Urvi Shah

Project 3 Report

1.

**Actor class**

Actor::Actor(int i, int x, int y, int dir, int d, StudentWorld\* sw)

Constructs Actor with position and direction by calling GraphObject and sets status of Actor to alive.

StudentWorld\* Actor::getWorld()

Returns a pointer to the current StudentWorld.

void Actor::reduceHitPoints(int n)

If an object does not have any hitPoints and another class tries to reduce its hitPoints, it should set the life to false. This function is virtual since any class with hitPoints needs to redefine the function.

void Actor::resetHitPoints(){};

void Actor::increaseFlame(int n){};

These functions are both in the Socrates class, but when a Socrates is passed as an Actor pointer, these functions are necessary. However, they do nothing in the Actor class and are only redefined in the Socrates class.

void Actor::setLife(bool status)

Sets Actor to alive or dead based on parameter. If the parameter is false, the Actor is dead.

bool Actor::getLife()

Returns true if Actor is alive and false if Actor is dead.

bool Actor::isDamageable()

Returns true if Actor can be damaged by a Spray object. This is virtual because Dirt and Bacterium objects will return true.

bool Actor::isEdible()

Returns true if Actor is a Food object. This is virtual because Food objects will return true.

bool Actor::isObstacle()

Returns true if Actor blocks movement of an object of the Bacterium class. This is virtual because Dirt objects will return true.

bool Actor::isBacteria()

Returns true if Actor is derived from the Bacterium class. This is virtual because Bacterium objects will return true.

**Socrates class**

Socrates::Socrates(StudentWorld\* sw)

Socrates constructor: Creates an object of Socrates and constructs the base Actor class using the image ID for Socrates along with its initial position and direction. Initializes number of charges that Socrates has and its health.

int Socrates::getHitPoints()

Returns Socrates' health. This is not virtual because no other class is derived from Socrates.

int Socrates::getSpray()

Returns number of Sprays that Socrates has left. This is not virtual because no other class is derived from Socrates.

int Socrates::getFlame()

Returns number of Flames that Socrates has left. This is not virtual because no other class is derived from Socrates.

void Socrates::doSomething()

Socrates takes its next action during the tick. Changes position or fires charges based on the key that is entered. This function is virtual for clarity for the user to know that it is defining Actor's doSomething() function.

void Socrates::reduceHitPoints(int n)

Redefines Actor's reduceHitPoints function to reduce Socrates' health and play the correct sounds if Socrates gets hurt or dies. This function is virtual for clarity for the user to know that it is defining Actor's reduceHitPoints function.

void Socrates::resetHitPoints()

Makes Socrates' health full again. This function is virtual for clarity for the user to know that it is defining Actor's resetHitPoints function.

void Socrates::increaseFlame(int n)

Increases number of flames that Socrates has. This function is virtual for clarity for the user to know that it is defining Actor's increaseFlame function.

**Dirt class**

Dirt::Dirt(int x, int y, StudentWorld\* sw)

Dirt constructor: Creates an object of Dirt and constructs the base Actor class using the image ID for Dirt along with it's intial position and direction.

void Dirt::doSomething()

Redefinition of Actor's doSomething function that is virtual for clarity. Does nothing because Dirt does not do anything.

bool Dirt::isDamageable()

Returns true for Dirt because it can be destroyed through sprays or charges. Virtual for clarity since it is a redefinition of Actor's function.

bool Dirt::isObstacle()

Returns true for Dirt because it blocks Bacteria movement. Virtual for clarity since it is a redefinition of Actor's function.

**Pit class**

Pit::Pit(int x, int y, StudentWorld\* sw)

Pit constructor: Creates an object of Pit and constructs the base Actor class using the image ID for Pit along with it's intial position and direction. Initializes number of bacteria that the pit holds.

void Pit::doSomething()

Pit's doSomething function that is virtual for clarity since it is redefining Actor's function. Releases bacteria into the game with a 1/50 chance.

**Food class**

Food::Food(int x, int y, StudentWorld\* sw)

Food constructor: Creates an object of Food and constructs the base Actor class using the image ID for Food along with it's initial position and direction.

void Food::doSomething()

This function is virtual for clarity because Actor's doSomething function is virtual and can be redefined. This function does nothing because Food doesn't do anything.

bool Food::isEdible()

Redefinition of Actor's function that is virtual for clarity. Returns true because Food objects can be eaten.

