1. **Description of Public Member Functions**
   1. **baseClass**
      1. sub-class of GraphObject
      2. my baseClass implementation has 7 boolean variables that aid in indentifying the different types of classes and how they should react when overlapping with other object types: bool blocksMove, bool blocksFlame, bool blocksVomit, bool infectable, bool flammable, bool canInfect, bool isAlive. Each of these Booleans have a trivial get method that returns there Boolean value along with one modifier that edits the isAlive variable to allow our code to “kill” my actors;
      3. virtual void doSomething() = 0
         1. I chose to define a pure virtual version of doSomething() in my baseClass because every type of actor will have a unique doSomething() method.
   2. **Terrain class**
      1. Sub-Class of baseClass
      2. My terrain class has no public member functions
      3. virtual void doSomething()
         1. All terrains doSomething does is call a get method in order to retrieve a pointer to the student world then passes itself into the student world’s overlapAction method then returns.
         2. I made this function virtual and not pure virtual because while exit and pit which are sub-classes to terrain have identical doSomethings, walls doSomething is slightly different so terrains doSomething is defined how it should work for exit and pits while wall redefines the doSomething function.
   3. **Wall class**
      1. Sub-Class of terrain
      2. Virtual void doSomething()
         1. A wall doesn’t do anything so while I defined this function to keep wall from being an abstract base class is does nothing.
   4. **Exit class**
      1. Sub-class of terrain
      2. Virtual void doSomething()
         1. All exits doSomething does is call a get method in order to retrieve a pointer to the student world then passes itself into the student world’s overlapAction method then returns.
   5. **Pit class**
      1. Sub-class of terrain
      2. Virtual void doSomething()
         1. All pits doSomething does is call a get method in order to retrieve a pointer to the student world then passes itself into the student world’s overlapAction method then returns
   6. **Human class**
      1. Sub-class of baseClass
      2. void setInfectionStatus(bool temp);
         1. This function changes the Boolean variable InfectionStatus to whatever value is passed to the function(true or false). I defined this function as non-virtual within the human class because all sub-classes of the human class can be infected and will need at somepoint in most games to set their infection status to true or false;
      3. virtual void doSomething();
         1. This function first checks to make sure the object is alive. If the object is not alive it immediately returns and does nothing. Otherwise it then checks if the humans infections status is true in which case it increments the infection count by 1. Then after if the infection count is 500 it calls a protected member function that dicates the human’s reaction to being fully infected. This protected member function is virtual and redefined in all of human’s subclasses as they react different to being fully infected. After this another protected member function, uniqueAction(), is called to determine what each sub-class of human does during the rest of the tick. This protected member function is virtual and redefined in all of human’s subclasses as they each do something different at the end of their doSomething function.
      4. int returnInfectionCount()/void modifyInfectionCount(int n)
         * 1. Both of these functions are trivial get and modification methods for the private human variable infection count.
   7. **Penelope class**
      1. Sub-class of human
      2. No public member functions
   8. **Citizen class**
      1. sub-class of human
      2. void setDistP(double n) & void setDistZ(double n)
         1. These two functions set the citizens private variables of distP(distance to Penelope) and distZ(distance to nearest zombie) to the value “n” that is passed into the function. These functions are only necessary in citizen because they are the only actor that needs to keep track of these specific statistics.
   9. **zombie class**
      1. sub-class of baseClass
      2. void modifyPlanDist(int n) & int returnPlanDist();
         1. These two functions return and set a zombies movement plan distance. I defined this as non-virtual because every sub-class will have a movement plan distance and will need to modify them at some point.
      3. virtual void doSomething();
         1. this function is labeled virtual purely for clarification to allow readers to know it is a redefinition of baseClass’s doSomething() method. None of its sub-classes redefine this.
