Reema Kshetramade

SID: 205328402

Project 3 Report

1. Function Descriptions

* Actor:
  + Actor(int imageID, int x, int y, Direction dir, int depth, StudentWorld\* studWorld) - Sets the private variables of Actor and calls the constructor the Actor's parent class GraphObject.
  + bool Actor::isDead() – this function returns whether the Actor object is alive. This is defined in Actor because every Actor in the game will have an alive status which it can return. This function is not virtual or pure virtual because every Actor would have the same implementation of the function.
  + void Actor::doSomething() – this function will be called in every tick of the game to make the actor do something specific to what actor it is. This is defined in the Actor class because every action will have a doSomething function. This is pure virtual because there is no base definition for what a generic actor can do. Also, because every actor must have a doSomething() implementation, making it pure virtual ensures that this is forced by the compiler.
  + void Actor::setAliveStatus(bool status) – this function sets the alive status of an actor to what is passed in as a parameter. This is defined in Actor because every Actor in the game will have an alive status that can be set. This function is not virtual or pure virtual because every subclass would have the same implementation of this function.
  + StudentWorld\* Actor::getWorld() – returns a pointer to the class’s StudentWorld pointer. This is defined in the actor class because every base class will need access to the game’s StudentWorld pointer. This is not virtual or pure virtual because every base class will need the same implementation.
  + bool isDamageable() – returns true or false depending on whether an actor of the class type is damageable by things such as a spray or flame. This is in the Actor class because every base class of actor will be either damageable or not damageable so it needs to be present in every class. This is defined virtual because other classes can have different implementations based on whether they are damageable or not. However, it is not pure virtual because since it is only returning a bool, there are only two options for the implementation of the function, so the default will just be to return false and when an actor is damageable, the class can override this implementation. This simply reduced unnecessary repetition when implementing the function in other classes.
  + bool Actor::isEdible() – returns whether the actor is edible. This is in the Actor class because every base class will be either edible or not edible. This is defined virtual because other classes will have different implementations of the function, depending on whether or not they are edible.
  + bool canOverlapWithDirtPile() – returns whether the actor can overlap with a dirt pile which is used when constructing the level. This is in the Actor class because every because every base class will either be able to overlap with a dirt pile or not. It is virtual because other base classes with have different implementations.
  + bool canBlockBacteria() – returns whether the actor can block bacteria from moving to that position. This is defined in the Actor class because every base class will either be able to block bacteria or not. It is virtual because base classes will have different implementations based on their properties.
  + void receiveDamage(int num) – does some form of damage to the actor which can be based on the number passed in. This is defined in the Actor class because there is overlap between damageable actors and other categories so there was not a clear category of only damageable actors. This is virtual because classes will have different implementations of it. It was not pure virtual because there is a common damage of simply becoming dead with any form of damage that was common to many base classes and I did not want to repeat this code.
* Socrates:
  + Socrates(StudentWorld\* studWorld) - sets the private variables of Socrates (number of flames, sprays, its positional angle around the game screen, and its number of health points) and calls the constructor of its base class Actor
  + void Socrates::doSomething() – this function checks whether Socrates is alive and completes some actions if he is. The actions include receiving a key input from the game player and either moving around the circumference of the game view or firing a projectile such as a spray or flame. If no user input is received, this function will do nothing. This is redefined in Socrates because he has unique actions that he needs to do every tick.
  + bool Socrates::isDamageable() – returns true and is redefined in Socrates because he is a damageable actor. It is virtual because it was originally defined in Actor and is being redefined.
  + void Socrates::receiveDamage(int num) – reduces Socrates hp by the number passed in. If Socrates health becomes below zero then he is set to dead and the sound of him dying is played. Otherwise the sound of Socrates being hurt is played.
  + void Socrates::changePoints(int num) – changes Socrates health points by the number passed in. This is in Socrates because he has points and it is not virtual because it is unique to Socrates.
  + void Socrates::changeFlame(int num) – changes Socrates number of flames by the number passed in. This is in Socrates because he has points and it is not virtual because it is unique to Socrates.
