Harsh Chobisa

Professor Carey Nachenberg

CS 32

27 February 2019

Project 3 Report

*Part One—High Level Descriptions:*

Actor:

Actor is a general class that is used to derive every other object from. Actor is derived from Graph Object which was provided for us.

// Action to perform for each tick.

virtual void doSomething() = 0;

This function is pure virtual because every object I derive from Actor will have to doSomething once it is called in the move() function of Student World.

// Is this actor dead?

bool isDead() const;

This function is self explanatory. This is not virtual as this is functionality that can be handled by the base class Actor.

// Mark this actor as dead.

void setDead();

This function is self explanatory setter for Dead. This is not virtual as this is functionality that can be handled by the base class Actor.

// Get this actor's world

StudentWorld\* world() const;

This function returns the world pointer for every Actor. This is not virtual as this is functionality that can be handled by the base class Actor, and because every Actor has a world() that it is a part of.

// Does this object block agent movement?

virtual bool blocksMovement() const;

This function returns true if the Actor blocks the movement of other Actors. This is virtual to take advantage of polymorphism, as some Actors return true and others return false.

// Does this object block flames?

virtual bool blocksFlame() const;

This function returns true if the Actor blocks flames. This is virtual to take advantage of polymorphism, as some Actors return true and others return false.

// If this is an activated object, perform its effect on a (e.g., for an

// Exit have a use the exit).

virtual void activateIfAppropriate(Actor\* a);

This function is virtual because it must be called on every Actor. What this function does is that it is called in the move function of student world for every single Actor, and if the Actor overlaps with another Actor that it should activate on, it activates. This function takes advantage of the polymorphic structure I have in the code, as we can now loop over every actor and run this function.

// If this object uses exits, use the exit.

virtual void useExitIfAppropriate();

Another function that takes advantage of Polymorphic structure of our code. If the Actor can use an exit, it will use the exit. This function goes hand in hand with activateIfAppropriate.

// If this object can die by falling into a pit or burning, die.

virtual void dieByFallOrBurnIfAppropriate();

Another function that takes advantage of Polymorphic structure of our code. If the Actor is in a position where it should die, it is set to dead and other appropriate actions are taken. This function goes hand in hand with activateIfAppropriate. This function is only called on Actors that can be set to dead.

// If this object can be infected by vomit, get infected.

virtual void beVomitedOnIfAppropriate();

Another function that takes advantage of Polymorphic structure of our code. If the Actor is in a position where it should be vomited on, it is set to infected and other appropriate actions are taken. This function goes hand in hand with activateIfAppropriate. This function is only called on Actors that can be vomited on.

// If this object can pick up goodies, pick up g

virtual void pickUpGoodieIfAppropriate(Goodie\* g);

Another function that takes advantage of Polymorphic structure of our code. If the Actor is in a position where it should pick up a goodie, the goodie is picked up. This function goes hand in hand with activateIfAppropriate. This function is only called on Penelope, as it is the only Actor that can pick up Goodies.

// Does this object trigger landmines only when they're active?

virtual bool triggersOnlyActiveLandmines() const;

This function returns true if the Actor blocks triggers landmines. Only Citizens and Penelope and Zombies trigger landmines. This is virtual to take advantage of polymorphism, as some Actors return true and others return false.

// Can this object cause a zombie to vomit?

virtual bool triggersZombieVomit() const;

This function returns true if the Actor blocks causes Zombies to vomit. Only Citizens and Penelope trigger Vomit. This is virtual to take advantage of polymorphism, as some Actors return true and others return false.

// Is this object a threat to citizens?

virtual bool threatensCitizens() const;

This function returns true if the Actor threatens Citizens. Only Zombies threaten citizens. This is virtual to take advantage of polymorphism, as some Actors return true and others return false.

// Does this object trigger citizens to follow it or flee it?

virtual bool triggersCitizens() const;

This function returns true if the Actor triggers Citizens to follow it or flee. Only Zombies and Penelope trigger citizens. This is virtual to take advantage of polymorphism, as some Actors return true and others return false.

Wall:

Wall is a class that stays in one place and blocks movement. Definitely the easiest class to write.

virtual void doSomething();

This function is virtual as it overwrites the base class and we need to be able to call it with an Actor pointer. Walls don’t do anything, so the doSomething does absolutely nothing.

virtual bool blocksMovement() const;

This function is virtual as it overwrites the base class and we need to be able to call it with an Actor pointer. Returns true, as walls block movement.

virtual bool blocksFlame() const;

This function is virtual as it overwrites the base class and we need to be able to call it with an Actor pointer. Returns true, as walls block flames.

