Ian Conceicao

Professor Smallberg, TA Ling Ding

CS 32

Thursday, February 28th 2019

Project 3 Report

I. Function Descriptions

Class Hierarchy:

GraphhObject

Actor

Wall

ActivatingObject

Exit

Pit

Flame

Vomit

Landmine

Goodie

VaccineGoodie

GasCanGoodie

LandmineGoodie

Agent

Person

Player

Citizen

Zombie

DumbZombie

SmartZombie

1. Class: StudentWorld
   1. virtual int init()
      1. Init starts a fresh game, including setting the correct level, and placing all actors.
      2. Declared virtual because it is declared virtual in its base class, GameWorld
      3. In StudentWorld because the high level world class can see and control all actors
   2. int move()
      1. Calls all actors to move, If two actors collide, and one is an “ActivatingObject” calls the objects, such as vomit infecting, an exit being stepped on etc., deletes all dead objects, sets the game score
      2. Declared not virtual because the class is not meant to be inherited
      3. In StudentWorld because it controls all actors and the game
   3. void cleanup()
      1. Deletes all actors
      2. Declared not virtual because the class is not meant to be inherited
      3. I In StudentWorld because it controls all actors and the game
   4. bool actorWouldBeBlocked(Actor\* a, int x, int y)
      1. Returns if an actors new position at x and y would be valid
      2. Declared not virtual because the class is not meant to be inherited
      3. In StudentWorld because student world can easily see all actors
   5. bool canFlameExist(Flame \* flame)
      1. Specifically let’s a flame know if it can be made somewhere
      2. Declared not virtual because the class is not meant to be inherited
      3. In StudentWorld because student world can easily see all actor
   6. int distanceBetween(Actor\* a, Actor \*b)
      1. Returns the Euclidian distance between two actor’s centers
      2. Declare dnot virtual because the class is not meant to be inherited
      3. In StudentWorld because the method is higher level involving two actors of any type
   7. int nearesetZombieDistance(Citizen\* cit)
      1. Returns the distance of the nearest zombie for a given citizen
      2. Declared not virtual because the class is not meant to be inherited
      3. In StudentWorld because the method requires access to every zombie is higher level involving two actors of any type
   8. int nearestZombieDistance(int x, int y)
      1. Returns the distance of the nearest zombie for any given x and y
      2. Not virtual because the class is not meant to be inherited
      3. In sSudentWorld because needs to know the location of every zombie
   9. Bool shouldVomitBeMade(int x, int y)
      1. Checks if an infectible actor is in vomit range
      2. Not virtual, class should not be inherited
      3. In StudentWorld because needs to know location of every infectible actor
   10. Actor\* nearestPerson(Actor\* zombie)
       1. For a given zombie, returns nearest person, whether a person or citizen
       2. Not virtual, class should not be inherited
       3. In studentWorld because needs access to person and all citizens
   11. Void addActor(Actor\* a)
       1. Adds a new actor to the arena
       2. Not virtual, class should not be inherited
       3. In studentWorld because that way actors can add other actors, for example a zombie can add vomit, but actors do not have access to other actors (except the player)
   12. int getCitizenCount()
       1. Returns the number of citizens alive
       2. Not virtual, class should not be inherited
       3. In studentworld so exit can see how many citizens are alive and act accordingly, without having access to actors
   13. void decreaseCitizenCount
       1. Decreases the count of citizens
       2. Not virtual, the class should not be inherited
       3. In StudentWorld so the count is kept at a high level, but can be decreased by exit objects
   14. Void setNextRound
       1. Sets a nextRound variable to true
       2. Not virtual, the class should not be inherited
       3. In StudentWorld so exit objects can call it and decide to set the round to the next
   15. Player\* getPlayer()
       1. Returns the player object
       2. Not virtual, the class should not be inherited
       3. In studentWorld so other actors such as zombies can know the position of the student
   16. StuduentWorld(string assetPath)
       1. The constructor sets the assetPath to its own assetPath member variable
       2. Constructors cannot be virtual
       3. It’s a constructor, should be in studentworld
   17. ~StudentWorld()
       1. Calls the cleanup method described above
       2. Not virtual because the class is not supposed to be inherited
       3. It’s a destructor, should be in studentworld
2. Class: Actor
   1. Actor(int imageID, int startX, int startY, StudentWorld\* gameWorld)
      1. Basic constructor, takes as little info as necessary including imageID, starting position, and the gameWorld to make an actor, uses default direction of right, depth of 0 and scale of 1.0
      2. Not virtual, cannot have virtual constructors
      3. It’s a constructor, should be in Actor
   2. Actor(**int** imageID, **int** startX, **int** startY, Direction dir, **int** depth, **double** size, StudentWorld\* gameWorld)
      1. Complicated constructor, does what the basic constructor does but uses custom direction, depth, and size
