’ and ‘y’ overlaps with a dirt actor by checking if each actor in the vector can block other objects. It is not pure virtual because other functions cannot redefine it.
      14. bool overlapsWithCanBeDamaged(int x, int y, int pointsDamaged); : This function returns whether or not the object with coordinates ‘x’ and ‘y’ overlaps with an object that can be damaged. It will then update the hit points of the actor that it overlaps with and sets it to dead if needed. It also dynamically allocates a new food object if the dead actor turns into food. It is not virtual because other functions cannot redefine it.
      15. int numOfDeadPits(); : This function returns the number of pits that have released all of the bacteria from it. It is not virtual because other functions cannot redefine it.
      16. void setNumOfDeadPits(); : This function adds one to the number of pits that have released all of the bacteria from it. It is not virtual because other functions cannot redefine it.
      17. int numOfDeadBacteria(); : The functions returns the number of bacteria that socrates has killed. It is not virtual because other functions cannot redefine it.
      18. void setNumOfDeadBacteria(); : This function adds one to the number of bacter that socrates has killed. It is not virtual because other functions cannot redefine it.
      19. int numOfBacteria(); : This function returns the number of bacteria that have/are/will be active in the game. It is not virtual because other functions cannot redefine it.
      20. void setNumOfBacteria(); : This function adds one to the number of bacteria that have/are/will be in the game. This could be called when a bacteria divides and spawns a new bacteria. It is not virtual because other functions cannot redefine it.
      21. bool overlapsWithFood(int x, int y); : This function returns whether or not an actor with coordinates x and y overlaps with a food object. If it does, then it will set that food object to dead. It is not virtual because other functions cannot redefine it.
      22. bool getCommon(int x, int y, int i, int dis); : This function is common to all of the functions that deal with overlap. It will return whether or not an object with coordinates x and y has a distance that is less that or equal to dis, meaning it will overlap.I t is not virtual because other functions cannot redefine it.
      23. int closeToFood(int x, int y, int& angle); : This function returns the distance to the from the coordinates x and y to the closest food object. It will also set angle to the angle between the object at coordinates x and y to the closest food object. You go through the whole actor vector to see which one is closest. It is not virtual because other functions cannot redefine it.
   2. Actor
      1. virtual void doSomething() = 0; : This function does nothing because it is pure virtual. I chose to make it pure virtual because all of the actors do something different in the doSomething() function and you cannot just make an actor.
      2. void setAlive(bool alive); : This function sets the alive state to what is passed into the parameter. For example, if false is passed in, the object will be set to dead. If true is passed in, it will be set to alive. It is not virtual because other functions cannot redefine it.
      3. bool getAlive(); : This function returns whether an actor is alive or dead. If it returns true, the actor is alive, otherwise it is dead. It is not virtual because other functions cannot redefine it.
      4. StudentWorld\* getStudentWorld() { return m\_studentWorld; } : This function returns a pointer to the current student world that the game exists in. It is not virtual because other functions cannot redefine it.
      5. Direction dir() { return m\_direction;} : This function returns what direction an actor is facing. It is not virtual because other functions cannot redefine it.
      6. virtual ~Actor() {}; : This function does nothing. The destructor in studentWorld does everything. It is virtual because is is good practice to make destructors virtual.
      7. int getHitPoints(); : This function returns how many hit points an object has. It is not virtual because other functions cannot redefine it.
      8. void updateHitPoints(int points); : This function adds ‘points’ hit points to the actors number of hit points. If adding ‘points’ hit points to the number of hit points results in the number of hit points being over 100, the number of hit points is set to 100. It is not virtual because other functions cannot redefine it.
      9. int returnMax(int a, int b); : This function returns the maximum integer between ‘a’ and ‘b’. It is not virtual because other functions cannot redefine it.
      10. bool overlapsWithSocrates(double x, double y); : This function returns whether or not an object with coordinates ‘x’ and ‘y’ overlap with socrates. It is not virtual because other functions cannot redefine it.
      11. virtual bool canBlock() = 0; : This function does nothing. It is pure virtual because each type of actor may or may not be able to block other actors.
      12. double distanceFromSocrates(int x, int y);
      13. virtual bool canBeDamaged() = 0; : This function does nothing. It is pure virtual because each type of actor may or may not be able to be damaged.
      14. int distanceFromCenter(int x, int y, Direction d, int pixels); : This function gets the x and y coordinates of an actor after getting the position in direction ‘d’ and moving ‘pixels’ pixels forward from ‘x’ and ‘y’. It will then return the number of pixels from this new location to the center of the circle. It is not virtual because other functions cannot redefine it.
