Report David Dinklage 305-087-789

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 Kontagion are able to sneeze, and each type of actor sneezes in a different way.”

**StudentWorld public member functions:**

Socrates\* thisSocrates();

**virtual** **int** init();

**virtual** **int** move();

**virtual** **void** cleanUp();

**double** distanceBetween(**double** x1, **double** y1, **double** x2, **double** y2);

**void** addEnemy(actorType enemyType, **double** xcord, **double** ycord);

**void** addProjectile(actorType projectileType, **double** xcord, **double** ycord, **double** direction);

**bool** overlapsWithFood(**double** xcord, **double** ycord); *// returns true if overlaps AND sets the food to dead*

**bool** overlapsWithDirt(**double** xcord, **double** ycord);

**bool** overlapsWithDamagable(**double** xcord, **double** ycord, **int** damage); *// returns true if overlaps AND does damage*

Direction findClosestFood(**double** xcord, **double** ycord);

thisSocrates() just returns the pointer to the Socrates object for other actors to use

init, move and cleanup just does what the spec specifies.

distanceBetween simply returns the distance between two coordinates

addEnemy adds a new Enemy to the game, specified as regular salmonella, aggressive salmonella, or Ecoli. It spawns the enemy at the given coordinate. It needs to be a public function so that the bacteria themselves can split and add new enemies.

addProjectile adds a projectile to the game at the specified coordinate with a specified direction. Public function so that Socrates can add projectiles

overlapsWithFood returns true if the given coordinate overlaps with food, and it sets the food to dead if it overlaps. Public so that enemies can see if they overlap with food

overlapsWithDirt returns true if given coordinate overlaps with dirt. Public so that enemies and projectiles can use it.

overlapsWithDamageable returns true if it overlaps with an actor that can take damage by a projectile. It also does damage via a public virtual method in the actor class that either does the correct amount of damage or kills the object if it has no hitpoints and is damageable.

findClosestFood returns the direction to the closest food object to the given coordinates. It’s public so that the enemies can use it in their movement search.

**Actor Public member functions:**

**virtual** **void** doSomething() = 0;

**virtual** **void** doDamage(**int** damage) = 0;

*// when objects are initialized in init(), they must not overlap with non - overlappable objects*

**virtual** **bool** isOverlappable() = 0;

**virtual** **bool** needsToBeDestroyedToFinishLevel() = 0;

**virtual** **bool** blocksPath() = 0;

**virtual** **bool** isDamagable() = 0;

**double** distanceBetween(Actor& a, Actor& b); *// returns the distance between two actors*

**bool** overlapsWith(**bool** isMovement, Actor& other); *// returns true if two existing actors overlap*

**bool** isAlive();

**void** setDead();

StudentWorld\* thisWorld();

doSomething is a pure virtual function, because a generic actor can’t do much of anything but every actor does something different.

doDamage is also pure virtual because a generic actor can’t take any damage, and every actor takes damage ( if it does at all ) in a different way

isOverlappable is a pure virtual function that returns true if other objects are allowed to be placed on top of them when they are created in the private populateActors method in StudentWorld. It’s pure virtual because it depends on the actor whether or not it can be overlapped.

needsToBeDestroyedToFinishLevel simply returns true if the object is a Pit or any type of enemy. Pure virtual because it returns true for some objects and false for others.

blocksPath returns true if the object blocks the path of an enemy. Pure virtual because Dirt returns true, while everything else returns false so that enemies can know whether there is something in their way.

isDamagable returns true if the actor can take damage or die from a projectile. Pure virtual because it depends on the actor or not whether it’s damagable.

**All of the above functions are pure virtual, and not just defined in their respective classes because we need to be able to call them from the actorVector.**

distanceBetween is a simpler version to use than the one defined in StudentWorld because you only need to input the actors.

overlapsWith returns true if two actors overlap. The variable movement specifies whether we are dealing with movement overlap or not.

isAlive determines whether an actor is alive or not. All actors use this same method.

SetDead is also used by all actors if we need to set them to dead so that they are removed from the screen.

thisWorld returns a pointer to the GameWorld we use. Very important that all actors have access so that all actors can access the game in some way.

**Sprite public member functions**

**int** getHealth();

**void** setHealth( **int** amt);

**virtual** **bool** blocksPath() { **return** **false**; }

All playable characters have health, and getting that value does the same thing for all Sprites, so getHealth simply returns the health of a sprite.

setHealth should work the same for all sprites, so it’s not a pure virtual function.

no Sprite can block the path of another object, so blocksPath always returns false

**Enemy public member functions**

**virtual** **bool** isOverlappable() { **return** **true** ; }

**virtual** **bool** needsToBeDestroyedToFinishLevel() { **return** **true**; }

**virtual** **bool** isDamagable() { **return** **true**; }

**virtual** **void** doDamage(**int** damage);

**int** movementPlanDistance();

**int** n\_foodsEaten();

**void** updateMovementPlanDistance(**int** amt);

**void** updateNfoodsEaten(**int** amt);

**void** moveRandomly();

**void** standardMovementSearch();

**void** multiply();

**void** hurtEatMove(**bool** doRegularmovement, **int** damage);

**void** setDirectionToSocrates();

Enemies can be placed on top of one another, so they are all overlappable. They all need to be destroyed in order to finish the level, and they all can take damage from projectiles.

the doDamage function decrements the health using setHealth of the enemy, with the correct amount. Works the same for any Enemy so no need to be pure virtual.

movementPlanDistance just returns the private member variable movementPlanDistance and should work the same for any Enemy so no need to be pure virtual.

n\_foodsEaten returns the number of food eaten for any Enemy, and it works the same for all enemies so it’s not pure virtual.