      2. Not virtual, cannot have virtual constructors
      3. It’s a constructor, should be in Actor
   3. virtual ~Actor()
      1. Does nothing
      2. Virtual so derived class can do other things
      3. It’s a destructor, must be in Actor class
   4. virtual void doSomething() = 0;
      1. Makes every derived class have the doSomething method, makes Actor a pure abstract class so an Actor is never initialized
      2. Virtual so other class can make their own
      3. In actor class because every derived actor class must have it
   5. StudentWorld\* getGameWorldPtr() const
      1. Returns a pointer to the game world an actor exists in
      2. Not virtual, this function should not be overwritten
      3. In Actor class because we want every actor to have access to the world they exist in
   6. virtual void setDead()
      1. Sets an actor to dead
      2. Virtual, some actors have specific deaths such as playing sounds or decreasing counts
      3. In Actor class because every actor should be able to be set dead and removed
   7. bool isAlive() const
      1. returns if an actor is currently alive
      2. Not virtual, just a getter that should not be changed
      3. In Actor class because every actor should have a method checking if it is alive for the world class
   8. bool doesBlockAgents() const
      1. Returns if an actor blocks agents, e.g. citizens, the player, and zombies based off of a member variable
      2. Not virtual, the output of the method can be changed by changing a member variable, but the method should remain the same
      3. In Actor class because every actor should be able to say whether it blocks Agents
   9. bool doesBlockFlames() const
      1. Returns if an actor blocks flames
      2. Not virtual, the output of the method can be changed by changing a member variable, but the method should remain the same
      3. In Actor class because every actor should be able to say whether it blocks flames
   10. bool TriggersLandmines() const
       1. Returns if an actor triggersLandmines
       2. Not virtual, the output of the method can be changed by changing a member variable, but the method should remain the same
       3. In Actor class because every actor should be able to say whether it triggers landmines
   11. void setBlocksFlame(bool a)
       1. Sets a member variable that says if an actor will block flames
       2. Not virtual, just a simple setter that should not be overwritten
       3. In Actor class because every actor should be able to say whether it blocks slames
   12. void setBlocksAgents(bool a)
       1. Sets a member variable that says if an actor will block agents
       2. Not virtual, just a simple setter
       3. In Actor class because every actor should be able to say whether it blocks Agents
   13. void setTriggersLandmine(bool a
       1. Sets a member variable that says if an actor will trigger landmines
       2. Not virtual, just a simple setter
       3. In Actor class because every actor should be able to say whether it triggers landmines
   14. bool canGetFlameDamaged() const
       1. Returns if an actor can get flame damaged
       2. Not virtual, the output of the method can be changed by changing a member variable, but the method should remain the same
       3. In Actor class because every actor should be able to say whether it can get hurt by flames
   15. bool canItGetInfected() const
       1. Returns if an actor can get flame infected
       2. Not virtual, the output of the method can be changed by changing a member variable, but the method should remain the same
       3. In Actor class because every actor should be able to say whether it can get infected
   16. void setFlameDamaged(bool a)
       1. Sets a member variable that says if an actor can get flame damaged
       2. Not virtual, just a simple setter
       3. In Actor class because every actor should be able to say whether it can get flame damaged
   17. Void setCanGetInfected(bool a)
       1. Sets a member variable that says if an actor can get infected
       2. Not virtual, just a simple setter
       3. In Actor class because every actor should be able to say whether it get finfected
3. Class: Wall
   1. Wall(int startX, int startY, StudentWorld\* gameWorld)
      1. Creates a wall at a given position in a world. Set its ability to block flames and agents as true
      2. Constructor, not virtual
      3. It’s a constructor, needed in the class
   2. Virtual void doSomething()
      1. Does nothing
      2. Virtual for syntax because it is overwriting the base class’s do something
      3. Needed otherwise the class would be abstract
4. Class: ActivatingObject
   1. ActivatingObject(**int** IID, **int** startX, **int** startY, StudentWorld\* GameWorld)
      1. Simple constructor, makes an activating object with IID, at a position in a world
      2. Constructor, not virtual
      3. It’s a constructor, needed in the class
   2. ActivatingObject(**int** IID, **int** startX, **int** startY, Direction dir, **int** depth, **double** size, StudentWorld\* gameWorld)
      1. More complicated constructor, makes an activating object with specific direction, depth and size
      2. Constructor, not virtual
      3. It’s a constructor, needs to exist in that class
   3. Void doSomething()
      1. Does nothing