**Projectile class**

Projectile::Projectile(int i, int x, int y, int dir, int d, StudentWorld\* sw, int mtd)

Projectile constructor: Creates an object of a Projectile and constructs the base Actor class using the image ID for Projectile along with it's initial position and direction. Initializes maximum travel distance for every projectile.

virtual void doSomething() = 0;

This function is pure virtual because an object of this class is never created and so the Projectile class never does anything by itself.

void Projectile::attack(int hp)

This functionality is common for both the Spray and Flame classes. This function checks if the Projectile object overlaps with a Dirt object or Bacterium object and kills it if it exists. Moves forward until hitting the maximum distance it can travel if it does not hit any other objects. This function is not virtual because no other class redefines it.

**Flame class**

Flame::Flame(int x, int y, int dir, StudentWorld\* sw)

Flame constructor: Creates an object of a Flame and constructs the base Actor class using the image ID for Flame along with it's initial position and direction.

void Flame::doSomething()

Flame's doSomething function that is virtual for clarity because it is redefining Actor's virtual doSomething function. This function calls the attack function in its Base class which damages an object that it hits by the number that is passed in, moves the object forward, or kills the Flame.

**Spray class**

Spray::Spray(int x, int y, int dir, StudentWorld\* sw)

Spray constructor: Creates an object of a Spray and constructs the base Actor class using the image ID for Spray along with it's initial position and direction.

void Spray::doSomething()

Flame's doSomething function that is virtual for clarity because it is redefining Actor's virtual doSomething function. This function calls the attack function in its Base class which damages an object that it hits by the number that is passed in, moves the object forward, or kills the Flame.

**Goodie class**

Goodie::Goodie(int i, int x, int y, StudentWorld\* sw)

Goodie constructor: Creates an object of a Goodie and constructs the base Actor class using the image ID for Goodie along with it's initial position and direction. Initializes lifetime with lifetime of Goodies.

int Goodie::getLifetime()

Returns the lifetime of the Goodie. This function is not virtual because it is never redefined.

bool Goodie::isDamageable()

This function returns true because Goodies can be damaged by Projectile objects and it is virtual for clarity since it is a redefinition of Actor's function.

void Goodie::doSomethingGeneral(int points)

This function does functionality that is common to all Goodies. It increases the player's score by the parameter that is passed in. It determines if Socrates is overlapping the object and does an action according to the type of Goodie and kills the Goodie if its' lifetime is over.

virtual void doSomething() = 0;

This function is pure virtual because an object of this class is never created and so the Goodie class never does anything by itself.

virtual void doSomethingSpecific(Actor \*a) = 0;

This function is pure virtual because every class that is derived from this class implements this function for functionality that differs for each Goodie, but it is never needed by this class.

**RestoreHealthGoodie class**

RestoreHealthGoodie::RestoreHealthGoodie(int x, int y, StudentWorld\* sw)

RestoreHealthGoodie constructor: Creates an object of a RestoreHealthGoodie and constructs the base Actor class using the image ID for RestoreHealthGoodie along with it's initial position and direction.

void RestoreHealthGoodie::doSomething()

Calls the base class doSomethingGeneral function with the correct number of points that the player gets if it overlaps with the Goodie. This function is virtual for clarity because it is a redefinition of Actor’s virtual function.

void RestoreHealthGoodie::doSomethingSpecific(Actor \*a)

Resets health points for Socrates since this is what the RestoreHealthGoodie should do when it is picked up. This function is virtual for clarity because it is a redefinition of Goodie’s virtual function.

**FlameThrowerGoodie class**

FlameThrowerGoodie::FlameThrowerGoodie(int x, int y, StudentWorld\* sw)

FlameThrowerGoodie constructor: Creates an object of a FlameThrowerGoodie and constructs the base Actor class using the image ID for FlameThrowerGoodie along with it's initial position and direction.

void FlameThrowerGoodie::doSomething()

Calls the base class doSomethingGeneral function with the correct number of points that the player gets if it overlaps with the Goodie. This function is virtual for clarity because it is a redefinition of Actor’s virtual function.

void FlameThrowerGoodie::doSomethingSpecific(Actor \*a)

Increase's Socrates' Flame charge count by five since this is what the FlameThrowerGoodie should do when it is picked up. This function is virtual for clarity because it is a redefinition of Goodie’s virtual function.

