Michael Huang

004935096

Project 3 report

Class Design Overview:

Actor

Wall

Agent

ActivatingObject

Exit

Person

Pit

Landmine

Zombie

Flame

Goodie

Citizen

Penelope

Vomit

VaccineGoodie

SmartZombie

DumbZombie

GasCanGoodie

LandmineGoodie

Class member function descriptions:

**Actor** - abstract base class

virtual void Actor::doSomething() = 0;

All actors must have the ability to do something during a tick of the game. This function is pure virtual because it must be implemented by all derived classes, since all actors must be able to do something each tick of the game.

virtual bool Actor::canBlockMovement() const;

Actors may have the ability to block other actors from moving. It should return true for actors that can block movement and false otherwise. This function is virtual since some actors may be able to block movement, which necessitates differing implementation from its default return value of false.

virtual bool Actor::canBlockFlame() const;

Actors may have the ability to block flames specifically. It should return true for actors that can block flames and false otherwise. This function is virtual since some actors may be able to block flames, which necessitates differing implementation from its default return value of false.

virtual void Actor::activateIfAppropriate(Actor\* a);

Actors may have the ability to elicit some sort of response on other actors. If an actor does elicit some response on other actors, implement this function to act on a passed through the parameter. This is a virtual function because it is expected for different types of actors to have different implementations of this function. This function returns by default.

virtual void Actor::useExitIfAppropriate();

Actors may be able to exit the level through an exit. This is a virtual function because different actors have different implementations of this function. This function returns by default.

virtual void Actor::dieByFallOrBurnIfAppropriate();

Actors may have the capability of dying by falling or burning. This is a virtual function because different actors have different implementations of this function. This function returns by default.

virtual void Actor::beVomitedOnIfAppropriate();

Actors may be able to get vomited on. This is a virtual function because different actors react to being vomited on differently, necessitating different implementations of this function. This function returns by default.

virtual void Actor::pickUpGoodieIfAppropriate(Goodie\* g);

Actors may be able to pick up goodies. If an actor can, it should do something with g passed through the parameter depending on the actor and the goodie. This is a virtual function because actors could react to picking up goodies differently, necessitating different implementations of this function. This function returns by default.

virtual bool Actor::triggersZombieVomit() const;

Some actors trigger zombies to vomit on them. It should return true if a zombie can vomit on it and false otherwise. This is a virtual function because if a zombie can vomit on an actor, it must reimplement the function and change its default return value of false.

virtual bool Actor::triggersOnlyActiveLandmines() const;

Some actors trigger active landmines upon stepping on them. It should return true for actors that trigger landmines and false otherwise. This is a virtual function because actors that trigger landmines must reimplement the function and change its default return value of false.

virtual bool Actor::threatensCitizens() const;

Some actors are threats to citizens. It should return true for actors that are threats to citizens and false otherwise. This is a virtual function because some actors that threaten citizens must reimplement the function and change its default return value of false.

bool Actor::isAlive() const;

All actors have a state of either being alive or dead. This function is an accessor that returns whether or not the actor is alive. This function is not virtual because its implementation is the same throughout all actors.

void Actor::setAlive(bool state);

All actors have an ability to set their alive status. This function sets the actor's alive state to the boolean value passed into the parameter. This function is not virtual because its implementation is the same throughout all actors.

StudentWorld\* Actor::getWorld() const;

All actors have a corresponding StudentWorld that they belong to. This function returns the private member storing a pointer to the actor's StudentWorld. This function is not virtual because its implementation is the same throughout all actors.

**Wall**

virtual void Wall::doSomething();

Walls don't do anything but exist. This function automatically returns.

virtual bool Wall::canBlockMovement() const;

Walls can block movement. This function returns true;

virtual bool Wall::canBlockFlame() const;

Walls can block flames. This function returns true;

**ActivatingObject** - abstract base class

virtual void ActivatingObject::activateIfAppropriate(Actor\* a) = 0;

ActivatingObjects must be able to be activated. This is a pure virtual function because all ActivatingObjects must have implementation of what to do if they are activated.