      2. Not virtual, activating objects do not do anything every tick
      3. In the activateobject class so all activating objects will have it automatically called and behave the same
   4. Virtual void activate(Actor\* actor) = 0
      1. Does nothing, requires all subclasses to have an activate method
      2. Virtual and set to 0 so all subclasses have to have this method
      3. In activating object so it makes the class pure abstract
5. Class: Exit
   1. Exit(**int** startX, **int** startY, StudentWorld\* gameWorld)
      1. Makes an exit object at a given position in a world
      2. Constructor, not virtual
      3. Constructor, needed in the exit class
   2. Virtual void activate(Actor\* actor)
      1. Called when activated by object actor, the exit checks if the actor is a citizen. If it is a citizen, then the citizen goes the exit. If it is a player and all citizens are dead or went through already, then the player gets to the next round
      2. Virtual for syntax reasons, the function is declared in the base class
      3. Exit needs an activate function
6. Class: Pit
   1. Pit(**int** startX, **int** startY, StudentWorld\* gameWorld)
      1. Creates a pit at a position in a world
      2. Constructor, not virtual
      3. Constructor, needed in the class
   2. Virtual void activate(Actor\* actor)
      1. Checks if the actor should die in the pit, if so it kills the actor
      2. Virtual for syntax, the method is declared in the superclass
      3. Pit needs an activate function
7. Class: Flame
   1. Flame(**int** startX, **int** startY, Direction dir, StudentWorld\* gameworld)
      1. Creates a flame at a position in a world with a direction dir
      2. Constructor, not virtual
      3. Constructor, needed in the class
   2. void doSomething()
      1. Returns right away if it is dead. Otherwise increases its tick count, kills itself if its tick count is over 2. (tickcount starts at 0)
      2. Virtual for syntax, the function is declared in the superclass
      3. Flames need their own specialized doSomething function
   3. virtual void activate(Actor\* actor)
      1. Checks if the actor can get flame damaged, kills the actor if it can. Checks if the actor blocks flames, if it does, set the flame to dead
      2. Virtual for syntax
      3. Needed in the class due to being an ActivatingObject
8. Class: Vomit
   1. Vomit(**int** startX, **int** startY, Direction dir, StudentWorld\* gameworld)
      1. Creates vomit at a position, with a direction, in a world
      2. Constructor, cannot be virtual
      3. Needed in the class, it’s a constructor
   2. Virtual void activate(Actor\* actor)
      1. Checks if the activator can get infected, if so, infect the actor
      2. Virtual for syntax
      3. Specialized to vomit class and required to be an Activating Object
9. Class: Landmine
   1. Landmine(**int** startX, **int** startY, StudentWorld\* gameWorld)
      1. Creates a landmine at a position in a world, sets its safety ticks to 30 and sets itself to unactive
      2. Constructor, not virtual
      3. Constructor is needed in this class
   2. Virtual void doSomething()
      1. Checks if safety ticks has run out of time, if so, it becomes active
      2. Virtual for syntax
      3. Needs a specialized method to account for ticks
   3. Virtual void activate(Actor\* actor)
      1. Checks if actor (the activator) triggers landmines and if it is current active, if so it crease a pit at its location and 8 flames in a square if the flames are possible to exist at the locations
      2. Virtual for syntax
      3. Needs a specialized activate function to be an activating object
10. Class Goodie
    1. Goodie(**int** IID, **int** startX, **int** startY, StudentWorld\* gameWorld)
       1. Creates a goodie at a position in a world, makes it possible to be hurt by flames
       2. Constructor, not virtual
       3. Constructor needed in its own class
    2. Virtual void activate(Actor\* actor) = 0
       1. Does nothing, cannot be called
       2. Makes Goodie class abstract, no Goodies can be initialized
       3. Requires all Goodies have an activate method
11. Class VaccineGoodie
    1. Goodie(**int** IID, **int** startX, **int** startY, StudentWorld\* gameWorld)
       1. Creates a vaccine goodie at a position in a world
       2. Constructor, not virtual
       3. Constructor needed in its own class
    2. Virtual void activate(Actor\* actor)
       1. Checks if the activator is the player, if so, give a vaccine to the player
       2. Virtual for syntax
       3. Needs its own activate function so it can give the player a vaccine
12. Class GasCanGoodie
    1. GasCanGoodie(**int** startX, **int** startY, StudentWorld\* gameWorld)
       1. Creates a gas can goodie at a position
       2. Constructor, not virtual
       3. Constructor, needed in this class
    2. Virtual void activate(Actor\* actor)
       1. Checks if the activator is the player, if so give 5 flames to the player
       2. Virtual for syntax
       3. Needs its own function so It can give the player 5 flames
13. Class LandmineGoodie
    1. LandmineGoodie(**int** startX, **int** startY, StudentWorld\* gameWorld)
       1. Creates a landmine goodie at a position
       2. Constructor, not virtual
       3. Constructor, needed in this class
    2. Virtual void activate(Actor\* actor)