         2. First my doSomething() function checks if the zombie is dead or paralyzed. If it is then it sets the isParalyzed function to false then immediately returns. Otherwise it enters a switch statement based on the direction its faceing. Then in the direction it faces it calls StudentWorlds overlapAction function which tells the zombie if in that direction an infectable target is present. If there is an infectable target there is a 1 in 3 chance that the zombie vomits(spawning a vomit actor) in the direction its facing and returns doing nothing more. If the zombie doesn’t vomit or there is no target to even vomit on it then checks if its movement plan distance is 0. If it is then it calls a protected zombieMove function to reset its direction that is unique depending on the sub-class. Then another switch statement is called once again based on the direction the zombie is facing. The switch statement sets two variables dest\_x(destination x value) and dest\_y(desitnation y value) after which the function calls studentWorld’s canActorMove function. If this functions true the zombie moves to the destination coordinates and decrements the zombie’s movement plan distance. If the zombie cannot move to the desired coordernates the movement plan distance is set to 0. Finally the zombies isParalyzed variable is set to true.
   10. **dumbZombie class**
       1. sub-class of zombie
       2. no public member functions
   11. **smartZombie class**
       1. sub-class of zombie
       2. no public member functions
   12. **projectile class**
       1. sub-class of baseClass
       2. virtual void doSomething();
          1. This function is labeled virtual purely for clarification to allow readers to know it is a redefinition of baseClass’s doSomething() method. None of its sub-classes redefine this.
          2. First this function checks to make sure the projectile is alive. If it is not alive it returns immediately. Then if the projectile has been alive for more than 2 ticks is sets its alive status to false then returns. Then is calls a protected member function uniqueAction which is different among sub-classes. Then it increments how long it has been alive by one.
   13. **flame class**
       1. sub-class of projectile
       2. no public member functions
   14. **vomit class**
       1. sub-class of projectile
       2. no public member functions
   15. **goodie class**
       1. sub-class of baseClass
       2. virtual void doSomething();
          1. This function is labeled virtual purely for clarification to allow readers to know it is a redefinition of baseClass’s doSomething() method. None of its sub-classes redefine this.
          2. First this function checks to make sure the projectile is alive. If it is not alive it returns immediately. Otherwise it calls a protected member function uniqueAction which is different among sub-classes.
   16. **vacGoodie class**
       1. sub-class of goodie
       2. no public member functions
   17. **gasGoodie class**
       1. sub-class of goodie
       2. no public member functions
   18. **mineGoodie class**
       1. sub-class of goodie
       2. no public member functions
   19. **landmine class** 
       1. sub-class of goodie
       2. no public member functions
   20. **StudentWorld class**
       1. Sub-class of GameWorld
       2. virtual int init();
          1. My init function first loads the desired level. Then it checks if loading the level returns load\_fail\_file\_not\_found or if the level is above 99 in which case it returns that the player has won. It also checks if loading the level returns load\_fail\_bad\_format in which case it returns GWSTATUS\_LEVEL\_ERROR. It then enters a for loop that checks every coordinate for its contents and spawns in the correct actor. It then returns continue game
       3. virtual int move();
          1. first my move function interates through the list of current actors in the level and calls each one’s doSomething() function it then calls the player’s doSomething() function. Then it once again iterates through the list of current actors and if any of them are now dead it deletes them and removes that pointer from the list. If the player is dead it plays the sound SOUND\_PLAYER\_DIED and decrements the number of lives then returns GWSTATUS\_PLAYER\_DIED. It then chekcs if its private Boolean variable levelFinished is true in which case it resets it to false and returns GWSTATUS\_FINISHED\_LEVEL. Finally it updates the game text at the top of the screen.
       4. virtual void cleanUp();
          1. my cleanup function first sets the private Boolean variable gameOver to true. It then checks to make sure our player pointer does not equal nullptr. If it does not then it deletes the m\_player pointer and sets it to nullptr. It then iterates through the list of actors and deletes every actor. It then sets the number of citizens to 0 and resets Penelopes inventory to 0 for every item.
       5. bool spawnVomit(double x, double y, int dir)  
          bool spawnZombie(double x, double y, int type);  
          bool spawnFlame(double x, double y, int dir);  
          bool spawnPit(double x, double y);  
          bool spawnLandmine(double x, double y);  
          bool spawnVaccine(double x, double y);
          1. each of these functions are somewhat trivial and do very similar things. Each one will spawn the specified object at the give x and y values. For vomit and flame, the direction of the objects when the spawn is also determined by the argument “dir”. For spawnZombie, whether or not a smart or dumb zombie spawns is determined by the argument “type”.