**Exit**

virtual void Exit::doSomething();

Checks for overlap with other actors in its StudentWorld and activates if appropriate. This is a virtual function for clarity since it is redefined from Actor.

virtual void Exit::activateIfAppropriate(Actor\* a);

Makes a exit the world. This is a virtual function for clarity since it is redefined from ActivatingObject.

virtual bool Exit::canBlockFlame() const;

Exits can block flames. This function returns true;

**Pit**

virtual void Pit::doSomething();

Checks for overlap with other actors in its StudentWorld and activates if appropriate. This is a virtual function for clarity since it is redefined from Actor.

virtual void Pit::activateIfAppropriate(Actor\* a);

Makes a die by falling if a is alive. This is a virtual function for clarity since it is redefined from ActivatingObject.

**Flame**

virtual void Flame::doSomething();

Checks for overlap with other actors in its StudentWorld and activates if appropriate. The flame lasts for two ticks of the game and then dies. This is a virtual function for clarity since it is redefined from Actor.

virtual void Flame::activateIfAppropriate(Actor\* a);

Makes a die by burning if a is alive. This is a virtual function for clarity since it is redefined from ActivatingObject.

**Vomit**

virtual void Vomit::doSomething();

Checks for overlap with other actors in its StudentWorld and activates if appropriate. The vomit lasts for two ticks of the game and then dies. This is a virtual function for clarity since it is redefined from Actor.

virtual void Vomit::activateIfAppropriate(Actor\* a);

Makes a get vomited on if a is alive. This is a virtual function for clarity since it is redefined from ActivatingObject.

**Landmine**

virtual void Landmine::doSomething();

Landmine is planted and passes by 30 ticks of the game before activating. Checks for overlap with other actors in its StudentWorld and activates if appropriate. This is a virtual function for clarity since it is redefined from Actor.

virtual void Landmine::activateIfAppropriate(Actor\* a);

Explodes if a can trigger landmines. This is a virtual function for clarity since it is redefined from ActivatingObject.

virtual void Landmine::dieByFallOrBurnIfAppropriate();

Explodes if a flame hits it. This is a virtual function for clarity since it is redefined from Actor.

**Goodie** - abstract base class

virtual void Goodie::activateIfAppropriate(Actor\* a);

Gets picked up by a if appropriate. This is a virtual function for clarity since it is redefined from ActivatingObject.

virtual void Goodie::dieByFallOrBurnIfAppropriate();

Sets its alive state to false. This is a virtual function for clarity since it is redefined from Actor.

virtual void Goodie::pickUp(Penelope\* p) = 0;

Notifies p what benefits she gets by picking up this goodie. This is a pure virtual function because all Goodies must have some perk that p gains.

**VaccineGoodie**

virtual void VaccineGoodie::doSomething();

Checks for overlap with other actors in its StudentWorld and activates if appropriate. This is a virtual function for clarity since it is redefined from Actor.

virtual void VaccineGoodie::pickUp(Penelope\* p);

Calls p's increaseVaccine() function. This is a virtual function for clarity since it is redefined from Goodie.

**GasCanGoodie**

virtual void GasCanGoodie::doSomething();

Checks for overlap with other actors in its StudentWorld and activates if appropriate. This is a virtual function for clarity since it is redefined from Actor.

virtual void GasCanGoodie::pickUp(Penelope\* p);

Calls p's increaseFlameCharges() function. This is a virtual function for clarity since it is redefined from Goodie.

**LandmineGoodie**

virtual void LandmineGoodie::doSomething();

Checks for overlap with other actors in its StudentWorld and activates if appropriate. This is a virtual function for clarity since it is redefined from Actor.

virtual void LandmineGoodie::pickUp(Penelope\* p);

Calls p's increaseLandmines() function. This is a virtual function for clarity since it is redefined from Goodie.

**Agent** - abstract base class

virtual bool Agent::canBlockMovement() const;

Agents can block movement. This function returns true.

virtual bool Agent::triggersOnlyActiveLandmines() const;

Agents can trigger active landmines. This function returns true.

