1. Description of all functions and their purpose:

virtual void doSomething()= 0;

* Does something for each object in the vector of actors
* Do something is a virtual class because all the last of the hierarchy objects implement what they do differently per tick.
* Since it doesn't make sense for actor itself to do anything during a tick, it is a pure virtual function

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

* Activates an object's particular attribute if it overlaps with another actor
* Similarly, since different objects get activated differently, the activateIfApt function is virtual.
* Again, since the function wouldn't make sense to be implemented in this class, it is pure ritual

virtual bool blocksMovement() const { return m\_blocksMovement; }

* A virtual const member function that returns if an object blocks movement by passing in a boolean that is a data member in the base class, down into the hierarchy.
* The function returns the data member which is initialized in the lower parts of the hierarchy.

virtual bool blocksFlame() const { return m\_blocksFlame; }

* A virtual const member function that returns if the respective object blocks a flame by passing in a boolean that is a data member in the base class, down into the hierarchy.
* The function returns something the data member which is initialized in the lower parts of the hierarchy.
* Useful for figuring out what object should be activated on, and generally identifying objects.

virtual bool canFallorBurn() const { return m\_canFallorBurn; }

* A virtual const member function that returns if the respective object can be fall down a pit, or be burned by flames by passing in a boolean that is a data member in the base class, down into the hierarchy.
* The function returns something the data member which is initialized in the lower parts of the hierarchy.
* Useful for figuring out what object should be set to dead as a result of interacting with pits, flames, and landmines, and generally identifying objects.

virtual bool triggersMines() const { return m\_triggersLandmines; }

* A virtual const member function that returns if the respective object can be triggers landmine explosions by passing in a boolean that is a data member in the base class, down into the hierarchy.
* The function returns the data member which is initialized in the lower parts of the hierarchy.
* Useful for figuring out what object should trigger a landmine explosion and generally identifying objects.

virtual bool threatensCitizens() const { return m\_threatensCitizens; }

* A virtual const member function that returns if the respective threatens citizenst, by passing in a boolean that is a data member in the base class, down into the hierarchy.
* The function returns the data member which is initialized in the lower parts of the hierarchy.
* Useful for figuring out what object is a zombie, landmine, flame, and pit

.

virtual bool triggersCitizens() const { return m\_triggersCitizens; }

* A virtual const member function that returns if the respective object that triggers a reaction from the citizens by passing in a boolean that is a data member in the base class, down into the hierarchy.
* The function returns the data member which is initialized in the lower parts of the hierarchy.
* Useful for figuring out what object should be set to dead as a result of interacting with pits, flames, and landmines, and generally identifying objects.

StudentWorld\* getWorld() const { return m\_world; }

* A const member function useful for accessing the StudentWorld public functions that allow us to interact with the game.
* Returns a data member that gets initialized in the creation of actors in StudentWorld.

virtual void setDead() { m\_isDead = true; }

* A virtual function that gets redefined in the subclasses when some of the classes cause the score to change, or trigger some kind of interaction with the game
* Sets the data member indicating an object isDead to true, thus, telling the StudentWorld to destroy it.

bool isInfected() const { return m\_isInfected; }

* As const member accessor of the Human class, the function’s purpose is to return if the Human has been infected.
* The function isn't virtual because it doesn't concern other classes other than Humans
* It makes sense to use these data members with their accessors and mutators because humans can be infected and have an infection count that changes

void setInfected() { m\_isInfected = true; }

* A mutator function for the isInfected bool data member that only concerns objects that are Humans.
* Therefore the implementation is only for Humans and doesn't change given a different kind of human, so its non virtual.

void incrementNumInfected() { m\_numInfected++; }

* A mutator that changes one of the data member that keep track of the infection count for a human.
* Only concerns Humans and doesn’t change amongst humans, so it is nonvirtual.

int infectionCount() const { return m\_numInfected; }

* A const member accessor function that return the amount a Human is infected for, to potentially do things that could interact with studentworld, zombies and other classes.
* The datamember only is interesting in Humans and it doesnt change amongst different kinds of humans, so it is nonvirtual.

void cureInfection() { m\_numInfected = 0; m\_isInfected = false; }

* A function that mutates a data member of a Human, while the function would only be used for Penelope, the implementation amongst humans wouldn't change, thus, it is nonvirtual.

int getNumLandmines() const { return m\_numLandmines; }

* A const member accessor function in Penelope that retrieves a data member that tracks the number of landmines that penelope has.
* This wouldn't be virtual because it only concerns penelope, thus the implementation wouldn't change in other classes, or be used.

int getNumFlames() const { return m\_numFlameThrowers; }

* A const member accessor function in Penelope that retrieves a data member that tracks the number of flames that penelope has.
* This wouldn't be virtual because it only concerns penelope, thus the implementation wouldn't change in other classes, or be used.

int getNumVaccines() const { return m\_numVaccines; }

* A const member accessor function in Penelope that retrieves a data member that tracks the number of vaccines that penelope has.
* This wouldn't be virtual because it only concerns penelope, thus the implementation wouldn't change in other classes, or be used.