      15. double angleTowardsSocrates(int x, int y); : This function returns the number of degrees an actor has to turn to be facing socrates. It does this by getting the difference in socrates x and an actors ‘x’ and socrates y and an actors ‘y’. It then calculates the arctan of the two differences and returns that value in degrees. It is not virtual because other functions cannot redefine it.
      16. int maxx(); : This function returns the m\_maxx variable. This is not virtual because nothing redefines it later on.
      17. virtual bool canBeEaten(); : This function returns false because most actors cannot be eaten. This is virtual because a child class of actor redefines it.
      18. int startHitPoints(); : This function returns the starting number of hit points for each object, which will stay the same for the entire game. It is not virtual because it is not redefined later.
   3. Socrates
      1. Socrates(StudentWorld\* myStudentWorld);
      2. virtual void doSomething(); : This function first checks if socrates is dead, and if he is nothing happens. Then he checks if there was any keyboard input and acts accordingly. If the right or left arrows are pressed, then he moves in that direction one time. If the space bar was pressed, he allocates a new spray and then shoots it in the direction he is facing. If enter is pressed, he allocates 16 flames objects circling around him in 22 degree increments.It is virtual because socrates redefined this function from the actor class, making his doSomething() special.
      3. void updateFlameThrower(int num); : This function adds ‘num’ flame throwers to socrates’ flame thrower count. It is not virtual because other functions cannot redefine it.
      4. int getFlameThrower(); : This function returns the number of flame throwers socrates has left. It is not virtual because other functions cannot redefine it.
      5. int getSprayCharges(); : This function returns the number of spray charges socrates has left. It is not virtual because other functions cannot redefine it.
      6. void setSprayCharges(int num); : This function adds ‘num’ spray charges to socrates’ spray charger count. It is not virtual because other functions cannot redefine it.
      7. virtual ~Socrates() {} This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
      8. void movement(Direction d); : This function moves socrates either to the right once or to the left once depending on the direction ‘d’. It then sets socrates direction so he is always facing the correct direction.It is not virtual because other functions cannot redefine it.
      9. virtual bool canBlock(); : This function returns false because socrates cannot block objects from moving onto him. It is virtual because it redefined actor’s version of this function.
      10. virtual bool canBeDamaged(); : This function returns true because socrates can be damaged (his hit points can be lowered). It is virtual because it redefined actor’s version of this function.
   4. Dirt
      1. Dirt(int dirtX, int dirtY, StudentWorld\* myStudentWorld);
      2. virtual void doSomething() {} : This function does nothing because dirt does not do anything. It is virtual becasue dirt redefined this method from actor, making it specialized.
      3. virtual ~Dirt() {} This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
      4. virtual bool canBlock(); : This function returns true because dirt objects can block other objects from moving onto it. It is virtual because it redefined actor’s version of this.
      5. virtual bool canBeDamaged(); : This function returns true because dirt objects can be damaged (their hit points can be lowered) if it encounters a spray or flame thrower. It is virtual because it redefined actor’s version of this function
   5. Bacteria
      1. Bacteria(int mid, bool alive, int bacX, int bacY, Direction dir, int depth, StudentWorld\* myStudentWorld, int hitPoints, int movementPlanDistance, int damagePoints);
      2. virtual void doSomething(); : This function makes sure the bacteria is dead, and if it is, it does nothing. It then checks if the bacteria overlaps with socrates. If it does, then it will update socrates hit points, subtracting the amount of points each type of bacteria lowers socrates hit points by. If it doesn’t overlap, it checks if it has eaten three food objects since it was born or divided last. If it does, it allocates a new actor of itself in the same location, otherwise it checks if it overlaps with a food object. If so, it increases the amount of food eaten and sets one food object to dead. Then, it checks if a section is true and if it does, a specialized action occurs. It is virtual because bacteria redefined actor’s version of doSomething(), making it specialized.
      3. virtual ~Bacteria() ; This function updates the number of dead bacteria using setNumOfDeadBacteria() from studentWorld. It is virtual because it is good practice to make destructors virtual.
      4. virtual bool canBlock(); : This returns false because it cannot block other actors from moving onto it. It is virtual because it redefined actor’s version of this, making it specialized.
      5. virtual bool canBeDamaged(); : This returns true because it can be damaged. It is virtual because it redefined actor’s version of this, making it specialized.