**ExtraLifeGoodie class**

ExtraLifeGoodie::ExtraLifeGoodie(int x, int y, StudentWorld\* sw)

ExtraLifeGoodie constructor: Creates an object of a ExtraLifeGoodie and constructs the base Actor class using the image ID for ExtraLifeGoodie along with it's initial position and direction.

void ExtraLifeGoodie::doSomething()

Calls the base class doSomethingGeneral function with the correct number of points that the player gets if it overlaps with the Goodie. This function is virtual for clarity because it is a redefinition of Actor’s virtual function.

void ExtraLifeGoodie::doSomethingSpecific(Actor \*a)

Increases the number of lives that Socrates has since this is what the ExtraLifeGoodie should do when it is picked up. This function is virtual for clarity because it is a redefinition of Goodie’s virtual function.

**Fungus class**

Fungus::Fungus(int x, int y, StudentWorld\* sw)

Fungus constructor: Creates an object of a Fungus and constructs the base Actor class using the image ID for Fungus along with it's initial position and direction.

void Fungus::doSomething()

Calls the base class doSomethingGeneral function with the correct number of points that the player gets if it overlaps with the Goodie. This function is virtual for clarity because it is a redefinition of Actor’s virtual function.

void Fungus::doSomethingSpecific(Actor \*a)

Decreases Socrates' health by 20 since this is what the Fungus should do when it is picked up. This function is virtual for clarity because it is a redefinition of Goodie’s virtual function.

**Bacterium class**

Bacterium::Bacterium(int i, int hp, int x, int y, StudentWorld\* sw)

Bacterium constructor: Creates an object of a Bacterium and constructs the base Actor class using the image ID for Bacterium along with it's initial position and direction. Initializes a Bacterium object's health, movement plan, and number of food eaten.

virtual void doSomething() = 0;

This function is pure virtual because an object of this class is never created and so the Bacterium class never does anything by itself.

bool Bacterium::isDamageable()

This function returns true because Bacterium objects can be damaged by Projectile objects and it is virtual for clarity since it is a redefinition of Actor's function.

bool Bacterium::isBacteria()

This function returns true because Bacterium objects are bacteria and it is virtual for clarity since it is a redefinition of Actor's function.

void Bacterium::reduceHitPoints(int n)

This function reduces the health of the bacteria by the number that is passed in and it is virtual for clarity since it is a redefinition of Actor's function.

int Bacterium::getHitPoints()

This returns a Bacterium object's health and it is not virtual because it is not redefined.

void Bacterium::createNewBacterium(int n)

This function creates a new Bacterium object at a distance of the radius away from it's current value depending on the position that it is currently at. This function is not virtual because it is never redefined.

void Bacterium::increaseFood(int n)

This function increases the number of food that the object has eaten by the number passed in. This function is not virtual because it is never redefined.

int Bacterium::getFood()

Returns the total food that the Bacterium object has eaten. This function is not virtual because it is never redefined.

void Bacterium::increaseMovementPlan(int n)

Increases the movement plan of the object by the number that is passed in. This function is not virtual because it is never redefined.

int Bacterium::getMovementPlan()

Returns the movement plan of the Bacterium object. This function is not virtual because it is never redefined.

void Bacterium::checkOverlap(int hp, int type)

Checks to see if the Bacterium object overlaps with Socrates and reduces Socrates' health by hp if it does. Also checks to see if the Bacterium object has eaten enough Food to create a new object or if it is overlapping with a Food object. This function is not virtual because it is never redefined.

**EColi class**

EColi::EColi(int x, int y, StudentWorld\* sw)

EColi constructor: Creates an object of a EColi and constructs the base Actor class using the image ID for EColi along with it's initial position and direction.

void EColi::doSomething()

Tries to move EColi object in the direction of the player and does not move forward if it cannot move towards the player, until the player moves and it can. This function is virtual for clarity because it is a redefinition of Actor's function.

void EColi::reduceHitPoints(int n)

Reduces the health of the EColi object and kills it if it's health is below zero. There is a one in two chance that the EColi will create a Food object when it dies. This function is virtual because it is a redefnition of Bacterium's function.