       1. Checks if the activator is the player, if so give 2 land mines to the player
       2. Virtual for syntax
       3. Needs its own function so It can give the player 2 land mine
14. Class Agent
    1. Agent(**int** IID, **int** startX, **int** startY, StudentWorld\* gameWorld)
       1. Creates an agent a position in a world, sets itself as being able to trigger landmines, blocking other agents, and can be damaged by flames
       2. Constructor, doesn’t need to be virtual
       3. Constructor, needs to be in this class
    2. Virtual void doSomething
       1. Nothing, cannot be called
       2. Virtual so other agents are required to make their own
       3. Makes the class pure abstract
15. Class Person
    1. Person(**int** IID, **int** startX, **int** startY, StudentWorld\* gameWorld)
       1. Creates a person at a position, in a world, allows itself to be able to get infected
       2. Constructor, not virtual
       3. Constructor, must be in this class
    2. virtual void setInfected(bool input)
       1. Sets infected status to input, if they are being cured, reset infection count to 0
       2. Virtual so citizen can overwrite the function
       3. In Person because all persons can get infected
    3. Bool insInfected()
       1. Returns if a person is infected
       2. Not virtual, just a simple getter, don’t want to have overwritten
       3. In person because that is where infected stats are kept
    4. Int getInfectionCount()
       1. Returns a persons infection count
       2. Not virtual, just a simple getter, don’t want to have overwritten
       3. In person because that is where infected stats are kept
    5. Void increaseInfectionCount()
       1. Increases a person’s infection count by one
       2. Not virtual because it will never be modified or overwritten
       3. In person because that is where infected stats are kept
    6. Bool checkInfectionCount()
       1. If a person is infected, increase their infection count, if infection count is greater than 500, set them as dead and return true. Otherwise return false
       2. Not virtual because we do not need it overwritten
       3. In person because that is where infected stats are kept
    7. Virtual void setDead() = 0
       1. Nothing, cannot be called
       2. Virtual, = 0 makes the class abstract
       3. In Person because it makes the class abstract and guarantees each base class type make their own setDead method
16. Class Player
    1. Player(**int** startX, **int** startY, StudentWorld\* gameWorld)
       1. Creates a player at a location in a world
       2. Constructor, not virtual
       3. Constructor, must be in player class
    2. Void doSomething()
       1. Calls check infection count method, updating infection statuses. Handles user input movement, and using goodies that the player is holding— the player uses wasd or arrow keys to move, space to shoot flames, tab to deploy a landmine, enter to use a vaccine.
       2. Virtual for syntax
       3. In player because player has unique things to do
    3. Void increaseVaccCount(int i)
       1. Increases the number of vaccines the player is holding by i
       2. Not virtual, no other class needs this
       3. In Player because only player can hold vaccines
    4. Void increaseFlameCount(int i)
       1. Increases the number of flames the player is holding by i
       2. Not virtual, no other class needs this
       3. In Player because only player can hold flames
    5. Void increaseLandmineCount(int i)
       1. Increases the number of landmines the player is holding by i
       2. Not virtual, no other class needs this
       3. In Player because only player can hold landmines
    6. Int getVaccCount()
       1. Returns the players vaccCount for the studentWorld to know
       2. Not virtual, just a simple getter that only pertains to player
       3. Only the player can hold vaccines, not needed by any other class
    7. Int getFlameCount()
       1. Returns the players flameCount for the studentWorld to know
       2. Not virtual, just a simple getter that only pertains to player
       3. Only the player can hold flame, not needed by any other class
    8. Int getLandmineCount()
       1. Returns the players landmineCount for the studentWorld to know
       2. Not virtual, just a simple getter that only pertains to player
       3. Only the player can hold landmines, not needed by any other class
    9. Virtual void setDead()
       1. Sets the players status to dead and players the sound, sound player die
       2. Virtual for syntax
       3. In player class because its unique to player
17. Class Citizen
    1. Citizen(**int** startX, **int** startY, StudentWorld\* gameWorld)
       1. Creates a citizen at a position in a world
       2. Constructor, not virtual
       3. Constructor, needed in this class
    2. Virtual void doSomething()
       1. Checks infection status, if it has become infected, create a zombie at the location of the citizen. There is a 30% chance the zombie is a smart zombie. Do the following every other tick. If the player is 80 units away, but closer than all zombies, follow the player. Otherwise, if a zombie is closer than 80 units move away from the zombie.
    3. Void exited()
       1. Called when a citizen goes on an exit pad, the sound citizen saved is played and the citizen is set to dead without points being deducted. Note that points were already awarded in the exit class.