  + void Socrates::changeSpray(int num) – changes Socrates number of sprays by the number passed in. This is in Socrates because he has points and it is not virtual because it is unique to Socrates.
  + int Socrates::getPoints() – returns the number of health points Socrates has. This is in Socrates and is not virtual because it is unique to Socrates.
  + int Socrates::getFlames() – returns the number of flames Socrates has. This is in Socrates and is not virtual because it is unique to Socrates.
  + int Socrates::getSprays() – returns the number of sprays Socrates has. This is in Socrates and is not virtual because it is unique to Socrates.
* Dirt Pile:
  + DirtPile(int x, int y, StudentWorld\* studWorld) - calls the constructor of its parent class Actor
  + void DirtPile::doSomething() – the dirt pile does nothing in its doSomething function it just sits there. This is in DirtPile and is virtual because it is unique to DirtPile.
  + bool DirtPile::isDamageable() – this returns true because the dirt pile is damageable. This is in DirtPile and is virtual because it is unique to DirtPile.
  + bool DirtPile::canBlockBacteria() – returns true because it can block the bacteria. This is in DirtPile and is virtual because it is unique to DirtPile.
  + bool DirtPile::canOverlapWithDirtPile() – returns true because dirt piles can overlap with other dirt piles. This is in DirtPile and is virtual because it is unique to DirtPile.
* FlameSprayActor:
  + FlameSprayActor::FlameSprayActor(int imageID, int x, int y, Direction dir, int depth, StudentWorld\* studWorld, int dist, int dmg) – sets the private variable of FlameSprayActor, including its amount of damage, its maximum distance and the distance its traveled and calls its parent constructor of Actor
  + void doSomething() – checks if it is alive or not. If it is alive, checks if it overlaps with an actors and if it does sets that actor to dead and sets itself to dead, while playing the appropriate noises. Otherwise moves in its current direction until it reaches the maxDist where it becomes dead. This is in FlameSprayActor and is virtual because it is unique to FlameSprayActor.
* Flame:
  + Flame::Flame(int x, int y, Direction dir, StudentWorld\* studWorld) – calls its parent class Constructor FlameSprayActor
* Spray:
  + Spray::Spray(int x, int y, Direction dir, StudentWorld\* studWorld) – calls its parent class Constructor FlameSprayActor
* Food:
  + Food(int x, int y, StudentWorld\* studWorld) – calls parent class constructor which is Actor
  + void doSomething() – does nothing because Food does not do anything.
  + bool isEdible() – returns true because Food is edible. This is redefined and is virtual in Food because food is edible while other actors are not.
* Goodie:
  + Goodie::Goodie(int imageID, int x, int y, Direction dir, int depth, StudentWorld\* studWorld) – calls parent class Actor’s constructor and sets max lifetime.
  + void doSomething() – if it is dead, does nothing. Otherwise, checks whether it overlaps with Socrates and applies its boost (unique to each type of goodie), sets its alive status to false and plays a specific sound (again is unique to each type of goodie). This is defined in Goodie because each type of goodie that will derive from this class does a similar set of actions, only with varying boosts and sounds that are specified the derived class.
* HealthGoodie:
  + HealthGoodie(int x, int y, StudentWorld\* studWorld) – calls parent class Goodie’s constructor
* FlameGoodie:
  + FlameGoodie(int x, int y, StudentWorld\* studWorld) – calls parent class Goodie’s constructor
* LifeGoodie:
  + LifeGoodie(int x, int y, StudentWorld\* studWorld) – calls parent class Goodie’s constructor
* Fungus:
  + Fungus(int x, int y, StudentWorld\* studWorld) – calls parent class Goodie’s constructor
* Pit:
  + Pit(int x, int y, StudentWorld\* studWorld) – sets private variables and calls parent class Actor’s constructor
  + void doSomething() – if all the bacteria have been emitted, the pit is set to dead and does nothing else and the number of pits in StudentWorld is decreased. Then there is a 1 in 50 chance that the pit will emit a bacteria and a 10% chance to emit an Ecoli, 30% chance to emit an aggressive salmonella and 60% chance to emit a regular salmonella, chosen from the bacteria that are still left in the pit.