Exit:

Exit is a class derived from activating object. It is similar to wall as it does not move, but is different as it needs to check for object overlap and may be used to end the level.

virtual void doSomething();

This function is virtual as it overwrites the base class and we need to be able to call it with an Actor pointer. Actor asks the world if anything is touching it by calling activate on appropriate actors by passing itself in.

virtual void activateIfAppropriate(Actor\* a);

This function tells the actor a to use the exit if appropriate.

virtual bool blocksFlame() const;

Pit:

If an actor walks over a pit, it should die. It has similar functionality to Exit, thus it is an Activating Object, as Exit is also derived from Activating Object.

virtual void doSomething();

This function is virtual as it overwrites the base class and we need to be able to call it with an Actor pointer. Actor asks the world if anything is touching it by calling activate on appropriate actors by passing itself in.

virtual void activateIfAppropriate(Actor\* a);

This function tells the actor a to die by fall or burn if appropriate.

Flame:

Is also an activating object. It is found when shot out of the flamethrower or after a landmine explodes. Destroys objects that touch it. The flame only briefly stays on the screen.

virtual void doSomething();

This function is virtual as it overwrites the base class and we need to be able to call it with an Actor pointer. Actor asks the world if anything is touching it by calling activate on appropriate actors by passing itself in.

virtual void activateIfAppropriate(Actor\* a);

This function tells the actor a to die by fall or burn if appropriate.

Vomit:

Similar to Flame, so is also in the Activating Object class. Infects any object that touches it and is created by a Zombie. Is set to Dead after two ticks.

virtual void doSomething();

This function is virtual as it overwrites the base class and we need to be able to call it with an Actor pointer. Actor asks the world if anything is touching it by calling activate on appropriate actors by passing itself in. It also counts the number of ticks it has been alive, and if this count reaches 2, then the vomit is set to dead.

virtual void activateIfAppropriate(Actor\* a);

This function tells the actor a to get infected if appropriate.

Landmine:

Landmine is also an Activating objects that explodes if anything that activates it touches it. The landmine becomes active after 30 ticks, after which anything that touches it causes it to explode.

virtual void doSomething();

This function is virtual as it overwrites the base class and we need to be able to call it with an Actor pointer. Actor asks the world if anything is touching it by calling activate on appropriate actors by passing itself in. It also counts the number of ticks it has been alive, and if this count reaches 30, then the landmine is set to active.

virtual void activateIfAppropriate(Actor\* a);

If the actor a triggers active landmines, call die by fall or burn.

virtual void dieByFallOrBurnIfAppropriate();

Explodes the landmine if it hits by a flame.

Goodie:

Goodie is an activating object that handles a lot of common functionality for all of the Goodie classes.

virtual void activateIfAppropriate(Actor\* a);

Tells the actor a to pick up the goodie if it can.

virtual void dieByFallOrBurnIfAppropriate();

Sets the goodie to dead if hit by a flame.

virtual void grantSpecificReward(Penelope\* p) = 0;

Grants a specific reward to Penelope. This is a pure virtual because it is implemented in the sub class differently for every Goodie.

virtual void pickUp(Penelope\* p);

If the goodie is picked up, tell the world to increase the points as well as play a sound.

Vaccine Goodie, Landmine Goodie, Gas Can Goodie. These functions are implemented similarly for all of these classes. All of these classes are Goodies and have the same implementation, except each different Goodie grants Penelope a different reward. That is why that function is implemented separately here.

virtual void doSomething();

Tell the world to activate on this object.

virtual void grantSpecificReward(Penelope\* p);

Increase the goodie count of Penelope, depending on what vaccine is picked up.

Agent:

Agent is a base class for Penelope, Human, and Zombie. It encapsulates the common functionality between those three classes, specifically for the two functions that follow.

virtual bool blocksMovement() const;

Returns true as agents block movement.

virtual bool triggersOnlyActiveLandmines() const;

Returns true as agents trigger active landmines.

Human:

Human is an Agent that encapsulates the functionality of Penelope and Citizens. The human class handles everything related to Penelope and Citizen getting infected.

virtual void beVomitedOnIfAppropriate();

Infects the human and plays a sound that the human has been infected.

virtual bool triggersZombieVomit() const;

Returns true as humans trigger zombie vomit.

int getInfectionDuration() const;

Returns how many ticks the Human has been infected for.