**Person** - abstract base class

virtual void Person::beVomitedOnIfAppropriate();

Increases infectionCount by 1. This is a virtual function for clarity since it is redefined from Actor.

virtual bool Person::triggersZombieVomit() const;

Person can trigger zombies to vomit on it. This function returns true.

void Person::clearInfection();

Sets infectionCount to 0. Not a virtual function since implementation does not change throughout different Person objects.

int Person::getInfectionCount() const;

Returns person's infectionCount. Not a virtual function since implementation does not change throughout different Person objects.

int Person::incrementInfectionCount();

Increments infectionCount by 1. Not a virtual function since implementation does not change throughout different Person objects.

**Penelope**

virtual void Penelope::doSomething();

// pseudocode

if penelope is not alive

return

if penelope is infected

increment infectionCount

if infectionCount reaches 500

penelope set to not alive

play player dead sound

return

get user key press

if pressed up

set direction up

compute destination coordinates and move there if not blocked

break

if pressed down

set direction down

compute destination coordinates and move there if not blocked

break

if pressed left

set direction left

compute destination coordinates and move there if not blocked

break

if pressed right

set direction right

compute destination coordinates and move there if not blocked

break

if pressed tab

if penelope has more than 0 landmines

introduce landmine in StudentWorld

decrement penelope number of mines

break

if pressed enter

if penelope has more than 0 vaccines

clear penelope's infection

decrement penelope number of vaccines

break

if pressed space

if penelope has more than 0 flame charges

repeatedly three times

compute coordinates for flames in direction penelope is facing

if flame blocked at destination coordinates

break

otherwise introduce new flame at coordinates in StudentWorld

break

if no key pressed

break

virtual void Penelope::useExitIfAppropriate();

Penelope uses the exit if there are no citizens remaining in the StudentWorld. This is a virtual function for clarity since it is redefined from Actor.

virtual void Penelope:: dieByFallOrBurnIfAppropriate();

Penelope dies and plays the sound corresponding with Penelope's death. This is a virtual function for clarity since it is redefined from Actor.

virtual void Penelope:: pickUpGoodieIfAppropriate(Goodie\* g);

Penelope calls g's pickUp() function, increases the score by 50, plays the sound corresponding with getting a goodie, and then sets g's alive state to false. This is a virtual function for clarity since it is redefined from Actor.

void Penelope::increaseVaccines();

Increments number of vaccines Penelope has by 1. Not a virtual function since implementation does not change throughout different Penelope objects.

void Penelope::increaseFlameCharges();

Increments number of flame charges Penelope has by 5. Not a virtual function since implementation does not change throughout different Penelope objects.

void Penelope::increaseLandmines();

Increments number of landmines Penelope has by 2. Not a virtual function since implementation does not change throughout different Penelope objects.

int Penelope::getNumVaccines() const;

Returns the number of vaccines Penelope has. Not a virtual function since implementation does not change throughout different Penelope objects.

int Penelope::getNumFlameCharges() const;

Returns the number of flame charges Penelope has. Not a virtual function since implementation does not change throughout different Penelope objects.

int Penelope::getNumLandmines() const;

Returns the number of landmines Penelope has. Not a virtual function since implementation does not change throughout different Penelope objects.

bool Penelope::atExit() const;

Returns if Penelope is at an exit. Not a virtual function since implementation does not change throughout different Penelope objects.

void Penelope::setExit(bool state);

Sets whether Penelope is at an exit or not to boolean parameter state. Not a virtual function since implementation does not change throughout different Penelope objects.