* Bacteria:
  + Bacteria(int imageID, int x, int y, Direction dir, int depth, StudentWorld\* studWorld, int hp) – sets private variables and calls parent class constuctor
  + void commonActions(bool doFirstPart, bool doSecondPart) – if doFirstPart is true, the bacteria will check if it has eaten 3 foods and will divide if it has and if it has not eaten 3 foods, it will check if it overlaps with a food and will increase the amount of food he has eaten. If doSecondPart is true, it will check if movementPlanDist is greater than 0, decrement movementPlanDist and attempt to move in its current direction. If it cannot move choose a random direction and reset the movement plan to the full amount. Otherwise, check if there is a food nearby and move towards it. This is defined in Bacteria because all the bacteria have these actions in common and is not virtual because it does not need to be redefined in the derived classes.
  + void receiveDamage(int num) – if the number is not 0, the bacteria’s health points are decreased by num. If the health points go below 0, then the bacteria is set to dead and a sound is played for the bacteria dying (unique for each bacteria) and it has a 50% chance of becoming food when it dies. This is in Bacteria because each bacteria does the same thing when it receives damage and it is virtual because it is redefining the receiveDamage function in Actor.
* RegularSalmonella:
  + RegularSalmonella(int x, int y, StudentWorld\* studWorld) – calls parent class constructor
  + void doSomething() – if it is not dead, checks if it overlaps with Socrates and damages him if so. Then does the common bacteria actions, doing selective parts based on whether it overlapped with Socrates. This is in RegularSalmonella and is virtual because it redefines a unique version of doSomething for the RegularSalmonella class.
* AggressiveSalmonella:
  + AggressiveSalmonella(int x, int y, StudentWorld\* studWorld) – calls parent class constructor
  + void doSomething() – if it is not dead, check if it overlaps with Socrates and damage Socrates accordingly. Checks if Socrates is within 72 pixels and attempts to move in that direction. Then it does the bacteria’s common actions accordingly depending on if it overlaps with Socrates and if it is able to find Socrates. This is in AggressiveSalmonella and is virtual because it redefines a unique version of doSomething for the AggressiveSalmonella class.
* EColi:
  + EColi(int x, int y, StudentWorld\* studWorld) – calls parent class constructor
  + void doSomething() – if it is not dead, check if it overlaps with Socrates and damage Socrates accordingly. Then it does the bacteria’s common actions accordingly depending on if it overlaps with Socrates. Then it tries to find Socrates if it is within 256 pixels and attemps to move in that direction. If it cannot move in that direction, then it chooses a random direction. This is in EColi and is virtual because it redefines a unique version of doSomething for the EColi class.
* StudentWorld:
  + StudentWorld(string assetPath) – set private variables and call parent class constructor
  + int init() – created the Socrates object and the Pit, Food, and Dirt Pile objects at random positions within the restriction on the game screen, making sure that only Dirt Piles are able to overlap with Dirt Piles. Initializes a list to hold pointers to all the actors in the game and adds all of the actors created to this list. This is in StudentWorld because this is the class which takes care of game movement and keeps track of the Actors in the game. It is virtual because it is redefining the init function defined in its parent class GameWorld.
  + int move() – calls the doSomething() function of Socrates and each actor in the game (kept in the list of Actor pointers which is a private member of the class). The function removes the dead characters from the list and deletes them from the program memory. The function also randomly determines whether to generate a goodie or fungus in this tick. The function also generates the status text for the game. This is in StudentWorld because this is the class which takes care of game movement and keeps track of the Actors in the game. It is virtual because it is redefining the move function defined in its parent class GameWorld.
  + void cleanup() – deletes all the characters in the game and removes them from the container which holds pointers to all the actors in the game. This is in StudentWorld because this is the class which keeps track of the Actors in the game. It is virtual because it is redefining the cleanup function defined in its parent class GameWorld.