**Salmonella class**

Salmonella::Salmonella(int i, int hp, int x, int y, StudentWorld\* sw)

Salmonella constructor: Creates an object of a Salmonella and constructs the base Actor class using the image ID for Salmonella along with it's initial position and direction.

void Salmonella::reduceHitPoints(int n)

Reduces the health of the Salmonella object and kills it if it's health is below zero. There is a one in two chance that the Salmonella will create a Food object when it dies. This function is virtual because it is a redefnition of Bacterium's function.

void Salmonella::runMovementPlan()

Since all Salmonella objects will move randomly at times this function moves the object in a random direction or moves it in the direction of its movement plan, choosing a random direction if it is blocked by an object. This function also moves objects towards food if possible. This function is not virtual because it is never redefined.

**RegularSalmonella class**

RegularSalmonella::RegularSalmonella(int x, int y, StudentWorld\* sw)

RegularSalmonella constructor: Creates an object of a Salmonella and constructs the base Actor class using the image ID for Salmonella along with it's initial position and direction.

void RegularSalmonella::doSomething()

RegularSalmonella objects mostly do what RegularSalmonella objects do so this function checks if the object overlaps with anything and moves according to the RegularSalmonella's runMovementPlan function

**AggressiveSalmonella class**

AggressiveSalmonella::AggressiveSalmonella(int x, int y, StudentWorld\* sw)

AggressiveSalmonella constructor: Creates an object of a AggressiveSalmonella and constructs the base Actor class using the image ID for AggressiveSalmonella along with it's initial position and direction.

void AggressiveSalmonella::doSomething()

This object moves similarly to its base class but if Socrates is nearby it will move towards Socrates so this function first tries to find Socrates and move towards it and if it cannot do that it begins to move according to how a Salmonella moves

**StudentWorld class**

int StudentWorld::init()

Initializes all the objects that show up at the beginning of the game. Creates Pit objects, Food objects and Dirt objects, choosing the number based on the level that the player is on. Creates the player object as well.

int StudentWorld::move()

Plays one tick of the game. Checks if any actors have died and removes them from the game if they have. Calls each Actor's doSomething function and resets game statistics to reflect current values. Adds any Goodies based on the chance that they have to be added.

void StudentWorld::cleanUp()

Deletes player from the game if it hasn't already been deleted and deletes all the other actors from the game and the vector.

Actor\* StudentWorld::getDamageableObject(int x, int y)

Returns true if there is a damageable object that overlaps with the given x and y position

Actor\* StudentWorld::getSocrates(int x, int y, int d)

Returns a pointer to Socrates if the given x and y position are less than a distance of d away from Socrates

Actor\* StudentWorld::findNearestFood(int x, int y)

Returns the Food object that is closest to the x and y position given and returns nullptr if it does not exist.

void StudentWorld::addSpray(int x, int y, int d)

Adds a Spray to the vector of actors with the given position and direction

void StudentWorld::addFlame(int x, int y, int d)

Adds a Flame to the vector of actors with the given position and direction

void StudentWorld::addBacterium(int x, int y, int type)

Adds a Bacterium to the vector of actors with the given position and type of Bacterium

void StudentWorld::addFood(int x, int y)

Adds a Food object to the vector of actors with the given position

bool StudentWorld::foodOverlap(int x, int y)

Checks if there is a Food object that overlaps with the object at the given position

bool StudentWorld::canMove(int x, int y)

Returns if an object can move to the given position and is not blocked by any obstacles

2.

My Salmonella have a tendency to remain in the top section of the circle and clump together. I didn’t have time to fix this because I had three midterms this week, but they seem to move correctly otherwise.

A sound is played at the beginning of the game before the player clicks enter which probably shouldn’t happen, but the sound goes away when the player starts the game.

3.

When certain objects overlapped each other and we had to choose one to delete it was not specified which object to delete, so I chose to delete the object at that position that was created first, as it comes first in my actors container.

In the spec, the h in health was lowercase in the text at the top of the game but it was capitalized in the actual game so I decided to capitalize it.

4.

**Actor**

I tested the Actor class by testing the functionality of all the classes derived from it. Since it is impossible for an object of an Actor class to be created since it is an abstract class, it works correctly if, when a derived class calls one of its functions, it does the correct things. Since derived classes die and get created correctly, I know that the functions that set the lives of the Actors work correctly and that the constructor works correctly. The Actor class also has functions that return key characteristics about Actors, and since the class is able to identify which objects are which, these functions also work correctly.

