*1. A high-level description of each of your public member functions in each of your classes, and why you chose to define each member function in its host class; also explain why (or why not) you decided to make each function virtual or pure virtual. For example, “I chose to define a pure virtual version of the sneeze() function in my base Actor class because all actors in Zombie Dash are able to sneeze, and each type of actor sneezes in a different way.”*

Actor

virtual void doSomething() = 0 – I chose to make a pure virtual function called doSomething() which all actors will have. DoSomething() for a particular actor will run one tick. This function is pure virtual because each type of Actor will have their own version of do something because each type will do something different. Also, we would never need to create an Actor object, so making Actor an abstract data type is okay.

bool isDead() const – this function returns the value of the private member variable m\_state of Actor, essentially serving as a getter. This function is const because it does not change the value of m\_state, and does not need to be virtual because it is the exact same for all of Actor’s children.

void setDead() – this function sets the value of Actor’s private member variable m\_state to false, indicating that the object is dead. This function does not need to be virtual since it behaves the same way for all types of Actors.

StudentWorld\* world() const – this functions returns a pointer to the StudentWorld. We need this function in order to perform any actions on the student world since we need to go into Actor to access that pointer.

virtual void activateIfAppropriate(Actor\* a) – this function has the actor do a particular function. I have made this virtual because a Goodie’s implementation would be different from a pit. This function is important for every Actor to have since we can loop over the actors container and perform activate on all actors.

virtual void useExitIfAppropriate() – this function allows objects to implement the exit functionality. It is virtual because some objects will do something if they are overlapping with the exit, but others will do nothing.

virtual void dieByFallOrBurnIfAppropriate() – this function is virtual and allows each object to implement their own version to handle what to do when they fall into a pit or hit flame.

virtual void beVomitedOnIfAppropriate() – this function is virtual and allows each object to implement their own version to handle what to do when they are vomited on.

virtual void pickUpGoodieIfAppropriate(Goodie\* g) – this function is primarily for Penelope, as it is the only actor that can pick up goodies, but we make it virtual so that each actor can implement an empty version so that we can call it on all actors.

virtual bool blocksMovement() const – this function indicates the property of a particular actor to block another actor’s movement, which some functions will implement differently.

virtual bool blocksFlame() const - this function indicates the property of a particular actor to block a flame, which some functions will implement differently.

virtual bool triggersOnlyActiveLandmines() const – this function indicates the property of a particular actor to set off a landmine, which some functions will implement differently.

virtual bool triggersZombieVomit() const – this function indicates the property of a particular actor to be vomited on, which some functions will implement differently.

virtual bool threatensCitizens() const – this function indicates the property of a particular actor to be a threat to citizens, which some functions will implement differently.

virtual bool triggersCitizens() const – this function indicates the property of a particular actor to trigger a citizen, which some functions will implement differently.

Wall

virtual void doSomething() – Wall’s do something is needed since it’s defined as pure virtual in actor, and does nothing.

virtual bool blocksMovement() const – indicates that the wall blocks movement

virtual bool blocksFlame() const – indicates that the wall blocks flames

Exit

virtual void doSomething() – calls the activate if appropriate on overlapping actors

virtual void activateIfAppropriate(Actor\* a) – actors use the exit

virtual bool blocksFlame() const – indicates that the exit blocks flames

Pit

virtual void doSomething() – calls the activate if appropriate on overlapping actors

virtual void activateIfAppropriate(Actor\* a) – actors fall into the pit

Flame

virtual void doSomething() – calls the activate if appropriate on overlapping actors

virtual void activateIfAppropriate(Actor\* a) – actors get hit by the flame

Vomit

virtual void doSomething() – calls the activate if appropriate on overlapping actors

virtual void activateIfAppropriate(Actor\* a) – actors get hit by the vomit

Landmine

virtual void doSomething() – calls the activate if appropriate on overlapping actors

virtual void activateIfAppropriate(Actor\* a) – calls dieByFallOrBurnIfAppropriate() if the actor can activate a landmine

virtual void dieByFallOrBurnIfAppropriate() – removes the landmine from the game, adds a pit, and flames if possible