**Citizen**

virtual void Citizen::doSomething();

// pseudocode

if citizen not alive

return

if citizen infected

increment citizen's infectionCount

if citizen's infectionCount reaches 500

set alive to false

play zombie born sound

decrease score by 1000

record citizen off of StudentWorld

70% chance of generating dumb zombie and 30% chance of generating smart zombie

return

if even number ticks

increment ticks and return

decrement ticks

store distance, otherX, otherY, and threat boolean for nearest citizen trigger

if nearest trigger is not a threat and within 80 pixels

if y coordinates the same

set direction toward citizen trigger that gets citizen closest to trigger

move to location if possible

if x coordinates the same

set direction toward citizen trigger that gets citizen closest to trigger

move to location if possible

otherwise

randomize whether citizen more horizontally or vertically

set direction toward citizen trigger that gets citizen closest to trigger

move to location if possible

store distance from nearest citizen threat

if nearest citizen threat is within 80 pixels

test all four directions of movement, see which moves citizen farthest from threat

if possible, move to farthest distance possible away from threat

otherwise, don't move

virtual void Citizen::useExitIfAppropriate();

Citizen uses the exit. This increases the score by 1000 points, plays the sound corresponding with saving a citizen, records citizen is removed from StudentWorld, and removes the citizen from StudentWorld. This is a virtual function for clarity since it is redefined from Actor.

virtual void Citizen::dieByFallOrBurnIfAppropriate();

Citizen dies and plays the sound corresponding with citizen's death. This decreases the score by 1000 points, plays the sound corresponding with citizen death, and records citizen is removed from StudentWorld. This is a virtual function for clarity since it is redefined from Actor.

virtual void Citizen::beVomitedOnIfAppropriate();

If citizen has not been vomited on before, play the sound corresponding with infecting a citizen and increase the citizen's infectionCount by 1 by calling Person's beVomitedOnIfAppropriate() method. This is a virtual function for clarity since it is redefined from Actor.

**Zombie** - abstract base class

virtual bool Zombie::threatensCitizens() const;

Zombies threaten citizens. This function returns true.

void Zombie::vomitIfPossible();

Computes potential vomit coordinates to be SPRITE\_WIDTH or SPRITE\_HEIGHT distance away from zombie based on zombie's current direction. If there is an actor at the potential vomit coordinates that can get infected, there is a 1 in 3 chance that a zombie introduces a vomit object at the potential vomit coordinates and plays the corresponding vomit sound. Not a virtual function since implementation does not change throughout different Zombie objects.

void Zombie::zombieMove(int& movementPlan);

Checks if a zombie can move one pixel toward its current direction without being blocked. If so, move there and decrement its movementPlan by 1. If not, reset its movementPlan to zero. Not a virtual function since implementation does not change throughout different Zombie objects.

**DumbZombie**

virtual void DumbZombie::doSomething();

// pseudocode

if zombie not alive

return

if even number tick

increment tick and return

decrement tick

vomit on nearby actor if possible

if movementPlan reaches 0

set movementPlan to random value between 3 and 10 inclusive

set direction randomly

move as a zombie does

virtual void DumbZombie::dieByFallOrBurnIfAppropriate();

Sets dumb zombie's alive state to false, increase score by 1000 points, and plays sound corresponding with a zombie's death. Then has 1 in 10 chance to fling vaccine goodie onto StudentWorld at death. If it flings the goodie, randomize the direction and check if SPRITE\_WIDTH or SPRITE\_HEIGHT distance away from the zombie's position in that direction overlaps with any current actor in the StudentWorld. If it does, do nothing. Otherwise, introduce the vaccine goodie into StudentWorld. This is a virtual function for clarity since it is redefined from Actor.

**SmartZombie**

virtual void SmartZombie::doSomething();

// pseudocode

if zombie not alive

return

if even number tick

increment tick and return

decrement tick

vomit on nearby actor if possible

if movementPlan reaches 0

set movementPlan to random value between 3 and 10 inclusive

if there is a vomit trigger within vomiting distance

if y values are equal

set direction toward nearest vomit trigger as left or right

if x values are equal

set direction toward nearest vomit trigger as up or down

otherwise

set direction toward nearest vomit trigger randomly with

50% chance being left or right and 50% chance being up or down

otherwise

set direction randomly

move as a zombie does

virtual void SmartZombie::dieByFallOrBurnIfAppropriate();

Sets smart zombie's alive state to false, increase score by 2000 points, and plays the sound corresponding with a zombie's death. This is a virtual function for clarity since it is redefined from Actor.