**Socrates**

I tested the Socrates class by testing the main functionality that a player should have. For the movement of Socrates I tested that it moved clockwise when the right key is pressed and counterclockwise when the left key is pressed by having it go around the circle and alternating keys as well to make sure it could differentiate correctly. I also watched to make sure the direction it was facing matched the given game. For the sprays and the flames I watched to make sure that the spray went in the direction that it faced and that the flames were in a circle around it. I tested that its health went down accurately along with it dying when its health reaches 0 by watching the score board and watching the bacteria to see how its health changes when the bacteria hits it and that the level ends when it dies.

**Dirt**

Since the dirt class doesn’t have much functionality, I knew its doSomething function already worked. I tested that it died at the correct times by watching when it was hit by a Spray or a Flame to make sure that it disappeared from the screen. In addition, I compared the spread of the Dirt objects to the spread in the provided game to make sure they matched and that I was distributing them correctly.

**Pit**

Most of the Pit object’s functionality is not visible. However, since it releases a number of Bacterium objects into the Pit, I watched to make sure that the correct number of each type of bacteria were released into the game. I had the program print out different values based on which bacteria was released into the game and since those values matched the number of each type of bacteria that is supposed to be in a Pit, I knew the game worked.

**Food**

Similar to Dirt, Food objects do not do much. I first tested that my Food objects were showing up in the correct places by comparing the locations that they were showing up to the locations in the game. They seemed pretty evenly distributed and were not overlapping with any Dirt objects so this was correct. I also checked that when any Bacterium objects came in contact with Food objects, they disappeared, which meant that they had died and been eaten, which is the correct functionality.

**Projectile**

I never create an object of this class because it is used as a base class for Flame and Spray. This class implements any functionality of the Flame and Spray classes that is common to both of them, so that is what I tested for this class. I made sure that both the Flame and the Spray went in the correct directions when they were created. I looked at the direction that Socrates was facing and if the projectile right in front went straight from that direction, my projectile worked correctly, which it did. In addition, when any projectile hit a Dirt, the Dirt object disappeared and the Bacterium object’s life decreased.

**Flame**

Since I knew that Projectile worked after testing it, I only had to test the functionality that differed for Flame. Since this was the creation of the Flames and the way that they were distributed, along with the number of points that Socrates lost when hit, that is what I tested. When I hit enter, I made sure that the Flames with the correct image were distributed. Then, I made sure that they were distributed in a circle around Socrates and that they traveled in the direction with which they were created, which they did. I also watched Socrates health to make sure that the player lost 5 points when hit by a Flame, which it correctly did.

**Spray**

Since I knew that Projectile worked after testing it, I only had to test the functionality that differed for Spray. Since this was the creation of the Spray along with the number of points that Socrates lost when hit, that is what I tested. When I hit enter, I made sure that the Spray with the correct image was distributed. Then, I made sure that the Spray was only created right in front of Socrates and not anywhere else. I also watched Socrates health to make sure that the player lost 1 points when hit by a Spray, which it correctly did.

**Goodie**

The Goodie class is a base class for all the objects that are goodies and the fungus, so an object is never created of the actual Goodie class. These objects have a lot of similar functionality, so I tested that to test this class. I made sure that, when a Goodie object appears in the game and Socrates overlaps with it, the Goodie object dies and gets removed from the game. In addition, since all the Goodie objects randomly appear on the edge of the game, I made sure that the objects only showed up on the edge. I also made sure that the objects disappear after a lifetime of ticks if Socrates does not overlap with the object. Each Goodie also changes the player’s point values by a certain amount so I made sure to check that the point values on the scoreboard are changing.

**RestoreHealthGoodie**

The main functionality that is distinct for the RestoreHealthGoodie class is that the image is different and the action that is taken when Socrates overlaps with the object. To test this, I made sure that the image of this Goodie matched with the image for the same Goodie in the given game. Since this Goodie resets Socrates’ health, I watched the scoreboard at the top of the game that shows Socrates’ health. When Socrates overlapped with the Goodie, the health changed back to 100, which is how I knew it worked. In addition, I check to make sure the points change by the correct point value.