       6. bool canActorMove(double dest\_x, double dest\_y, baseClass\* p);
          1. This function first iterates through the list of actors. If any actor is within 16 pixels in the x and y directions of the destination coordinates of whatever object is passed to the function (pointer p) and has the property that it blocks movement this function will return false indicating that the object passed to the function cannot move to the desired coordinates. It will then check the same thing specifically with the player.
       7. int returnNZombies()  
          bool returnGameOver();  
          int returnInven(string type);  
          int getnCitizens();
          1. These are all get methods that return the specified private variables. For the returnInven function the integer returned is the number of remaining items based on the string passed in.(e.g. if Penelope has 5 flames left you would pass in the string “flames” and the function would return 5).
       8. void modInven(int n, string type);
          1. This function will set the desired object specified by the string argument “type” to the integer “n”.
       9. void decCitizens();
          1. this function will decrement the private variable containing the number of citizens in the current level by one.
       10. void calcCitDists(citizen\* p, double x, double y);
           1. The purpose of this function is to calculate the dist\_z (distance to nearest zombie) and dist\_p(distance to player) variables of the citizen passed into the function. It does this by iterating through the list of actors in the current level. If an object in the list has the ability to infect(indicating it is a type of zombie), it will calculate the Euclidian distance to that object and if it the smallest distance calculated so far it will store it in variable. Once the entire list has been iterated through it will calculate the Euclidian distance to the player. It will then call the citizens setDistP and setDistZ functions to update the citizens dist\_z and dist\_p values.
       11. void smartCalcs(smartZombie\* p, double x, double y);
           1. The purpose of this function is to calculate the distances to all the citizens and the distance to player of the smart zombie passed in. . It does this by iterating through the list of actors in the current level. If n object in the list is infectable(indicating it is either the player or a citizen) it will calculate the Euclidian distance from the smart Zombie to the actor. It will then take the lowest calculated value and set that to a variable. If this distance is less than 80 then the smartZombie will be given a random direction. Otherwise if this distance is less than 80 the function will check if the smartZombie is in the same row or column as its target. If either is true it will set the smart Zombie’s direction facing towards its target. If the zombie is not in the same row or column as its target then it will calculate the two directions that would bring it closer to its target and will choose randomly between the two.
       12. int sameRowCol(citizen\* p);
           1. this function will take the citizen pointer passed in and determine if the object that the pointer points to is in the same row or column as the player and if it is whether or not the player is above, below, left, or right of the citizen.
       13. void twoDir(double x, double y, int\* arr);
           1. given the x and y coordinates passed the function will check which two directions from the given coordinates bring an object closer to the player and will add those two directions to the array passed
       14. int overlapAction(double x, double y, baseClass\* p, double less);
           1. this function takes in coordinates, a baseClass pointer and a double. The purpose of this function is to check if the given object(accesed via pointer p) will be within an Euclidian distance of “less” (the double passed in) of anything at the given x and y coordinates and if this overlap warrants any actions. Initially it will create a new vector of baseClass pointers. It will then iterate through the list of current actors and add any objects to the new vector that are an Euclidian distance of “less” (the double passed in) of anything at the given x and y coordinates. After this if the vector is empty the function will immediately return. Otherwise it goes through the vector of all objects that are overlapping with the object that “p” points to and will determine the appropriate action via the various Boolean variables stored as private members in our baseClass. The function will then either perform the function if possible or indicate the object pointed to by “p” to perform an action that only it can perform. (e.g. if it detects a citizen overlapping with an exit it will increase the score and tell the citizen to sets its alive status to false, etc. ). This function takes care of almost all interactions specified in the speck with a few edge cases being taken care of by other functions.
2. **Failed Functionality**
   1. All functionality was implemented successfully
3. **Design Decisions**
   1. It was not specified in the spec if the flames from landmines were supposed to go over walls. Within the finished game provided to us, the flames spawned by landmines did indeed go over walls however, within the spec it specifically mentions that walls have the property of blocking flames. Because of this I decided to not allow flames spawned by landmines to go over walls. Furthermore the spec did not specify where to handle Penelope’s inventory so I decided to have my StudentWorld class deal with keeping track of it using a map to link integers and strings. Other than this I felt that the specification document was otherwise clear on what was necessary and I had no need to make any other specifications.