  + bool checkOverlap(double x, double y, int toDamage, bool toEat) – checks all the actors in the game to determine whether any overlap with the actor at the position x and y and acts on those actors depending on the parameters of toDamage and toEat which specify what action to take. The function returns true if there is an overlap at that position with an actor that is either damageable or edible depending on the parameters, and returns false if not. This is in StudentWorld because this is the class which takes keeps track of the Actors in the game. It is not virtual because it is not redefining the function defined in its parent class.
  + bool checkOverlapSocrates(double x, double y, int damage) – checks whether the actor at position x, y overlaps with Socrates and applies the specified amount of damage to Socrates. It returns true if it does overlap with Socrates and false if not. This is in StudentWorld because this is the class which takes keeps track of the Actors in the game. It is not virtual because it is not redefining the function defined in its parent class.
  + bool checkOverlapBacteria(double x, double y, int angle, double distOfOverlap) – checks whether a bacteria at position x, y will overlap with an actor in the game that can block bacteria a distance of distOfOverlap away in the direction of angle. If there is overlap, the function returns true, and if there is not overlap, the function returns false. This is in StudentWorld because this is the class which takes keeps track of the Actors in the game. It is not virtual because it is not redefining the function defined in its parent class.
  + bool findFood(int x, int y, Direction& dir) – returns whether a Food object can be found within 128 pixels of the position x, y and changes dir to the direction to that Food object if that is true. This is in StudentWorld because this is the class which takes keeps track of the Actors in the game. It is not virtual because it is not redefining the function defined in its parent class.
  + bool findSocrates(int x, int y, int& finalX, int& finalY, Direction& dirSoc, int maxDist) – returns whether Socrates can be found within maxDist pixels of position x, y. If so, dirSoc is set to the direction needed to move in to move to Socrates and finalX and finalY are set to 3 pixels in the direction that Socrates is if the bacteria is not blocked by a dirt pile on this path. This is in StudentWorld because this is the class which takes keeps track of the Actors in the game. It is not virtual because it is not redefining the function defined in its parent class.
  + void addActor(Actor\* a) – adds a to the list of Actor pointers that StudentWorld contains. This is in StudentWorld because this is the class which takes keeps track of the Actors in the game. It is not virtual because it is not redefining the function defined in its parent class.
  + void changeNumBacteriaAliveBy(int num) – changes the number of bacteria alive, that is tracked as a private member variable of StudentWorld by num. This is in StudentWorld because this is the class which takes keeps track of the Actors in the game. It is not virtual because it is not redefining the function defined in its parent class.
  + void decNumPits() – reduces the number of pits that have bacteria to emit, that is tracked as a private member variable of StudentWorld by 1. This is in StudentWorld because this is the class which takes keeps track of the Actors in the game. It is not virtual because it is not redefining the function defined in its parent class.
  + void increaseSocratesFlame(int num) – increase Socrates number of flames by num. This is in StudentWorld so that other Actors (such as the goodies) can change the information of how many flames Socrates has without needing a pointer to Socrates. It is not virtual because it is not redefining the function defined in its parent class.
  + void restoreSocratesHealth() – restores Socrates’ health to the full amount. This is in StudentWorld so that other Actors (such as the goodies) can change the information of how Socrates’ health points has without needing a pointer to Socrates. It is not virtual because it is not redefining the function defined in its parent class.

1. Functionality Issues

* The Ecoli do not move completely correctly because they are able to generally follow Socrates but sometimes stop in one position for a long period of time before starting to move again.
* The status text does not display the score correctly when it it less than 0

1. Design Decisions and Assumptions

* Originally, I generated the random position of the Dirt Piles by creating a random angle from 0 to 359 degrees and a random radius from 0 to 120 pixels (the view radius) and placed the Dirt Piles at a position corresponding to this radius and angle. However, this resulted in the Dirt Piles being more densely distributed towards the center of the view and they did not seem evenly dispersed despite being placed randomly. So, I made the design decision to choose the position of the Dirt Piles by generating a random x and y position and checking whether it was within the allowable radius from the center of the view screen. This resulted in more evenly dispersed Dirt Piles.