      6. virtual void doDifferent1(int & movementPlanDistance) = 0; This function does nothing. It is pure virtual because each type of bacteria does something different in this part of the code.
      7. virtual void addNewActor(int x, int y) = 0; : This function does nothing. It is pure virtual because when each type of bacteria allocates a new object of itself, it looks different.
      8. virtual void aSalmonellaDifferent(bool& a) { a = true; }; : This function just makes a true so that the code will go into one of the if statements in doSomething(). It is virtual because one of its child actors will redefine this function.
   6. EColi
      1. EColi(int eColiX, int eColiY, StudentWorld\* myStudentWorld);
      2. virtual ~EColi() {}: This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
      3. virtual void doDifferent1(int& movementPlanDistance); : This function checks if it is less than or equal to 256 pixels away from socrates. If it is, it gets the directional angle toward socrates using angleTowardSocrates(). It then tries to move in this direction by two pixels. If it can't, it increases the angle by 10. It is virtual because it is EColi’s specialized version of what occurs in doSomthing() that is different from the other bacteria.
      4. virtual void addNewActor(int x, int y); : This function allocates a new eColi object into the game and adds it to the actor vector. It is virtual because it redefined bacteria’s version of this.
   7. Salmonella
      1. Salmonella(int mid, int salX, int salY, StudentWorld\* myStudentWorld, int hitPoints, int damagePoints);
      2. virtual ~Salmonella() {} : This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
      3. virtual void doDifferent1(int& movementPlanDistance); : This function gets the position three pixels forward from the current x and y. If the movement plan distance is greater than 0, it checks if it overlaps with dirt or is outside of the circle with the new x and y values. If either is true, it sets the direction to a random angle and then sets the movement plan distance to 10. Otherwise it moves 3 pixels in the current direction. If the movement plan distance is less than or equal to 0, it checks if there is food within 128 pixels. If there is, setDirection to a number between 0 and 359 and set the movement plan distance to 10. Otherwise, check if moving three pixels forward overlaps with dirt. If it does, setDirection to a number between 0 and 359 and set the movement plan distance to 10, and if it doesn't, move 3 pixels toward the food. It is virtual because it is Salmonella’s specialized version of what occurs in doSomthing() that is different from the other bacteria.
      4. virtual void addNewActor(int x, int y) = 0; : This function does nothing. It is pure virtual because when each type of salmonella allocates a new object of itself, it looks different.
   8. RSalmonella
      1. RSalmonella(int rSalX, int rSalY, StudentWorld\* myStudentWorld);
      2. virtual ~RSalmonella() {} : This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
      3. virtual void addNewActor(int x, int y); This function allocates a new RSalmonella object into the game and adds it to the actor vector. It is virtual because it redefined bacteria’s version of this.
   9. ASalmonella
      1. ASalmonella(int aSalX, int aSalY, StudentWorld\* myStudentWorld);
      2. virtual ~ASalmonella() {} : This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
      3. virtual void addNewActor(int x, int y); : This function allocates a new RSalmonella object into the game and adds it to the actor vector. It is virtual because it redefined bacteria’s version of this.
      4. virtual void aSalmonellaDifferent(bool& a); : This function checks if the distance from socrates is less than or equal to 72 pixels. If so, if will move forward in the direction of socrates as long as it does not overlap with dirt. It will then set ‘a’ to false so it will not go into the one if statement in doSomething() for bacteria. It is virtual because it redefines bacteria’s version of this function.
   10. Projectile
       1. Projectile(int mid, bool alive, int proX, int proY, Direction dir, int depth, StudentWorld\* myStudentWorld, int maxTravel, int decAmount);
       2. virtual void doSomething(); : This function checks if the projectile is dead and if it is, does nothing. It then checks to see if it overlaps with an actor that can be damaged. If it does, it sets its status to dead, otherwise it moves SPRITE\_RADIUS \* 2 pixels forward until the max distance it can travel is reached. It is virtual because projectile redefined actor’s version of doSomething(), making it specialized.
       3. virtual ~Projectile() {}; : This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
       4. virtual bool canBlock(); : This returns false because a projectile cannot block another actor from moving onto it. It is virtual because it redefined actor’s version of this function.
       5. virtual bool canBeDamaged(); : This returns false because a projectile cannot be damaged. It is virtual because it redefined actor’s version of this function.