Goodie

virtual void activateIfAppropriate(Actor\* a) – calls the activate on appropriate actors

virtual void dieByFallOrBurnIfAppropriate() – removes the goodie from the game if it has to be removed

virtual void pickUp(Penelope\* p) = 0 – this function updates Penelope based on the type of goodie picked up. The function is pure virtual because each goodie will update Penelope differently, and there is no reason for a Goodie object

VaccineGoodie

virtual void doSomething() – asks the world if anything is touching it by calling activate on appropriate actors, passing itself in

virtual void pickUp(Penelope\* p) – increases Penelope’s vaccine count and kills the goodie

GasCanGoodie

virtual void doSomething() – asks the world if anything is touching it by calling activate on appropriate actors, passing itself in

virtual void pickUp(Penelope\* p) – increases Penelope’s flame count and kills the goodie

LandmineGoodie

virtual void doSomething() – asks the world if anything is touching it by calling activate on appropriate actors, passing itself in

virtual void pickUp(Penelope\* p) – increases Penelope’s landmine count and kills the goodie

Agent

virtual bool blocksMovement() const – overrides the actor’s blocksMovement() function to indicate that agents do block movement

virtual bool triggersOnlyActiveLandmines() const – overrides the actor’s blocksMovement() function to indicate that agents do trigger active landmines

Human

virtual void beVomitedOnIfAppropriate() – sets the human to be infected if vomited on

virtual bool triggersZombieVomit() const – indicates that the Human class will trigger zombie vomit, overriding that of actor

int getInfectionDuration() const – getter for the Human’s infection level

Penelope

virtual void doSomething() – This function handles a tick of Penelope. The first check is if Penelope is alive. Next we will handle the infection status. Now we read in the key from the user, and either move Penelope accordingly or use the flame, vaccine, or landmine.

virtual void useExitIfAppropriate() – This function overrides Actor’s useExitIfAppropriate() to let Penelope exit if the level is done.

virtual void dieByFallOrBurnIfAppropriate() – This function overrides Actor’s dieByFallOrBurnIfAppropriate() to set Penelope to dead.

virtual void pickUpGoodieIfAppropriate(Goodie\* g) – call the goodie’s pickup function, passing a pointer as a parameter

void increaseVaccines() – increases the vaccine count variable by 1

void increaseFlameCharges() – increases the flame count variable by 5

void increaseLandmines() – increases the flame count variable by 2

int getNumVaccines() const – getter to give the num vaccines

int getNumFlameCharges() const – getter to give the num flames

int getNumLandmines() const – getter to give the num landmines

Citizen

virtual void doSomething() – Run a tick of the citizen object. If the citizen is not alive or paralyzed, return immediately. Otherwise, update infection status and move appropriately either toward Penelope or away from zombies.

virtual void useExitIfAppropriate() – This is Citizen’s implementation of the useExitIfAppropriate() function that all actors share. Set the citizen to dead since it can be removed from the game and award points.

virtual void dieByFallOrBurnIfAppropriate() - This is Citizen’s implementation of the dieByFallOrBurnIfAppropriate () function that all actors share. Set the citizen to dead since it can be removed from the game and subtract points as a penalty.

Zombie

void vomitIfAppropriate() – attempts to vomit 1/3 of the time in the direction that the zombie is facing

bool threatensCitizens() const – indicates that zombie are a threat to citizens, overriding that of Actor’s

virtual void doSomething() - checks to see if the zombie is still alive and not fully infected yet, then makes the zombie try to vomit, and then either follow the current movement plan or make a new one. This function also calls a helper function to flip the paralysis state of the zombie. I chose to make Zombie handle doSomething because it is the same for dumb and smart, aside from movement plan calculation. I have made this function modular so that now I do not need to change anything in DumbZombie and SmartZombie.

DumbZombie

virtual void dieByFallOrBurnIfAppropriate() – This function implements overrides Actor’s dieByFallOrBurnIfAppropriate() to kill the zombie and fling out a vaccine goodie 1/10 of the time.