4. **Description of Class Testing**
   1. **baseClass – ABC**
      1. baseClass is an abstract base class so its testing was incorporated in the testing of all its sub-Classes and making sure that they worked as intended which would indicate that this class is working.
   2. **Terrain**
      1. To test terrain I first made sure I was able to create the class since this should not produce any errors as it is not an abstract base class. I then made sure that in a test function I could pass in a pointer to a terrain object and that this allowed all my base classes to be passed in and that the test function ran as intended. Otherwise not other testing was done because while the terrain class is not an abstract base class it has no real place within the game and is there majorly for organization and simplification purposes.
   3. **Wall**
      1. In order to test the wall class I first made sure that it properly spawned into a level with the correct sprite and in the correct direction and orientation. Then once I knew my Penelope class was working I made sure that I was unable to walk through walls or clip into its bounding box. I also conducted this test with zombies and citizens making sure that walls also stopped their movement. Finally I tested the final integral property of walls by making sure that Penelope’s flamethrower didn’t overlap with walls and that flames from landmines were not spawned on top of a wall.
   4. **Exit**
      1. In order to test the exit class I first made sure that it properly spawned into a level with the correct sprite and in the correct direction and orientation. I then made sure that it didn’t complete the level if Penelope overlaps with it while there are still citizens in the level. Then once citizens were implemented I made sure that the exit properly added points and deleted citizens it overlaps with. I then tested to make sure that once all the citizens either died or were saved that it did complete the level when Penelope overlapped with it. I also made sure that the exit did block flames from both a flamethrower and from a landmine as intended.
   5. **Pit**
      1. In order to test the pit class I first made sure that it properly spawned into a level with the correct sprite and in the correct direction and orientation. I then made sure that it didn’t block any flames from any source as intended and that it didn’t block movement(which would stop any actor from overlapping with it). I then made sure that it correctly killed the player, any citizens, and any zombies that it overlaps with. I also made sure that it responded correctly based on what actor it killed such as decrementing score for a citizen, increasing score for a zombie, or ending the current iteration of the level if it kills the citizen.
   6. **Human – ABC**
      1. Human is an abstract base class so its testing was incorporated in the testing of all its sub-Classes and making sure that they worked as intended which would indicate that this class is working.
   7. **Penelope**
      1. In order to test the Penelope class I first made sure that it properly spawned into a level with the correct sprite and in the correct direction and orientation. I then first made sure that Penelope correctly moved in the right direction when each of the four directional keys were pressed or WASD were pressed. I then made sure that she properly got infected by a zombie vomiting on her and that she died when her infection count hit 500. I then tested that her flamethrower worked upon pressing the space button if and only if she had a positive number of flames in her inventory. I made sure that the flamethrower never spawned flames in walls or things that are intended to block flames. I then tested that her vaccines worked upon pressing the enter key if and only if she had a positive number of vaccines her inventory. I then tested that her landmines worked upon pressing the tab key if and only if she had a positive number of landmines her inventory.
   8. **Citizen**
      1. In order to test the citizen class I first made sure that it properly spawned into a level with the correct sprite and in the correct direction and orientation. I first made sure that the citizens would follow Penelope throughout the level. I then made sure that they would fun away from zombies throughout the level. During this process I also made sure that they would not run through walls. . I then made sure that they properly got infected by a zombie vomiting on them and that they died when their infection count hit 500. Finally I made sure that when they had the choice between running away from a zombie or running towards Penelope they decided what to do based on what object was closest.
   9. **Zombie – ABC**
      1. Zombie is an abstract base class so its testing was incorporated in the testing of all its sub-Classes and making sure that they worked as intended which would indicate that this class is working.
   10. **smartZombie**
       1. In order to test the smartZombie class I first made sure that it properly spawned into a level with the correct sprite and in the correct direction and orientation. I first made sure that the zombie would vomit correctly if their was a citizen or Penelope in front of them and that they would not vomit in any other circumstance. I then checked to make sure the smart zombie would follow the closest infectable actor to it in the correct matter if they were close enough. Otherwise I made sure that the zombie would move randomly and that they would not go through walls. Finally I tested that when this zombie dies that it correctly modifies the players score.