       2. Not virtual because it is specific to the citizen class
       3. In the citizen class because no other person
    4. Virtual void setDead()
       1. Sets the actor to dead, decreases score by -1000, plays the sound citizen died, decrease the citizen count in studentworld
       2. Declared virtual for syntax
       3. Citizen needs the specific setdead to play the sound and detract points
    5. Virtual void setInfected(bool input)
       1. If the player is just getting infected, plays the citizen is infected sound, and always call the person set infected method.
       2. Declared virtual for syntax
       3. Exists in citizen because citizen has to play a specific sound, but it does utilize players setInfected method
18. Class Zombie
    1. Zombie(**int** startX, **int** startY, StudentWorld\* gameWorld)
       1. Creates a zombie at a position in a world
       2. Constructor, not virtual
       3. Must be in this function, it’s a constructor
    2. Virtual void setDead()
       1. Set as dead, and play the zombie died sound
       2. Virtual, dumbzombie needs to modify it
       3. Both zombies get set dead in similar ways.
    3. Void doSomething() = 0
       1. Does nothing, cannot be called
       2. Pure virtual, derived classes must have it
       3. Makes the Zombie class pure abstract
    4. Void setParalyzed(bool a)
       1. Sets paralyzed member variable to a
       2. Virtual so it can be used by all types of zombie
       3. Since all zombies get paralyzed, paralyzed logic is handled by the base class
    5. Int getMovementPlanDistance()
       1. Returns the movementplan distance
       2. Not virtual, just a getter, does not need to be overwritten
       3. In zombies because both zombies have a movement plan
    6. Void setMovementPlanDistance(int i)
       1. Sets the movement plan distance to i
       2. Not virtual, just a setter, does not need to be overwritten
       3. In zombies because both zombies have a movement plan
    7. Bool hasVommited()
       1. Returns if a zombie just vomited
       2. Not virtual, vommiting is handled by super class
       3. In Zombies class because both zombies vomit
    8. Void tryAndMove()
       1. If the movement plan has not expired, try and move in the direction of the movement plan. If a move is not possible, set movement plan distance to 0
       2. Not virtual, does not need to be overwritten
       3. Both zombies need to try and move
    9. Void chooseRandomDir()
       1. Sets a random movement plan distance between 3 and 10. Chooses a random direction
       2. Not virtual, does not need to be overwritten
       3. In zombies class because both zombies need to choose random directions the same way
    10. Void vomitCalculations()
        1. Determine if vomit should be made with the help of the shouldVomitBeMade function in studentworld. If vomit should be made, then a third of the time add vomit in front of the zombie.
        2. Not virtual, other zombies classes should not rewrite this
        3. Zombies both vomit the same, therefore in their doSomething functions they should call this function
19. Class DumbZombie
    1. DumbZombie(int startx, int startY, StudentWorld\* gameWorld)
       1. Creates a dumbzombie at a position in the gameworld
       2. Constructor, not virtual
       3. Constructor, must be in the class
    2. Virtual void doSomething()
       1. If the zombie is paralyzed (every other tick) do nothing. Run the vomit calculations function, if the zombie just vomited, do nothing more. Check if the movement plan distance ran out, if so, choose a new direction. Try and move by calling the tryAndMove function in the zombie class
       2. Virtual for syntax
       3. In dumbzombie class because dumbzombie decides its movementplan different than a smart zombie
    3. Virtual void setDead()
       1. Call zombie set dead function, 1 out 10 times drop a vaccine goodie at the location
       2. Virtual for syntax (it is overwriting the base class function)
       3. In dumb zombie class because dumb zombies drop vaccines sometimes while smartzombies do not
20. Class Smart Zombie
    1. SmartZombie(**int** startX, **int** startY, StudentWorld\* gameWorld)
       1. Creates a smart zombie at a given location in a given world
       2. Not virtual, it’s a constructor
       3. Must be in the class, it’s a constructor
    2. Virtual void doSomething()
       1. If the zombie is paralyzed (every other tick) do nothing. Run the vomit calculations function, if the zombie just vomited, do nothing more. If the movementplan distance ran out then set a movement plan distance between 3 and 10. If the nearest citizen or player is greater than 80 pixels away, choose a random direction. Otherwise, set the direction to the direction of the nearest citizen or player (choose randomly if it is diagonal). Try and move by calling the try and move function.
       2. Declared virtual for syntax
       3. Made in the smartzombie class because the smart zombie needs to do different things than the dumb zombie.
    3. Void setDead()
       1. Call zombie setDead function, give 2000 score
       2. Virtual for syntax, it is overwriting another virtual function in the base class
       3. In smartZombie class because it needs to add a specific score