**StudentWorld**

virtual int StudentWorld::init();

Initializes all actors in StudentWorld by loading a level text file containing the keys to locations and identification of each actor. Virtual because GameWorld requires its implementation. Returns constant that continues the game.

virtual int StudentWorld::move();

Allows all actors to do something each tick. Cleans dead actors after each tick movement ends and updates the printed stringstream information on the screen. Virtual because GameWorld requires its implementation. Returns a constant determining the game state.

virtual void StudentWorld::cleanUp();

Destroys all actors in the StudentWorld. Virtual because GameWorld requires its implementation.

int StudentWorld::getNumCitizens() const;

Returns number of citizens alive in StudentWorld.

void StudentWorld::addActor(Actor\* a);

Adds actor a to StudentWorld.

void StudentWorld:: recordCitizenGone();

Decrements a citizen from numCitizens.

void StudentWorld:: activateOnAppropriateActors(Actor\* a);

Checks if a overlaps with any actor in the StudentWorld and acts on overlapping actors according the a's activating implementation.

bool StudentWorld::isAgentMovementBlockedAt(double x, double y, Actor\* itself);

Checks if coordinate (x,y) intersects with any other actor that blocks movement in StudentWorld. It goes through each actor and checks all boundaries of the image box to test for intersecting. Returns true if there is blocking, false if not.

bool StudentWorld:: isFlameBlockedAt(double x, double y) const;

Checks if coordinate (x,y) overlaps with any other actor that blocks flames in StudentWorld. It goes through each actor and checks if the Euclidean distance between coordinates is less than or equal to 10 pixels away. Returns true if there is overlapping, false if not.

bool StudentWorld:: isZombieVomitTriggerAt(double x, double y) const;

Checks if there is an actor in all actors in StudentWorld that can be vomited on within the Euclidean distance under 10 pixels away from x,y. Returns true if there is an actor that can be vomited on within that distance from x,y, false if not.

bool StudentWorld:: locateNearestVomitTrigger(double x, double y, double& otherX, double& otherY, double& distance);

Locates the closest citizen or Penelope to the smart zombie at x,y and stores its coordinates in otherX, otherY by iterating through all citizens alive and Penelope in StudentWorld. Stores the Euclidean distance between coordinates in distance. Returns true if found, false if not.

bool StudentWorld:: locateNearestCitizenTrigger(double x, double y, double& otherX, double& otherY, double& distance, bool& isThreat) const;

Locates the closest zombie or Penelope to the citizen at x,y and stores its coordinates in otherX, otherY by iterating through all zombies alive and Penelope in StudentWorld. Stores the Euclidean distance between coordinates in distance and sets isThreat to true if the closest trigger is Penelope, false if the closest trigger is a zombie. Returns true if found, false if not.

bool StudentWorld:: locateNearestCitizenThreat(double x, double y, double& otherX, double& otherY, double& distance) const;

Locates the closest zombie to the citizen at x,y and stores its coordinates in otherX, otherY by iterating through all zombies alive in StudentWorld. Stores the Euclidean distance between coordinates in distance. Returns true if found, false if not.

bool StudentWorld:: isThrownGoodieBlockedAt(double x, double y) const;

Checks if coordinate (x,y) intersects with any other alive actor in StudentWorld. Returns true if there is an intersection.

Functionality:

Every feature in the ZombieDash specification is implemented in this project.

Design decisions:

1) It was not specified whether actors act on newly introduced actors in its current tick or not. I chose to have actors act on newly introduced actors on their current ticks. I also allow for newly introduced actors to act starting on the tick in which it was introduced.

2) It was not specified what would happen if a DumbZombie's dropped goodie would be able to get burned by the same flame that killed the DumbZombie. I chose to have the fire also burn the dropped vaccine goodie if it overlaps with it in its current tick in accordance with precedent established by design decision 1.

3) It was not specified what would happen if the landmine flames overlapped with objects that blocked flames. I followed the sample executable and chose to have landmine flames work in the same way as flamethrower flames. They get blocked by objects that can block flames.