SmartZombie

virtual void dieByFallOrBurnIfAppropriate() – This function implements overrides Actor’s dieByFallOrBurnIfAppropriate() to kill the zombie

StudentWorld

virtual int init() – This function implements GameWorld’s virtual init() and initializes all of the actors onto the display based on the level file it is reading in.

virtual int move() – This function implements GameWorld’s virtual move(). The function runs the doSomething() for Penelope and all of the actors in the vector. If Penelope died on this tick, then we return that value.

virtual void cleanUp() – This function implements GameWorld’s virtual cleanup(). The function loops over the actors container and Penelope and deletes each one.

void addActor(Actor\* a) – This function adds the actor to the actors vector. This function is not virtual because it only exists in StudentWorld.

void recordCitizenGone() – This function subtracts one from the citizen count. This function is not virtual because it only exists in StudentWorld.

void recordLevelFinishedIfAllCitizensGone() – This function sets the m\_finished Boolean to true, indicating that the level is over. This function is not virtual because it only exists in StudentWorld.

void activateOnAppropriateActors(Actor\* a) – This function is not virtual because it only exists in StudentWorld. For each actor overlapping a, activate a if appropriate.

bool isAgentMovementBlockedAt(double x, double y) const – This function is not virtual because it only exits in StudentWorld. It checks to see if the coordinates (x,y) intersect with any bounding box in the actors container.

bool isAgentMovementBlockedAt(double x, double y, Actor\* a) const – This function is not virtual because it only exits in StudentWorld. It checks to see if the coordinates (x,y) intersect with any bounding box in the actors container, except for the actor a passed in. The purpose of this is to avoid cases where an actor would block itself.

bool isOverlap(double x, double y) const – This function checks to see if the coordinates (x,y) intersect the Euclidian distance of Penelope or anything in actors. This function is not virtual because it is only needed in StudentWorld.

bool isFlameBlockedAt(double x, double y) const – This functions loops through all actors, and sees if any actor that blocks flames is overlapping with the coordinate (x,y). Not virtual because only exists in StudentWorld.

bool isZombieVomitTriggerAt(double x, double y) const – This function finds if something at the indicated location that might cause a zombie to vomit (i.e., a human) by checking Euclidian dist. Not virtual because only exists in StudentWorld.

bool locateNearestVomitTrigger(double x, double y, double& otherX, double& otherY, double& distance) – This function tries to find the nearest thing that triggers zombie vomit using the Euclidian distance function. Not virtual because only exists in StudentWorld.

bool locateNearestCitizenTrigger(double x, double y, double& otherX, double& otherY, double& distance, bool& isThreat) const – This function tries to find the nearest thing that triggers citizens using the Euclidian distance function. Not virtual because only exists in StudentWorld.

bool locateNearestCitizenThreat(double x, double y, double& otherX, double& otherY, double& distance) const – This functions finds the nearest threat (zombie) to a particular citizen and stores the data for that threat in otherX, otherY, and distance by using the Euclidian dist function. Not virtual because only exists in StudentWorld.

bool isLevelDone() – This checks if m\_numCitizensRemaining is equal to 0, meaning there are no zombies left, so the level is done. Not virtual because only exists in StudentWorld.

*2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. “I didn’t implement the Flame class.” or “My smart zombie doesn’t work correctly yet so I treat it like a dumb zombie right now.”*

I did not find any errors in my program.

*3. A list of other design decisions and assumptions you made; e.g., “It was not specified what to do in situation X, so this is what I decided to do.”*

One of the things that I assumed was that the game would take care of ending itself when the player has finished all of the levels. The spec does not mention anything about sending an end game signal once the last level (level 6) is completed.

I also assumed that the level file was properly formatted in a 16x16 grid, and that the size of the game is 256x256.

I also noticed that on my laptop (windows), my Penelope was moving much faster than my friends who were working on a mac. However, my Penelope runs the same speed in my files as well as the executable provided, so I assumed this is something that I could not control. On the topic of system, in the executable the line at the top with the score, lives, level, etc, was partially cut off. My code matches this functionality. I assumed this was a weird issue with my computer and the TA Trevor agreed.