void changeNumVaccines(int x) { m\_numVaccines += x; }

* A mutator function in Penelope that changes a data member that tracks the number of vaccines that penelope has.
* This wouldn't be virtual because it only concerns penelope, thus the implementation wouldn't change in other classes, or be used.

void changeNumLandmines(int x) { m\_numLandmines += x; }

* A mutator function in Penelope that changes a data member that tracks the number of landmines that penelope has.
* This wouldn't be virtual because it only concerns penelope, thus the implementation wouldn't change in other classes, or be used.

void changeNumFlames(int x) { m\_numFlameThrowers += x; }

* A mutator function in Penelope that changes a data member that tracks the number of flame throwers that penelope has.
* This wouldn't be virtual because it only concerns penelope, thus the implementation wouldn't change in other classes, or be used.

void shootFlameThrower();

* A function that creates new flames as a result of the player pressing tab and the being gas cans left. There are three flames produced in the direction penelope is facing for 2 ticks that stops once it hits an object that blocks flames
* This function isn't virtual being only penelope uses it as part of one of her attributes as te player

double distance(double dx, double dy, Actor\* a1, Actor\* a2);

* A function to get the distance between two actors given that one of them were to move in a horizontal direction dx away from its curr location and a vertically direction dy from its curr direction.
* The function is nonvirutal because it is implemented for all StudentWorld objects and actors the same

double closestZombieDist(double dx, double dy, Actor\* a);

* A function that iterates through the actors container and spits out the distance to a zombie from an actor a given a were to move in a horizontal direction dx away from its curr location and a vertically direction dy from its curr direction.
* The function is nonvirutal because it is implemented for all StudentWorld objects and actors the same

double\* newCoords(double db[2], Direction checkDir, int pixelsToMove);

* A function that outputs a coordinate in the form of a 2 element array of doubles of where an actor would be in the next tick if it were to move in direction checkDir for pixelToMove distance.
* The function is nonvirutal because it is implemented for all StudentWorld objects and actors the same

std::string readInLevel();

* Uses stringstream to read in the level
* The function is nonvirutal because it is implemented for all StudentWorld objects and actors the same

bool doesOverlap(Actor\* a1, Actor\* a2);

* Checks if a1 overlaps with a2
* The function is nonvirutal because it is implemented for all StudentWorld objects and actors the same

bool doesOverlap(double x, double y);

* Checks if the actor specified by x,y overlaps with any other actor in the container of actors and if the overlap is with a actor that blocks flames, a helper function for allwoing flames to work correctly.
* The function is nonvirutal because it is implemented for all StudentWorld objects and actors the same

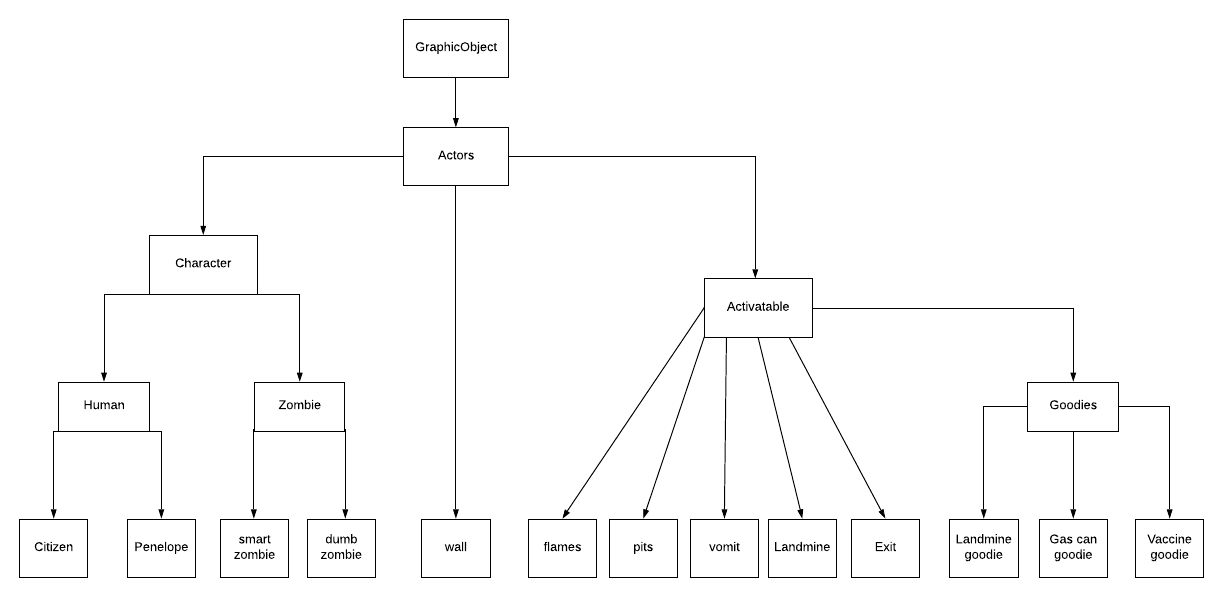
void updateGameStatText();

* Changes the status bar on top with next given information after move() is called and the tick is done
* The function is nonvirutal because it is implemented for all StudentWorld objects and actors the same

1. Issues encountereed

While developing the game, I ran into issues with the case of my citizen’s doSomething(). Essentially, the citizen’s running away function is glitchy. Furthermore, the Zombie and Vomit class were not implemented, but there is skeleton code developed for the start of some Zombie and Vomit class implementation.