   11. Spray
       1. Spray(int sprayX, int sprayY, Direction dir, StudentWorld\* myStudentWorld);
       2. virtual ~Spray() {} : This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
   12. Flame
       1. Flame(int flameX, int flameY, Direction dir, StudentWorld\* myStudentWorld);
       2. virtual ~Flame() {} : This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
   13. Goodie
       1. Goodie(int mid, bool alive, int gooX, int gooY, Direction dir, int depth, int lifetime, StudentWorld\* myStudentWorld, int hitPoints, int updatePoints);
       2. virtual void doSomething(); : This function checks if the goodie is dead and if it is, does nothing. It then checks to see if it overlaps with socrates. If true, if increased the score as specified by each goodie. It then sets the goodie to dead and does a specialized actor determined by the specific type of goodie. It will set to dead if the number of ticks is greater than the goodie’s lifetime. It is virtual because goodie redefines actor’s version of this function.
       3. virtual void doDifferent() = 0; : This function does nothing. It is pure virtual because each goodie has a specialized version of this and you cannot make just a goodie.
       4. virtual ~Goodie() {}; : This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
       5. virtual bool canBlock(); : This returns false because it cannot block other objects from moving onto it. It is virtual because it redefines actor’s version of this.
       6. virtual bool canBeDamaged(); : This returns true because it can be damaged. It is virtual because it redefines actor’s version of this.
   14. RestoreHealth
       1. RestoreHealth(int rhX, int rhY, StudentWorld\* myStudentWorld);
       2. virtual void doDifferent(); : This function sets socrates’ health/hit points back to 100. It is virtual because it is the specialized version of the function needed in goodie’s doSomething(). It redefined goodie’s version of this.
       3. virtual ~RestoreHealth() {} : This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
   15. FlameThrower
       1. FlameThrower(int flX, int flY, StudentWorld\* myStudentWorld);
       2. virtual void doDifferent(); : This adds 5 flame throwers to socrates flame thrower count. It is virtual because it is the specialized version of the function needed in goodie’s doSomething(). It redefined goodie’s version of this.
       3. virtual ~FlameThrower() {} : This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
   16. ExtraLife
       1. ExtraLife(int elX, int elY, StudentWorld\* myStudentWorld);
       2. virtual void doDifferent(); : This function increases socrates lives by one. It is virtual because it is the specialized version of the function needed in goodie’s doSomething(). It redefined goodie’s version of this.
       3. virtual ~ExtraLife() {}; : This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
   17. Pit
       1. Pit(int pitX, int pitY, StudentWorld\* myStudentWorld);
       2. virtual void doSomething(); : This function checks if it has released all of the bacteria, and if it has, it sets the pit to empty, updates the number of dead pits, and sets the pit to dead. It then generates a random integer between 1 and 50. If the number is 40, it will continue to add either a aSalmonella, Salmonella, or EColi to the game and add it to the actor vector until there is none left in the pit. This is virtual because it redefined actor’s version of this function.
       3. virtual ~Pit() {} : This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
       4. virtual bool canBlock(); : This returns false because it cannot block other objects from moving onto it. It is virtual because it redefines actor’s version of this.
       5. virtual bool canBeDamaged(); : This returns false because it cannot be damaged. It is virtual because it redefines actor’s version of this.
   18. Fungus
       1. Fungus(int fungusX, int fungusY, StudentWorld\* myStudentWorld);
       2. virtual void doSomething();
       3. virtual ~Fungus() {}: This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
       4. virtual bool canBlock(); : This returns false because it cannot block other objects from moving onto it. It is virtual because it redefines actor’s version of this.
       5. virtual bool canBeDamaged(); : This returns true because it can be damaged. It is virtual because it redefines actor’s version of this.
   19. Food
       1. Food(int foodX, int foodY, StudentWorld\* myStudentWorld);
       2. virtual void doSomething() {}: This function does nothing because food does not do anything. It is virtual becasue dirt redefined this method from actor, making it specialized.
       3. virtual ~Food() {} : This function does nothing. The destructor in studentWorld does everything. It is virtual because it is good practice to make destructors virtual.
       4. virtual bool canBlock(); : This returns false because it cannot block other objects from moving onto it. It is virtual because it redefines actor’s version of this.
       5. virtual bool canBeDamaged(); This returns false because it cannot be damaged. It is virtual because it redefines actor’s version of this.
       6. virtual bool canBeEaten(); : This function returns true because a food object can be eaten. It is virtual because it redefines an earlier version of this function.