*4. A description of how you tested each of your classes (1-2 paragraphs per class).*

Goodie

For this class, the first thing I did was make sure that its functionality matched what was given in the spec, and that it behaved the same on the linux server. To test Goodie, I ran the program to make sure everything was working. The first test was that Penelope could walk over a Goodie and collect that Goodie. This part tested the collision detection between Penelope and Goodie. I also tested that goodies die when they are hit by a flame by shooting flames at the vaccine goodie, landmine goodie, and gas can goodie. On top of the visual testing, I also put print statements in my code to make sure that the display matches up.

Flames

For this class, the first thing I did was make sure that its functionality matched what was given in the spec, and that it behaved the same on the linux server. I tested flames in two ways – with the landmine explosion and the flamethrower. I shot flames at the goodies and make sure that they disappeared to check that things were actually being killed with flames. I also shot flames at zombies and citizens to make sure that they died. I made sure that flames also did not go through walls and exits by attempting to do so. I also made sure that if the first flame of a flamethrower was blocked that the subsequent flames were not created. For landmines, I placed landmines adjacent to walls and made sure the flames did not go into the walls. I also tried putting landmines in open space to make sure all radial flames were created.

Wall

For this class, the first thing I did was make sure that its functionality matched what was given in the spec, and that it behaved the same on the linux server. To test wall, I made sure that nothing could go through walls. I did this by trying to shoot flames through walls, move Penelope over a wall, lure zombies over walls, lure citizens over walls. I also tried to explode landmines near walls and make sure that they interact with walls as expected. In summary, the wall tests were trying to move all game objects into the walls and making sure that they are unable to do so.

Exit

For this class, the first thing I did was make sure that its functionality matched what was given in the spec, and that it behaved the same on the linux server. To test exits, I made sure that the only things that can go through exits initially were citizens, and only after all citizens were gone could Penelope go through the exit. I tried to walk over the exit when I still had remaining citizens which should not have done anything. I also tried to bring zombies over the exit, which should not have done anything. Last, I went over the exit after all citizens were gone. To make sure all citizens were gone, I printed the number of citizens remaining incrementally and watched the value change as citizens passed through the exit or died. I also tried to shoot flames and get zombies to vomit on the exit to make sure that the exit was not interfering with those operators.

Pit  
For this class, the first thing I did was make sure that its functionality matched what was given in the spec, and that it behaved the same on the linux server. In order to test a pit, I placed pits on the screen to start and made sure that zombies, Penelope, and citizens could fall into the pit. The definition of falling into the pit is that the object should die, which is what happened visually. I also made sure that when I exploded a landmine, that a pit was created at the center, and that the functionality was the same as the normal pits. I made sure that things like flames and vomit would not fall into the pits too.

Vomit:

For this class, the first thing I did was make sure that its functionality matched what was given in the spec, and that it behaved the same on the linux server. In order to test vomit, I had to first get the zombie functionality working. Once I knew that my zombies were working right, I moved Penelope around the screen and made sure I could get vomited on. For testing, I gave myself 100 vaccines to start so that I could keep playing and clearing my infection in order to make sure that zombies were vomiting on me as expected. I also made sure that my infection count was working right, and that if it hit 500, I would die. I tested the interactions between citizens and vomit too, with the only difference being that if citizens hit 500, then they will turn into zombies.

Landmine:

For this class, the first thing I did was make sure that its functionality matched what was given in the spec, and that it behaved the same on the linux server. For landmine testing, I wanted to test that landmines are blown up by the right actions. To do this, I shot a flame at the landmine and made sure it went off. Similarly, I lured a zombie to walk over the landmine and blow it up. Last, I walked Penelope over the landmine and saw that landmine blew up and Penelope died as expected. I also made sure that I could actually drop a landmine using the tab key, and that my number of landmines would decrease when I drop it. I placed the landmines in weird corners of the field to make sure that the flames were only being created if that direction was not blocked previously. This was actually the source of a big bug in my program, where I was checking if the direction was not blocked instead of checking if it was blocked. Another bug I had was that I was not checking if things were already dead so I tried to kill already dead landmines which threw a thread access error. Lastly, I made sure that when landmines exploded, any goodies in the vicinity would die. I tested this by dropping a landmine by a vaccine goodie, hitting it with a flamethrower, and making sure the vaccine goodie disappeared.