Penelope:

Penelope is a human because it can get infected. Penelope is the actor the user controls, therefore Penelope’s do something method takes in user input from the keyboard. Penelope also has numerous getters and setters so other classes can interact with it easily.

virtual void doSomething();

Get the key press from the user, and based on the key press, move the actor or deploy a goodie or a flamethrower.

virtual void useExitIfAppropriate();

If there are no more citizens left, set Actor to dead and tell student world to end the game.

virtual void dieByFallOrBurnIfAppropriate();

Sets Penelope to dead and tells student world to end the game.

virtual void pickUpGoodieIfAppropriate(Goodie\* g);

Increases Penelope’s Goodie count by the specific goodie.

virtual void dieByInfection();

Sets the actor to dead by calling dieByFallOrBurnIfAppropriate.

virtual bool threatensCitizens() const;

Returns true as Penelope does not threaten citizens.

virtual bool triggersCitizens() const;

Returns true as Penelope triggers citizens.

void increaseVaccines();

Increase the number of vaccines the object has.

void increaseFlameCharges();

Increase the number of flame charges the object has.

void increaseLandmines();

Increase the number of landmines the object has.

int getNumVaccines() const;

Returns the number of vaccines the object has.

int getNumFlameCharges() const;

Returns the number of flame charges the object has.

int getNumLandmines() const;

Returns the number of landmines charges the object has.

Citizen:

Citizen is a human that runs away from Zombies and runs towards Penelope. It is a human because it can be infected and has similar functionality to Penelope.

virtual void doSomething();

If the citizen is not alive or paralyzed, return. Else, update infection status and move either toward Penelope or away from zombies.

virtual void useExitIfAppropriate();

If citizen overlaps with an exit, set the citizen to dead and don’t decrement the points.

virtual void dieByFallOrBurnIfAppropriate();

Set the citizen to dead, play a sound, and decrement the appropriate amount of points.

bool isParalyzed();

Returns true is the citizen is paralyzed, false if it isn’t.

void switchParalyzed();

Switches the Paralysis state of the Citizen.

virtual void dieByInfection();

Plays a sound and the calls dieByFallOrBurnIfAppropriate.

Zombie:

Zombie is an Agent that encapsulates all the commonalities between Dumb and Smart Zombie. It handles the movement of the Zombies and the vomiting of the zombies. The only thing the specific Zombie classes determine is what direction the Zombie moves in.

virtual bool threatensCitizens() const;

Returns true as Zombies threaten citizens.

virtual bool triggersCitizens() const;

Returns true as Zombies triggers citizens.

virtual void doSomething();

If the zombie is not paralyzed, move the zombie using either the SmartZombie functionality or the DumbZombie functionality depending on what type of Zombie it is. This function is virtual because it overwrites the doSomething method from actor.

Dumb Zombie:

Dumb Zombie is a zombie that moves randomly and occasionally drops a vaccine goodie upon death.

virtual void dieByFallOrBurnIfAppropriate();

Set the zombie to dead and increase the amount of points in student world. Drops a vaccine goodie ten percent of the time.

Smart Zombie:

Smart Zombie is a zombie that moves towards Penelope or towards citizens. Determine movement plan is used to decide the direction of movement, while the base class handles the actual moving.

virtual void dieByFallOrBurnIfAppropriate();

Set the zombie to dead and increase the amount of points in student world.

virtual void determineMovementPlan();

Determine which way the Zombie should move to go towards Penelope or a Citizen of they are in range. Else, use the same movement plan as a dumb zombie and randomly pick direction.

StudentWorld:

virtual int init()

Initializes the level and places all the actors where they need to be based on the level text file specification.

virtual int move()

This function uses GameWorld’s virtual move(). The function runs the doSomething() for all actors. If Penelope dies, we end the game. If the level is finished, we end the game.

virtual void cleanUp()

This function essentially creates the destructor by freeing up all of the dynamically allocated memory by deleting all the actors and Penelope.

void addActor(Actor\* a)

Adds dynamically created actors to the vector of actors I have.

void recordCitizenGone()

Decrements m\_numCitizens every time a citizen dies.

void recordLevelFinishedIfAllCitizensGone()

If all the citizens are gone and Penelope exits, record that the level is finished and set m\_finished to true.

void activateOnAppropriateActors(Actor\* a)

For each actor that overlaps with A, take the appropriate action on A, such as burning it or picking up a goodie etc.

bool isAgentMovementBlockedAt(double x, double y, Actor\* a) const

Returns true if any actors that block movement’s bounding box intersect with the point x and y and that the blocking actor is not a.

bool overlap(double x, double y) const

Returns true if the Euclidian distance of an actor and these points is less than ten pixels.

bool isFlameBlockedAt(double x, double y) const

Returns true if flames are blocked at the point x,y.

bool isZombieVomitTriggerAt(double x, double y) const

Returns true is vomit is triggered at the point x,y.