* The spec contains a lowercase h for the health portion of the status bar; however, this is capital in the sample game so I made it capital as well.

1. Testing of Classes

* Actor: I tested the Actor class by checking that the Actors appear on the screen with the correct images. Also, I checked that the actors can be set to dead when necessary and you can access information about whether the actor is dead or not through its interface. I also checked that an instance of the Actor class cannot be made because it is an abstract class, as it has pure virtual functions.
* Socrates: I tested the Socrates class by checking that the Socrates doSomething() does the correct action corresponding to the user input, such as moving clockwise when the right key is pressed, moving counterclockwise when the left key is pressed, generating flames when the enter key is pressed, and generating a spray when the space key is pressed, or doing nothing when no key is pressed. I also tested the Socrates class by ensuring that its health is reduced by the correct amount when it comes in contact with a Bacteria or fungus and that it receives the correct boost when it comes into contact with each goodie. I also checked that the spray and flame numbers are tracked correctly and are decreased when Socrates uses a flame or spray projectile. I also tested that the movement for Socrates is correct and that it moves only around the circumference of the circle, a few degrees for each click. I also checked that the direction Socrates faces is correct even when it moves, meaning it always points into the center of the circle.
* DirtPile: I tested that the DirtPiles work correctly by ensuring that they block bacteria from moving when a bacteria overlaps with them. I also checked that the Dirt Piles do not overlap with the food and pits in the game, but are able to overlap with each other. I also checked that DirtPiles are able to be damaged, meaning that when they are hit by a projectile, they die and disappear from the game. I also ensured that the dirt piles do not take any actions themselves during each tick of the game.
* FlameSprayActor: I tested that the FlameSprayActor works by ensuring that during each tick that it is alive it moves in its current direction (specified at construction) for the max distance that each type is able to move. I made sure that once it reaches that distance it disappeared from the game screen.
* Spray: I tested that the Spray class works by ensuring that when Socrates presses space, a Spray object appears in front of him in the same direction he is facing and moves forward in a straight line toward the center of the screen. I also tested that it correctly damages objects that can be damaged by projectiles by ensuring that when it comes into contact with these damageable objects, their health points are reduced if they are bacteria or they die and are removed from the game.
* Flame: I tested that the Flame class works by ensuring that when Socrates presses enter, Flame objects appear circularly around him and move radially outwards and disappear after they reach a certain distance. I also tested that it correctly damages objects that can be damaged by projectiles by ensuring that when it comes into contact with these damageable objects, their health points are reduced if they are bacteria or they die and are removed from the game.
* Food: I tested that the Food class works by ensuring that when they come into contact with a bacteria (which are the only actors able to eat the food) that they are set to dead and disappear from the game screen. I also checked that the food do not overlap with other foods, pits, or dirt piles.
* Goodie: I tested that the Goodie class works by ensuring that they appear around the circumference of the game screen so that Socrates can overlap with them. I also checked that the goodies were being chosen randomly, so there was not just one or two types of goodies appearing and that the goodies with greater chance, such as the HealthGoodie, appear more frequently than the other goodies. I also checked that Socrates correctly receives the intended boost (or damage in the case of the fungus) for the goodie by checking the information reported in the status bar. I also checked that the goodies do not stay permanently in the screen and disappear after their lifetime expires. I also ensured that when Socrates overlaps with the goodie, they do die and disappear from the screen and the boost is applied to Socrates.
* HealthGoodie: I tested that the HealthGoodie class works by ensuring that its boost is correctly provided to Socrates, and that it increases the player’s score by 250 and restores Socrates health to full. I also ensured that it played the correct sound (the sound indicating Socrates picked up a boosting goodie) when picked up by Socrates and disappears when picked up by Socrates. I also checked that it appeared relatively at the highest frequency because it has the highest chance of being put in the game each tick.
* FlameGoodie: I tested that the FlameGoodie class works by ensuring that its boost is correctly provided to Socrates, and that it increases the player’s score by 300 and increases Socrates number of flames by 5. I also ensured that it played the correct sound (the sound indicating Socrates picked up a boosting goodie) when picked up by Socrates and disappears when picked up by Socrates. I also checked that it appeared relatively at the second highest frequency because it has the second highest chance of being put in the game each tick.