II. Failed Functionality

1) My score integer does not display with a fixed amount of digits. It shows “0” instead of “00000”.

III. Design Assumptions

None

IV. Tested Classes

**StudentWorld:**

I tested student world for having correct score keeping, sounds, and level progressing by playing the game. I picked up goodies, destroyed zombies and progressed through the levels to test the student world class. I also tested the helper functions that give distance by printing out the locations of zombies and the player.

**Wall:**

I ran into the walls with the player. I made sure citizens and zombies were blocked by walls. I shot fire at the walls. I tried to explode the walls.

**Exit:**

I shot fire at exits. I ran through exits under different circumstances, when all citizens were alive, when all citizens were dead, when some citizens had become zombies etc.

**Pit:**

I shot fire at pits, I ran into the pits, I lured citizens and zombies into the pits. I made sure a pit was made after a landmine

**Flame:**

I shot flames at everything to make sure the behavior was correct. I watched carefully the flame count on the top of the screen. I tried to shoot flames when I did not have any. I watched how many flames were being made each time.

**Vomit:**

I made sure that vomit was being deployed when an agent was in front of a zombie. I watched to see if citizens died because of vomit. I watch to see if the infection count rose correctly and the player died at 500.

**Landmine:**

I placed landmines down and lured citizens and zombies into them. I also ran into them myself. I made sure a pit was made where the landmine was and fire surrounding it.

**VaccineGoodie:**

I picked up vaccines and made sure I was given one. I watched the vaccine goodie disappear. I shot fire at the goodie and saw it disappear.

**GasCanGoodie:**

I picked up gas can goodies and made sure I was given 5 flames. I watched the gas can goodie disappear. I shot fire at the goodie and saw it disappear.

**LandmineGoodie:**

Same for the other goodies, but I made sure I was given two landmines instead.

**Player:**

I moved around using wasd and arrow keys. I ran into walls, people, goodies, pits, landmines, and vomit. I picked up goodies, including the gas can, landmines, and vaccines. I tested these power ups by pressing space, tab, and enter and seeing if the correct amount were used and place correctly. I ran through the exits. I died and ran out of lives.

**Citizen:**

I watched citizens die because of vomit, landmines, pits and fire. I made citizens follow me around the levels. I watched citizens turn into zombies. I watched the score counts change as citizens died and went through the exits.

**DumbZombie/SmartZombie:**

My testing for these classes was admittedly limited. I got near some zombies to see if some would follow me, and some did— those I assumed were SmartZombies. I let zombies vomit on me to see if they would correctly, and they did. I zombies with fire.