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

Finds the nearest Citizen or Penelope from the point x , y and sets distance to the distance of the actor found. Set otherX and otherY to the location of that object.

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

Finds the nearest Penelope or Zombie to x, y and sets distance to the distance of the actor found. Set otherX and otherY to the location of that object. If we find a Penelope, set isThreat to true, else set it to false.

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

Finds the nearest to Zombie to x, y and sets distance to the distance of the actor found. Set otherX and otherY to the location of that object

bool noMoreCitizens()

returns true if m\_numCitizens is equal to zero.

*Part Two—Unfinished Functionality and Bugs:*

I have implemented all required functionality and am not aware of any bugs in my program.

*Part Three—Design Decisions and Assumptions:*

I did not have to make many assumptions during the project. I had a few questions about whether or not I could hold the pointer with Penelope in an Actor pointer or a Penelope pointer. I felt like this would significantly change how I used the program. I looked over the spec and I really could not find anything, so I emailed Carey on what to do, and he told me either was fine. Now, having completed the program, I realized that doing what I did, which was putting Penelope in a Penelope pointer, was likely the best way to do it.

I was also unsure whether or not I was supposed to play SOUND\_CITIZEN\_INFECTED when Penelope is infected. It worked a lot better for my inheritance structure if it was played, and the spec was a little unclear. Once again, an email from Carey clarified the situation, and the email said that the sound is not played. Because of this I had to make some changes to my code, but there were no major changes.

Another assumption I made was that the game would take care of what would happen once the user finished all the levels. I did not implement the functionality to take care of what would happen once the user ran out of levels, as it was not specified in the spec anywhere.

The last assumption I made was that the text files for the levels were formatted correctly and stayed within the bounds of 256 pixels by 256 pixels. This assumption was important when placing objects onto the screen.

*Part Four—Testing Descriptions:*

Goodie:

I tested Goodie by running the program and checking that all of the required functionality was present. I would walk over Goodie with Penelope and make sure that the Goodie disappeared. I would also print the number of Goodies that Penelope had in order to make sure that that Penelope’s Goodie count was increasing properly.

I did all of these tests for every single Goodie subclass. I could test all of the Goodies in the same way because they are all essentially the exact same thing besides the different benefits they give to Penelope. Thus, I was able to consolidate my testing into one. Of course, testing was finished by making sure that the program could build on G++.

Flames:

I tested flames by building Penelope’s flamethrower and then attempting to complete a variety of options and making sure that they executed correctly. I first went up to Goodies and used the flamethrower and made sure that the Goodie disappeared. I then went to walls and exits and made sure that the flames did not go through the walls and exits. I also went up to zombies and citizens and made sure that when hit by flames, they would go away. Lastly, I went up to Landmines and made sure that they were activated whenever they were hit by flames.

To sum it up, the testing for flames was essentially going into the game and making sure the flames did what I expected them too. It is important to note that I checked whether or not the flames went away after a second. Of course, testing was finished by making sure that the program could build on G++.

Wall:

I tested wall by essentially making sure that it followed all the requirements given in the spec. What I did was first I made sure that Penelope could not go through the wall. Then I made sure that any zombies attempting to move were stopped by the wall. I also checked that any flames that were shot at a wall would be stopped. I also placed a landmine near a wall to make sure that only the flames that were not stopped by the wall were created. Essentially, testing the wall was a lot of running things into the wall that shouldn’t go through the wall, and then making sure that the wall stopped their movement.

Exit:

I tested exit by making sure that the implementation I had created for it met the implementation that was outlined in the spec. I would check that I could run citizens and Penelope through the exit and would also check that flames and goodies would behave appropriately around exits. I was careful to make sure that an Penelope would only be able to use to exit if there were no more citizens left in the game. I also made sure that the number of citizens variable in Student World decreased by one every time a citizen was saved. I feel that Exit was one of the simpler parts of the program, so that was the extent of my testing. Of course, I still attempted to build on g++, but that was about the only other thing I did with exit. I also made sure that every time I completed the level and went to the exit I actually moved on to the next level.

Pit:

Pit was another one of the easiest actors to test. All I had to do was go through the entire program and wait for Zombies to move onto the pit, and then make sure that they died if they did move on to the Pit. I also ran Penelope into the pit a few times just for good measure. I also ran citizens into the Pit just for double good measure, and my Pit had the required implementation every time. I also made sure I could build on g++.