Testing:

I did not test my abstract base classes because I could not make instances of them. However, the functionality of their derived classes proves that those classes function as expected.

**Wall**

The wall object was tested by first introducing it into StudentWorld and seeing if it constructed at the proper location with its proper image. Penelope then walked into walls to see if they could block her movement. Then, Penelope shot fire at the walls to see if flames were blocked. Next, mines were planted next to walls and the user waited for a citizen or zombie to activate the mine. It was observed that the walls blocked the flames from the mine. Lastly, zombies and citizens were introduced into the StudentWorld and the program was run for 5 minutes without user intervention, analyzing whether or not other agents ran into walls.

**Exit**

The exit object was tested by first introducing it into StudentWorld and seeing if it constructed at the proper location with its proper image. With no other persons in StudentWorld, Penelope walked onto the exit to see if the level would end. Then, citizens were introduced into StudentWorld. Penelope then walked over the exit with citizens remaining in the game to test that nothing should happen. Next, Penelope led citizens to the exits to test if citizens could go through exits. Lastly, Penelope went on the exit after all citizens exited from the game to test if the level would end.

Exit also blocks flames, which was tested by having Penelope shoot flames at the exit at various directions and distances. The flames were able to "skim" the edges of the objects blocking flame, as they should according to the specification, and were not able to pass through objects blocking flame.

**Pit**

The pit object was tested by first introducing it into StudentWorld and seeing if it constructed at the proper location with its proper image. First, Penelope walked into the pit to see if Penelope would die and restart the level with a decremented life. Then, other agents, both types of zombies and citizens, were introduced into StudentWorld and lured into walking into the pit using Penelope to see if they would die. This also tested if respective death by falling algorithms would play for each type of zombie and the citizen, noting the sounds played and the score change after each death by pit. Lastly, fire was shot at the pit by Penelope to ensure it did not block flames.

**Landmine**

The landmine object was tested by giving Penelope landmines to start the level and by planting them. This checked that the landmine was planted at the correct location with the correct image. Then, Penelope would walk over the landmine herself, prompting an explosion sound and Penelope death, ending the level and decrementing Penelope's lives. Then, Penelope planted landmines for other agents to walk onto. The mines were first planted in the open, where other agents would cause the mine to generate flames in the landmine position and in the eight adjacent positions. It also tested if a pit object generated at the location, and if the pit object worked by luring agents into it and walking into it with Penelope herself. Finally, the explosion was tested near locations that blocked flames, like walls. This tested if the flames were still blocked by actors that could block flames.

**Flame**

Flame was tested through Penelope using the flamethrower and through triggering landmines. First, Penelope shot flames at actors that could die by flame. It was noted that the actors successfully died by leaving the StudentWorld, playing sounds, and changing the score. This included goodies, agents, and landmines. Specifically, landmines were expected to explode once being shot at by flames in the same manner that activating landmines would explode. Lastly, flames were shot at actors that blocked flames, such as walls and exits. This tested if the flames were blocked by actors blocking flames.

**Vomit**

Vomit was tested first by manually walking to zombies with Penelope and then trapping citizens with smart zombies. First, Penelope would walk toward zombies to get vomited on. The vomit image was expected to appear with corresponding vomit sounds and rapid incrementation of infectionCount. Penelope then waited until her infectionCount reached 500 to test death by infection. Penelope was vomited on through pits and exits to test that these actors did not block vomit. Then, citizens were vomited on by smart zombie cornering them. It was expected for the corresponding citizen infected and vomit sounds would play. The user waited until the citizen turned into a zombie to see if the infection by vomit worked.

**VaccineGoodie**

The vaccine goodie was tested by generating it in a StudentWorld. First, Penelope would pick up the goodie by overlapping with it. This tested that the goodie would be removed and the vaccine count of Penelope would increment by one. The vaccine was then used after infection to test if the vaccine was properly gained by Penelope. Also, other agents were introduced to the StudentWorld to test that they would not pick up the goodie. Finally, Penelope shot flames at the goodie to see if it would get destroyed.