2. 2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. “I didn’t implement the Flame class.” or “My aggressive salmonella doesn’t work correctly yet so I treat it like a regular salmonella right now.”
   1. The score on top of the screen is not the proper distance apart according to the sample game provided. I could not figure out the correct distance between and I tried multiple things to get it right, but they did not work.
3. 3. A list of other design decisions and assumptions you made; e.g., “It was not specified what to do in situation X, so this is what I decided to do.”
   1. It did not say that the bacteria had to face socrates when he moves and they are stuck behind a dirt, but that is what the example did, so I did that as well.
   2. It was not specified what the max amount of health socrates could have, so I capped the hit points to 100 because that is what the sample game did and 100 is usually the max amount of health a player can have in any game.
   3. It didn’t specify how many pixels the aggressive salmonella should move toward the food if it didn’t overlap with a dirt, so I chose 2 pixels.
4. 4. A description of how you tested each of your classes (1-2 paragraphs per class).
   1. StudentWorld
      1. I checked that no objects overlapped with one another, besides dirt with other dirt, by making the overlap distance very big. When I did this, there was a big circle of emptiness around each object, besides dirt, showing that it will not overlap one actor with another.
   2. Actor
      1. I checked all of the things that returned a number by printing it out on the screen. I did this for each object in the actor vector and for socrates. I checked the overlapsWithSocrates by moving right by a goodie and making sure the goodie would disappear when this happened. I made sure the angleTowardsSocrates was correct by making objects face this angle at all time, and this resulted in them during and facing socrates depending on where he was in the grid. I checked that the canBeEaten was correct by making sure no other objects would disappear when a bacteria overlapped with them. I checked updateHitPoints by updating socrates score by 1 and making sure the score increased by 1. I also updated it by 100 and made sure it would only say 100. I made sure setAlive and getAlive worked by making an actor die and then making sure it would disappear from the game.
   3. Socrates
      1. I made sure that each button on the keyboard did the correct thing. I started by testing the left and right arrows. I made sure that he went all the way around the circle and stayed facing the correct direction. Then, I moved him around the other way. Then, I moved him in both directions to make sure he wouldn't skip over any spots on the grid. I then started testing the enter button. I made sure that when I pressed it, one spray would come out. I made sure that the spray count decremented by one by printing out the spray count and making sure it went down by one after shooting. I tested that it wouldn’t add any when the count was 0 or less by shooting the 20 times and making sure it wouldn’t shoot anymore. Next, I tested the flames. I counted the number of flames that appeared on the screen to make sure that it was 16. I printed out the number of flame throwers left and made sure it decremented by one when I pressed the enter button. I also made sure that it would only shoot the number of flame throwers by seeing when the count was 0 and making sure it wouldn't shoot again after that. I tested the increase in spray charges if the user didn’t press a button by printing out the count of spray charges. I pressed the spacebar 20 times and made sure the count increased back to 20, not over and not under.
   4. Dirt
      1. I made sure this object blocked others from moving onto it. I did this by having the objects move freely and made sure they did not move over a dirt pile.
   5. Bacteria
      1. I started by just making the bacteria move around. I made sure they did not go outside of the grid and would not overlap with dirt. I then had them go over socrates to make sure his hit points would decrease the proper amount for each type of bacteria. I made sure when the flame thrower or spray charge overlapped with it, the bacteria would die and be removed from the game. I also kept a count of the number of dead bacteria here and made sure it incremented every time a bacteria died by printing out the value on the screen. I made sure each bacteria did the right specified task. I did this by comparing my version with the sample version of this game. I made sure that the eColi would only move in the direction of socrates and would kill him on the spot if they could. I made sure the salmonellas chose a radon direction and would move in that direction as long as it did not overlap with dirt. I made sure that none of the dirt moved into a dirt by watching their actions during the game.
   6. EColi
      1. I made sure that the eColi would move direction toward socrates as long as it was not blocked by dirt. I tested this by moving socrates to different areas around this grid and making sure the eColi would follow. I also moved so it would get blocked by a dirt. I made sure that the eColi did not go over the dirt, but stopped there. I then moved one spot over and made sure that the eColi turned to face me and tried to move in socrates’ direction. I tested that it added a new ecoli object when it ate 3 food by printing the amount of ecoli and not adding any other bacteria. I then let it eat 3 food and made sure the count updated by one and there was one more in the grid.