Vomit:

Vomit was a tough one to test because vomit relies on other classes heavily in order to actually work. One of the largest problems I had with my vomit was that the zombies kept vomiting randomly and I had absolutely no idea why. After an hour of searching for the bug, I realized that in my function to loop through and check for vomit triggers, I had the same variable name for a parameter and as a counter for my variable. After that, I essentially just ran the program, and made sure that whenever Penelope or a Citizen got close to a Zombie, it would create vomit at the location.

I also had an issue where the Zombies would vomit and then freeze, but I realized, after reading the spec again, that the issue came down to me not using the random integer function to determine the one in three chance of it actually vomiting.

Landmine:

Ah the Landmine. The landmine was one of the most frustrating things for me to build because I kept running into small issues all over the place. The first issue I had was that any time the Landmine would go off, the entire game would freeze, I would have a bad access, and my game would end. I had no idea how to fix this, so I went in to office hours and Trevor helped me work through the problem. Turns out I did not have a checker on whether or not something was dead before looping through it in one of my functions which was causing the issue. Next, only one flame would show up when my landmine blew up. I realized that I had a bounding box issue, and upon fixing that the flames showed up properly.

Essentially, to test it, I set up Landmines and made sure that they were properly set off whenever a citizen zombie or Penelope would walk on them. I also made sure that the pit showed up in the right spot, and also that the flames from the Landmine were blocked by walls. I also made sure that the landmine exploding would destroy any Goodies in the area.

Penelope:

In my opinion, Penelope was one of the easiest ones to test. Penelope essentially came down to making sure that when it got hit by vomit, its infection count went up. If any Goodie’s were used, I made sure to check that the Goodie count went down. The header text at the top of the screen was very useful in this situation, as I could just use that to track the Goodie amounts rather than sending the goodie amounts to the console.

I also had to make sure that Penelope could not go through walls and triggered the appropriate actors. For example, I had to make sure that a landmine would blow up if it was stepped on by Penelope. I had to make sure that Penelope was vomited on, and also had to make sure that its flamethrower worked as specified. Flamethrower testing is described in more detail in the flames section. The most important thing to test was that when Penelope died, we decreased the number of lives she had and also that we restarted the level. I tested this by killing Penelope by walking over a pit and making sure her lives decreased.

Dumb Zombie:

Dumb zombie does not have very extensive functionality, so testing it was quite simple. I just loaded a level with dumb zombies and checked their movement. At first, the zombies were going too fast, and I realized that I did not have paralysis implemented. After I implemented that, the zombies worked perfectly. I made sure that they vomited when near a citizen or Penelope.

The hardest to test about dumb zombie was the fact that it could randomly drop vaccine goodies when it died. However, the test was extremely hard to do, as the probability of a vaccine goodie being dropped was extremely low. So, to test it, I raised the probability of a vaccine goodie being dropped to 100%. After identifying that my dumb zombie would drop the vaccine goodie properly, I lowered the probability back to ten percent.

StudentWorld:

Student world was tested as a whole by printing out information to the console to make sure things lined up. Student world functions were tested by making sure anything that happened on the screen was exactly what happened in the console. For example, every time a zombie died, I printed out the number of zombies left to the console. Student World was just tested by making sure all of the interactions between objects were handled correctly. I played the game about fifteen times once I was done to make sure that everything was working consistently. I would also use the debugger extensively to make sure that objects were being removed properly. To test the movement of Zombies and Citizens I printed out distances between objects to make sure everything was working properly. Overall, StudentWorld was tested by running and playing the game.

Smart Zombie:

The smart zombie was tested the exact same way as the dumb zombie (except for the dropping of vaccines), with the difference being that I had to make sure that the smart zombie actually followed me around or tried to follow a citizen. In order to make the testing easier, I made the minimum distance to follow very large so the zombie would follow me no matter where on the screen I was. This aided me a lot in being able to track the location of zombie across the screen. Of course, I also had to test whether or not the Zombie vomited when it got near me, but to do that, I used the same techniques that I used to test Dumb Zombie.

Citizen:

Testing citizen was easily the hardest testing I had to undertake. To test the citizen, I put it in the game and checked its behavior in different situations. I put citizen in a corner and checked if it would pick the correct direction to move away from the zombie. Also I put a lot of citizens next to each other and made sure that they had the correct functionality in that case. Initially I had a lot of problems with citizens spazzing and doing random unexpected movements. However, after looking at my code more carefully, I found errors in the direction that I set my citizen to move in, and upon fixing those errors, the citizen moved as expected. The hardest thing to test was whether or not the citizen was able to correctly pick whether or not it should go towards Penelope or away from the Zombie or both. To test this, I would put the zombie near the citizen, and then run towards the citizen with Penelope to ensure the correct implementation.

Of course, I finished my testing by ensuring that the code would build on g++.