**GasCanGoodie**

The gas can goodie was tested by generating it in a StudentWorld. First, Penelope would pick up the goodie by overlapping with it. This tested that the goodie would be removed and the flame charge count of Penelope would increment by five. The flames were then used by firing the flamethrower to test if the flame charges were properly gained by Penelope. Also, other agents were introduced to the StudentWorld to test that they would not pick up the goodie. Finally, Penelope shot flames at the goodie to see if it would get destroyed.

**LandmineGoodie**

The landmine goodie was tested by generating it in a StudentWorld. First, Penelope would pick up the goodie by overlapping with it. This tested that the goodie would be removed and the landmine count of Penelope would increment by two. The mines were then planted to test if the landmines were properly gained by Penelope. Also, other agents were introduced to the StudentWorld to test that they would not pick up the goodie. Finally, Penelope shot flames at the goodie to see if it would get destroyed.

**Penelope**

Penelope would first be introduced into the StudentWorld to test if she was generated properly with a proper image. First, the movement controls were tested by hitting respective keys and testing if Penelope moved as expected. Penelope then collected all three types of goodies and tested if they all worked with respective key presses. Penelope then walked into landmines and got vomited on to test her death capability. Lastly, Penelope walked to the exit before and after all citizens on the current level exited. This tested if Penelope correctly handled exiting, specifically not exited with citizens still present and exiting successfully when citizens were all gone from the StudentWorld.

**Citizen**

Citizen was first introduced into the StudentWorld to test if it was generated properly with the proper image. Then, Penelope walked slowly within range of the citizen to test if the citizen followed Penelope. Penelope then led the citizen to the exit to test if the citizen exited correctly. Then, zombies were added to the StudentWorld. The citizen was left alone to test if it would run away intelligently from the zombies, attempting to move as far away as possible from zombies within a range of 80 pixels. Eventually, zombies cornered the citizen and infected it, which tested the citizen infection sounds. After a while, the citizen was seen to turn into a zombie. A respective sound was played with a score update, the removal of citizen, and introduction of a zombie. In 10 tests, Penelope moved toward the new zombie to see if roughly 30% of them became smart zombies, which would follow her around.

Citizens also had death capabilities that were tested. First, Penelope would shoot flames at the Citizen. This reflected a score change and citizen death sound along with the removal of the citizen from the StudentWorld. This was also tested with leading citizens into pits. Lastly, citizens were led to goodies to ensure they were not collected.

**DumbZombie**

Dumb zombies were tested by introducing them into StudentWorld and seeing if they generated correct images at the correct location. They were then allowed to move freely and randomly, making sure they did not pass through objects that block movement. Then, the zombies were killed by Penelope's flames and mines, ensuring correct death sounds, score updates, and zombie removal occurred. Penelope moved toward these zombies to ensure that they were not intelligently following her. Penelope then killed many, many dumb zombies until a vaccine was thrown at a dumb zombie's death to test its "death fling" feature. Lastly, Penelope moved in front of the zombie and tested for its vomiting capability.

**SmartZombie**

Smart Zombies were tested in the same way as Dumb Zombies were with the exception of the "death fling" of a vaccine occasionally. The additional testing was introducing Penelope and citizens within an 80px range and testing if the Smart Zombie would make movements toward Penelope or the citizens, gradually getting closer rather than walking randomly. Finally, the Smart Zombie was killed to test for a score increase of 2000 points as opposed to the Dumb Zombie's score update of 1000 points.

**StudentWorld**

StudentWorld was tested by first running the game and checking if level files properly loaded. As each level completed, it was noted that the score was retained with reset goodie and infection values and incremented level count. The biggest init() test was properly loading all members of the text file into the StudentWorld. The level format was changed to test for corresponding error response and the final level was set to be 99 to test for player winning. At each tick of the game, each actor should have the ability to do something and the game should run smoothly. Blocking functionality was tested by the previous actor tests. The trigger functions were also tested by running previous actor tests. Level handling was tested by dry-running the game and seeing if it matched the executable.