**FlameThrowerGoodie**

The main functionality that is distinct for the FlameThrowerGoodie class is that the image is different and the action that is taken when Socrates overlaps with the object. To test this, I made sure that the image of this Goodie matched with the image for the same Goodie in the given game. Since this Goodie adds another Flame charge to Socrates’ storage, I waited for this Goodie to appear, and when Socrates overlapped with the Goodie, I made sure that the scoreboard reflected that Socrates had one more charge. I also tested this by making Socrates run out of Flame charges and then picked up a Goodie and tested that the player could shoot Flame five more times. In addition, I check to make sure the points change by the correct point value.

**ExtraLifeGoodie**

The main functionality that is distinct for the ExtraLifeGoodie class is that the image is different and the action that is taken when Socrates overlaps with the object. To test this, I made sure that the image of this Goodie matched with the image for the same Goodie in the given game. When Socrates overlaps with this Goodie, the player should get one more life. I tested this by watching the scoreboard and making sure that it reflected one additional life when Socrates overlapped with the Goodie. In addition, I played the game and made sure that it allowed the player to play the game with Socrates dying four times instead of three times if the player picked up one ExtraLifeGoodie. In addition, I check to make sure the points change by the correct point value.

**Fungus**

The main functionality that is distinct for the Fungus class is that the image is different and the action that is taken when Socrates overlaps with the object. To test this, I made sure that the image of this Goodie matched with the image for the same Goodie in the given game. When Socrates overlaps with this Goodie, Socrates’ health should go down by 20 points and I make sure that this is happening by checking the scoreboard to make sure the player’s health is lower. In addition, I check to make sure the points change by the correct point value.

**Bacterium**

This class serves as a base class for all the bacteria so its functionality is based on the similarities of all the bacteria. The common functionality is mainly with the objects that they are blocked by. The objects that block Bacterium objects are Dirt objects and the edge of the circle in the game. I checked that the bacteria followed these restrictions by making sure they never went outside of the game circle and that they went in random directions if they had a Dirt object in front of them. In addition, if a Bacterium object overlaps with Socrates, it will lower Socrates’ health points by a certain amount based on the type of bacteria. Since all the Bacterium objects lower the health points of the player, this functionality works correctly.

**EColi**

The functionality that differs for EColi is that they go directly towards the Socrates at all times and do not try to move in another direction if given the opportunity. This means that they occasionally get blocked by Dirt objects and do not move since they are trying to get to the player. I tested this by moving Socrates around the circle and making sure that the EColi object changed the direction that it was moving based on Socrates’ position.

**Salmonella**

This class serves as a base class for the two Salmonella objects so its functionality is based on the similarities of these objects. The common functionality is mainly in the way that they move when they are not moving towards Socrates. The Salmonella objects went in random directions when faced with the obstacles and when they were not near Socrates. They would try to go in this direction until they were blocked again, and they would then proceed to choose another random direction. I also made sure that the correct image showed up for the Salmonella class since all objects of the Salmonella class have the same image ID.

**RegularSalmonella**

The RegularSalmonella class always moves in random directions which is accomplished by the Salmonella class. I tested this class by making sure that the correct number of points was removed from the health points of Socrates when the RegularSalmonella overlapped with it.

**AggressiveSalmonella**

The AggressiveSalmonella class differs from the base Salmonella class because when the Socrates object is nearby, it will try to go towards it. I tested this functionality by moving slightly away from Socrates and checking that some of the Salmonella objects are coming towards the object. Then, I moved away from the Salmonella objects and made sure that they continued to move in random directions as they are supposed to when Socrates is far enough away from the objects.

**StudentWorld**

I tested the StudentWorld class by testing the init(), move(), and cleanUp() function. I know the init() function works because I made sure that every object is appearing on the screen correctly. I tested the move() function by making sure that all the Actors doSomething() functions are called. Since all the Actors behave correctly, I know that their functions are being called. In addition, the Actors are being deleted at the correct times and do not remain for longer than one tick if they have died in that tick, which means that the move function is deleting Actors as their lives are being set to false. I tested the cleanUp function by making sure that there are no memory leaks. All of my pointers are deleted when a level ends since they do not show up when the next level starts, and the program never tries to delete something that already exists, so I know that my cleanUp function behaves correctly. I tested the public functions within the StudentWorld class by testing the classes that are derived from the Actor class because these are the classes that use the public functions by StudentWorld. Since I tested the functionality of all the classes in Actor.cpp, I know that the public functions in StudentWorld work because without their functionality being correct, the Actor classes would not work.