1. Description of Heiarchy/Polymorphism of all major classes in the came

[](https://www.lucidchart.com/documents/edit/265fa73d-6405-4a68-9d71-0c155067f9e1/0?callback=close&name=docs&callback_type=back&v=1244&s=612)

Using this hierarchy implementation, I introduced several boolean variables that were kept inn actors but initialized at the bottom of the tree image. These booleans kept track of if the object could be triggered by a zombie/ vomit, could be threatened, could block movement, could block flame, etc. the purpose of this that it becomes handy in identifying the particular objects that you are dealing with in the container of actor pointers, so when a activeIfApt() function is called, you do the proper thing.

Furthermore, this structure was useful in defining small functions in the base class that could be used in the derived class. A simple is example is how the virtual function “setDead()” allowed certain objects to simply call the base class implementation version of the class, and how others called their own implementations that still used the base classes’ implementation of the base class.

4. Testing and strategy on developing classes for full game

Testing classes would be done, even base classes would be done by seeing how derived classes were implemented. Notely, derived classes seem to have the least information in its declaration, yet they have the most data in the program. Thus to test Actor, Character, Goodie, and other base classes, I would incrementally test the classes that derive from them.

To start with, I created the Penelope class, what required implementing the constructor of Human, which required implementing the constructor of character, which then finally required implementing the constructor of actor. After doing this, I would implement the pure virtual functions and the virtual functions that would require changes. As I tested, I would see how basic Penelope functions, such as moving, picking up goodies, and being stopped by walls and exists would all function. For Penelope, It was hard to test her comprehensively until I had implemented most of the other classes, thus i would go back and test her after making a new function in another class, or making a new class.

Next the wall class provided to be straightforward, Essentially I tested to see if my wall showed to have the proper attributes that it must in order to interact with other characters on the game. Consequently, after making sure the walls blocked, didn’t left flames, through, and a few other bool variables were set correctly in the initialization. The implementation was done.

Next to test the Citizens class. I had to consider the different things it does respectively. First I tested if the citizens would fallow penelope when penelope was in the same column as the citizen. Then after testing this, i would write and test if the citizens followed penelope when she was in the same row as the citizens. After, I tested to see if the random choice in a horizontal or vertical direction was seemingly random by comparing it to the executable on the site, and seeing if the random directions would bring the citizen closer to penelope. Finally, I tested the run away from Zombie part of doSomething(), however, i had several issues in having the function run in a direction that wasn’t blocked, and i never managed to fully get it to work.

After I implemented the short code necessary for Goodies to work. Then I implemented all the different kinds of goodies all at once since their code, and behavior was very similar. I first made sure that penelope could take a goodie, therefore increase that goodies count in the status bar (and consequently in the data members). Then I made sure that goodie would be set to dead and be released from the actor\* container and from the screen. I ran into some issues there when I would pick up a goodie, and it would increment the wrong goodie on the status bar. Eventually, I got that to work and moved on to test the different utilities the goodies provided.

Since Vaccine was least work, I did this first and got the implementation done in a few lines. The test was essentially to make sure that when I. had vaccines and I used them, my numInfected would reset, and penelope wouldn't be infected anymore. Since my zombies weren’t able to infect me, I would trace through the calls of the function that would so that vaccines did, and made sure it was being called in the right place at the right time.

Next, I moved on to writing and testing Flames, since landmines used flames. Flames required that I burn the right objects and not produce flames on exists and walls. Therefore, I would pick up gas can goodies and shoot them at different objects to see how they affected the score, the appearance of the object being shot, and how the flames came out from penelope ( for 2 ticks, of 3 flames). Another thing to worry about was that the sound was being produced by the game when firing flames.

Next I would test the landmine by picking up its goodie and checking the status bar. Then I would place the landmine and try to stand still to make sure it blew up and killed penelope. Next, I. would try to run after placing the landmine, which should activate the landmine soon after my exit of the trigger zone. However, I was running into error in getting that to work and decided to leave it until later. However, I wouldn’t understand why I was getting this issue, and played through the game on the site and saw that this happens in the executable version online too.

Last class that I was able to implement was the exit class. The testing involved making sure I incremented levels correctly, and only when all the citizens were saved and penelope was overlapping with the exit. Next I made sure that flames would be blocked from appearing on the exit. Finally, I made sure that citizens would disappear by overlapping the exit.

While I didn’t full implement the zombie and vomit class, if I were to have finished the classes, I would have tested how different zombies interact with citizens and penelope. Along with if they die in the cases that they should: falling in a pit, landmine explosion, flame burning, etc. Next I would make sure that the zombies didn’t do anything unexpected by playing through the completed game and comparing game performances. Lastly, I would keep an eye out for small things such as if a zombie ever overlaps an exit.