Penelope

For this class, the first thing I did was make sure that its functionality matched what was given in the spec, and that it behaved the same on the linux server. The key functionality to test from Penelope was that it could do all of the actions that the user could type in. I made sure Penelope moved up/down/left/right accordingly and did not run over anything that Penelope should not have run over. Then I made sure that when I hit tab, I could drop a landmine. I checked if I hit space if I fired flames and if I hit enter if the infection was cleared. I also made sure that when a zombie vomited on Penelope that Penelope was set to infected and that it’s infection count started going up. Lastly, I made sure that as Penelope walked around, it could pick up goodies, set off landmines, walk through exits, not walk through walls, etc. Essentially, I tested normal game functionality as a player.

Dumb Zombie:

For this class, the first thing I did was make sure that its functionality matched what was given in the spec, and that it behaved the same on the linux server. To test dumb zombies, I made sure that the Penelope could kill them. I also tested that Penelope maintained movement functionality, such as not being able to walk through walls, or go through other people like Penelope or the Citizen. I made sure that the dumb zombie was able to vomit on Penelope and the citizens, and that those actors were being set as infected. In order to test zombie movement path functionality, I printed out the movement path distance and the direction and made sure that every time the distance hit 0, I recalculated the direction. I also tested that when the zombie was blocked, that the distance was overridden and set to 0, warranting a direction recalculation.

Smart Zombie:

For this class, the first thing I did was make sure that its functionality matched what was given in the spec, and that it behaved the same on the linux server. For smart zombie, I tested it in almost the exact same way as Dumb Zombie, only I needed to focus on the direction of the smart zombie. The spec says that zombies should follow the nearest human object, so I moved Penelope near the zombie and observed the zombie coming toward the person, and likewise for zombies. For testing purposes, I manually changed the distance to 10000 so that the zombie would always be following a human and that made it easier to tell if my zombie was tracking. Also, I moved Penelope to each of the corners of the screen and observed that the zombie followed. I knew that the zombie will not change direction if the zombie is in the same line as Penelope, so I set up that situation and made sure that the zombie’s direction was the same by printing that out. I also made sure that once the zombie was coming toward Penelope, it may vomit on Penelope.

Citizen

For this class, the first thing I did was make sure that its functionality matched what was given in the spec, and that it behaved the same on the linux server. Testing the citizen required me to first have the zombie functionality implemented, since citizen’s movement can not only depend on Penelope but also zombies. To test citizen, I had to create situations were there was no obstruction and the citizen could just follow Penelope. I also had to create a situation were the X-direction, for example, was blocked, forcing the citizen to move in the Y-direction. I created situations were both the X and Y directions were blocked, which meant that the citizen had to flee from the zombie, instead of following Penelope. Lastly, I created the situation where the citizen could not move toward Penelope or away from zombie, in which case the citizen should not move at all. I also put many citizens near each other and that worked out too. On top of citizen movement, I tested if citizens could move through exits, and if so, if the number of citizens variable was being decremented. I made sure the zombies could vomit on citizens, and that they would be infected. If the infection ever hit 500, I made sure that either a dumb or smart zombie was created. Since it’s hard to tell the difference between a new dumb or smart zombie, I printed which type was being created and saw an approx. 70-30 split, which is expected.

StudentWorld

In order to test my Student World class, I basically tested the whole game, since the Student World class is responsible for handling all game items. I played the executable game approximately 20 times to get a good feel for how the game should run, and then I played my own game another 20 times to make sure that the functionality was the same. If I ever encountered anything weird when playing the official version or my version, I would try to recreate the situation in the other version and see if it is a glitch in the game itself or whether I had some thing wrong. I printed out the actors vectors repeatedly to make sure that actors were being added appropriately, such as when a flame is deployed, and that actors were also being removed appropriately, such as when a zombie is hit by a flame. I also printed out the distances of the nearest zombie that was being calculated and moved zombies closer together to make sure that my distance calculation was being done correctly.