   7. Salmonella
      1. I tested that only salmonella did this by making the score increase by a lot when it a salmonella went into the first else statement, but would only do this once. I watched the salmonella closely and saw when it was close to a food, and the score went up. I made sure the salmonella would not move outside the grid by letting the game sit, and not having them kill socrates or eat food and making sure there was the same amount after that time. I watched their movements and made sure they were similar to the sample game. I made sure that if it overlapped with dirt it would just go a random direction by watching the screen.
   8. RSalmonella
      1. The only thing specific to this function is adding a new regular salmonella to the game. I tested that it did this right by printing the amount of regular salmonella and making not adding any other bacteria. I then let it eat 3 food and made sure the count updated by one and there was one more in the grid.
   9. ASalmonella
      1. I tested that it did the one thing specific to aggressive salmonella by printing out “yes” on the screen. I checked that it did the right thing by watching the screen. I made sure that when it got stuck behind a dirt, it would always be facing socrates. I also made sure that if it was close enough to socrates, then it would attack him. I tested this by moving right by the pit and made sure that the aggressive salmonella would come right towards me, which it did. I tested that it added a new aggressive salmonella object when it ate 3 food by printing the amount of aggressive salmonella and not adding any other bacteria. I then let it eat 3 food and made sure the count updated by one and there was one more in the grid.
   10. Projectile
       1. I tested this by shooting directly at one of the bacterias. I made sure that if it overlapped with it, then that bacteria would be set to dead as long as it’s hit points were less than or equal to 0. I made sure that each dirt it encountered was destroyed. I made sure that when it encountered an object that could be damaged, it would set its alive to dead. I did this by shooting at a dirt and then making sure the spray disappeared as soon as it hit the dirt and did not go right through it. I made sure it continued moving as long as it did not encounter any actors by shooting the spray in open space and making sure it went to the center of the circle and then disappeared.
   11. Flame
       1. The only thing this function is responsible for is setting the location, direction, ID, depth, the maximum travel amount, and the number of hit points it should subtract from whatever actor it encounters. All of this happens in the constructor. I made sure it was right by looking at the screen and making sure it looked right.
   12. Spray
       1. The only thing this function is responsible for is setting the location, direction, ID, depth, the maximum travel amount, and the number of hit points it should subtract from whatever actor it encounters. All of this happens in the constructor. I made sure it was right by looking at the screen and making sure it looked right.
   13. Goodie
       1. I tested that it appeared for a random amount of time by measuring how long each goodie appeared on the screen. I also made sure that if socrates did not overlap with it, then it would disappear at some point. I did this by letting socrates just sit still. I checked that whenever socrates overlapped with it, it disappeared and updated the score the correct amount. I checked this by moving onto the goodie and making sure it disappeared as soon as socrates was close enough to it. I printed out the score to the screen and made sure it increased the proper amount when socrates overlapped with it. I also made sure that the specific doDifferent() was called depending on the type of goodie. I did this by looking at the type of gooding, overlapping with it, and then making sure the right thing occurred.
   14. RestoreHealth
       1. I made sure that whenever socrates overlapped with this goodie, his health went back up to 100. I did this by printing out his health on the screen and making sure that it increased the proper amount.
   15. FlameThrower
       1. I made sure that socrates flame thrower charges updated 5 when he overlapped with this goodie. I checked this by printing out his flame thrower charges on the screen and making sure it went up 5.
   16. ExtraLife
       1. I printed out the number of lives and made sure it increased by one when socrates overlapped with it.
   17. Pit
       1. I counted the number of bacteria that was released from it. I then made sure that the pit disappeared after releasing all of the bacteria. I made sure they spawned at random time increments by timing the difference in bacterias coming out of the pit. I made sure studentWorld was aware that the pit was empty by saying a level was complete after the pit was empty.
   18. Fungus
       1. I started by making sure it would disappear once socrates overlapped with it. I did this by moving onto it with socrates. I then made sure the score and hit points for socrates decreased by the proper amount. I did this by printing out these values on the screen. I also made sure socrates made a sound after and died if this set his hit points below 0. I made sure it was a random amount of time appearing for by timing how long each fungus appeared for and made sure it was for different time amounts. I also tested that it would disappear if socrates did not overlap with it by just staying in place.
   19. Food
       1. I tested this by making sure an object could move over it because it doesn’t block anything. I made it so that the bacteria would not eat the food, but just move over it, and they went over it freely. I also checked that it couldn’t be damaged by shooting the spray at it and making sure it did not disappear. I then made it so that the bacteria could eat food and checked that when it overlapped with food, it would disappear.