* LifeGoodie: I tested that the LifeGoodie class works by ensuring that its boost is correctly provided to Socrates, and that it increases the player’s score by 500 and increases the player’s number of lives by 1. I also ensured that it played the correct sound (the sound indicating Socrates picked up a boosting goodie) when picked up by Socrates and disappears when picked up by Socrates. I also checked that it appeared relatively at the lowest frequency because it has the lowest chance of being put in the game each tick.
* Fungus: I tested that the Fungus class works by ensuring that its boost is correctly provided to Socrates, and that it decreases the player’s score by 50 and decreases Socrates’ health points by 20. I also ensured that it played the correct sound when picked up by Socrates (the sound indicating Socrates got hurt) and disappears when picked up by Socrates. I also checked that it appeared relatively at the lowest frequency because it has the lowest chance of being put in the game each tick.
* Pit: I tested that the Pit class works by ensuring that it releases different kinds of bacteria randomly through the course of the game while it still has bacteria. I made sure that the pit released a finite number of bacteria, and did not continue to release bacteria after it emitted the maximum number each pit can.
* Bacteria: I tested that the Bacteria class worked by ensuring that the derived classes of bacteria all performed the common actions defined in bacteria correctly, such as eating food, dividing when eating food, and attempting to move. I also ensured that an instance of the Bacteria class could not be made because it is an abstract class because it does not implement every function in Actor. I also tested that the bacteria receive damage correctly, as specified by the Bacteria class, meaning that they are damaged the correct amount of points are when they have health points less than or equal to zero, they play a sound indicating that the bacteria has died, reduce the number of bacteria alive in the game (as tracked by StudentWorld), and have a 50% chance of becoming food after death.
* RegularSalmonella: I tested that the RegularSalmonella class worked by ensuring that the regular salmonella moved correctly, meaning they were blocked by dirt piles and moved either randomly or in the direction of food. I also tested that when they were damaged they played the correct sound indicating that a salmonella was damaged. I also tested that when they overlapped with Socrates, he received the correct amount of damage. I also checked that when they died, they played the correct sound indicating that a salmonella had died.
* AggressiveSalmonella: I tested that the AggressiveSalmonella class worked by ensuring that the aggressive salmonella moved correctly, meaning they were blocked by dirt piles and moved in the direction of Socrates, randomly, or in the direction of food. I also tested that when they were damaged they played the correct sound indicating that a salmonella was damaged. I also tested that when they overlapped with Socrates, he received the correct amount of damage for an aggressive salmonella. I also checked that when they died, they played the correct sound indicating that a salmonella had died.
* EColi: I tested that the EColi class worked by ensuring that the Ecoli moved correctly, meaning they were blocked by dirt piles and moved either randomly or in the direction of Socrates. I also tested that when they were damaged they played the correct sound indicating that an Ecoli was damaged. I also tested that when they overlapped with Socrates, he received the correct amount of damage. I also checked that when they died, they played the correct sound indicating that an Ecoli had died.
* StudentWorld: I checked that the game was initialized correctly, meaning that Socrates was created correctly and the food, pit and dirt piles were created correctly and distributed randomly throughout the screen and only dirt piles were able to overlap with other dirt piles. I also ensured that the functions checking the overlap between various actors worked correctly so that when an actor was damaged or eaten it correctly received that damage, meaning it either died or its health points were reduced. I also checked that the overlap was determined correctly by checking that visually the actors were overlapping when the action was taken. I also checked that the functions that attempted to find Socrates or the food worked correctly by checking visually that the bacteria were able to follow Socrates or find some food. I also checked that the class worked correctly by ensuring that the actors were all able to do something in each tick of the game and that all the actors were correctly kept track of and were deleted from memory and from the container with pointers to each object when they were dead. I also ensured that the function correctly randomly created and placed the goodies. I also checked that the class correctly kept track of whether all the bacteria were emitted and whether all the bacteria were killed.