   11. **dumbZombie**
       1. In order to test the dumbZombie class I first made sure that it properly spawned into a level with the correct sprite and in the correct direction and orientation. I first made sure that the zombie would vomit correctly if their was a citizen or Penelope in front of them and that they would not vomit in any other circumstance. I made sure that the zombie would move randomly and that they would not go through walls. Finally I tested that when this zombie dies that it correctly modifies the players score. I also made sure that upon dying that occasionally a dumbZombie will throw a vaccine in a random direction.
   12. **projectile – ABC**
       1. projectile is an abstract base class so its testing was incorporated in the testing of all its sub-Classes and making sure that they worked as intended which would indicate that this class is working.
   13. **vomit**
       1. In order to test the vomit class I first made sure that it properly spawned into a level with the correct sprite and in the correct direction and orientation. I then made sure that if it overlapped with something that was able to get infected that it properly changed their status. I also used the games feature of being able to walk through tick by tick to make sure that it only existed for two ticks before removing itself form the level.
   14. **Flame**
       1. In order to test the flame class I first made sure that it properly spawned into a level with the correct sprite and in the correct direction and orientation. I first made sure that if it overlapped with any game object that is flammable that it correctly told StudentWorld to kill that object. I also used the games feature of being able to walk through tick by tick to make sure that it only existed for two ticks before removing itself form the level. Furthermore, I made sure that no flame every spawned overlapping with any other object that is supposed to block flames.
   15. **goodie – ABC**
       1. goodie is an abstract base class so its testing was incorporated in the testing of all its sub-Classes and making sure that they worked as intended which would indicate that this class is working.
   16. **vacGoodie**
       1. In order to test the vacGoodie class I first made sure that it properly spawned into a level with the correct sprite and in the correct direction and orientation. First I made sure that upon overlapping with Penelope that not only did it set its own state to dead by also correctly incremented the score and added the correct number of items to Penelope’s inventory. I also made sure that the goodie died when overlapping with a flame and that the only actor capable of picking it up is Penelope.
   17. **gasGoodie**
       1. In order to test the gasGoodie class I first made sure that it properly spawned into a level with the correct sprite and in the correct direction and orientation. First I made sure that upon overlapping with Penelope that not only did it set its own state to dead by also correctly incremented the score and added the correct number of items to Penelope’s inventory. I also made sure that the goodie died when overlapping with a flame and that the only actor capable of picking it up is Penelope.
   18. **mineGoodie**
       1. In order to test the mineGoodie class I first made sure that it properly spawned into a level with the correct sprite and in the correct direction and orientation. First I made sure that upon overlapping with Penelope that not only did it set its own state to dead by also correctly incremented the score and added the correct number of items to Penelope’s inventory. I also made sure that the goodie died when overlapping with a flame and that the only actor capable of picking it up is Penelope.
   19. **Landmine**
       1. In order to test the Landmine class I first made sure that it properly spawned into a level with the correct sprite and in the correct direction and orientation. I then made sure that it would correctly spawn when the player hits the tab key. I then made sure that during the first 30 ticks of its lifespan that it was unable to be triggered by any actor. Then once it is active I made sure that it did trigger when overlapping with zombies, citizens, or Penelope. Upon being triggered I made sure it would spawn flames in all directions around it as it was intended to do and that it correctly spawned a pit at its same location.
   20. **StudentWorld**
       1. To test studentworld I primarily tested the game as a lot of its functions are called by actors in various situations so if the actors work as intended I can assume that StudentWorld also worked as intended. However I did test certain functions within StudentWorld. First I made sure that the init function correctly spawned all actors in the correct place as specified by the level file. I then made sure that the init function would initialize the correct level and the level would increment upon completion. For the move function I made sure that if an actor’s Alive status was false that it was correctly removed from the actors list and no memory leaks occurred. I also made sure that the gametext was correctly updated every tick and that every actors doSomething function would only be called a single time. I also made sure that in between levels that the inventory was reset, infection count was set back to 0, and that lives were consistent. I also made sure that the correct screen would show if a level was completed and if the game was finished. Furthermore, with the various spawn functions that were included in StudentWorld I made sure that the object that was being spawned ended up in the correct position.