The next two update functions update the movement plan distance and the number of foods eaten respectively. They work the same for all enemies so there’s no need to make it virtual.

moveRandomly is a function called when an Enemy simply changes its direction to a random direction. Any Enemy can use it.

standardMovementSearch completes the steps 5 and 6 for a regular salmonella. An aggressive Salmonella also does standard movement after checking for other things

multiply is called when any Enemy needs to duplicate because it has eaten enough food. All enemies use so not virtual.

hurtEatMove describes the extent of an entire regular Salmonella’s actions. The other enemies also do these actions at some point during their doSomething method, but in a different order.

setDirectionToSocrates just points the player towards wherever Socrates is. Ecoli and Aggressive Salmonella both do this so it’s defined in the generic Enemy class.

**Regular Salmonella, Aggressive Salmonella, Ecoli**

Each of these only has a doSomething method, which calls a combination of the above functions, and a little more. doSomething is defined at this level because each enemy type does something different.

**Socrates public member functions**

**virtual** **void** doSomething();

**virtual** **void** doDamage(**int** damage);

**virtual** **bool** isOverlappable() { **return** **true**; }

**virtual** **bool** needsToBeDestroyedToFinishLevel() { **return** **false**; }

**virtual** **bool** isDamagable() { **return** **false**; }

**void** posAngleToXY(**int** posAngle, **double**& x, **double**& y);

**int** getPosAngle();

**void** changePosAngle(**int** newAngle);

**void** addFlames(**int** numFlames);

**void** resetSprays();

**int** getNflames ();

**int** getNsprays ();

**void** decFlames ();

**void** decSprays ();

Socrates has a specifc doSomething method detailed in the spec.

Socrates’ own projectiles don’t do damage to him, so his doDamage function doesn’t do anything.

Socrates can be placed on other game objects when created, he does not need to be destroyed to finish the level, and he can not be damaged by his own projectiles.

posAngleToXY converts his angle on the petri dish to X,Y coordinates.

getPosAngle returns his angle on the dish.

changePosAngle changes his positional angle on the dish

addFlames is used to increase flames by 5 when goodie is picked up

resetSprays resets his sprays to 20 if no key is pressed during a tick

the next two getter functions just return the number of flames and sprays left in Socrates inventory

the next two dec functions just decrease flames or sprays when Socrates uses them

**Goodie public member functions**

**int** getLifetime();

**void** decLifetime();

**virtual** **void** doSomething();

**virtual** **void** doDamage(**int** damage);

**virtual** **void** doSpecializedGoodieStuff() = 0;

**virtual** **bool** isOverlappable() { **return** **true** ;}

**virtual** **bool** needsToBeDestroyedToFinishLevel() { **return** **false**; }

**virtual** **bool** blocksPath() { **return** **false**; }

**virtual** **bool** isDamagable() { **return** **true**; }

Each goodie does roughly the same thing, so a doSomething method is defined here

each goodie just dies when it takes any damage

A pure virtual function is defined doSpecializedGoodieStuff, because each goodie will do something special in its doSomething method

Goodies can be placed on other objects at initialization, they do not need to be destroyed to finish the level, they do not block the path, and they can be damaged by projectiles.

**RestorehealthGoodie, FlamethrowerGoodie, ExtraLifeGoodie, Fungus**

Each of these only has a doSpecializedGoodieStuff method, which accounts for anything specific that a generic goodie doesn’t do.

**Projectile member functions**

**int** getMaxTravelDistance();

**int** getDistanceTraveled();

**void** addDistanceTraveled(**int** amt);

**virtual** **void** doSomething();

**virtual** **void** doDamage(**int** damage);

**virtual** **bool** isOverlappable() { **return** **true**; }

**virtual** **bool** needsToBeDestroyedToFinishLevel() { **return** **false**; }

**virtual** **bool** blocksPath() { **return** **false**; }

**virtual** **bool** isDamagable() { **return** **false**; }

getMaxTravelDistance returns the maxTravelDistance for any projectile.

getDistanceTraveled returns the current distance traveled for any projectile.

addDistanceTraveled is used to update the distance traveled for any projectile.

Every projectile basically does the same thing, so doSomething is defined here

every projectile can not take damage from itself, so its doDamage just does nothing

Every projectile can be placed over other objects, does not need to be destroyed in order to finish the level, does not block the path of anything, and can not be damaged by other projectiles.

**Spray, Flame**

Sprays and Flames don’t have any additional functions defined.

**Dirtpile public member functions**

**virtual** **void** doSomething();

**virtual** **void** doDamage(**int** damage);

**virtual** **bool** isOverlappable() { **return** **true**; }

**virtual** **bool** needsToBeDestroyedToFinishLevel() { **return** **false**; }

**virtual** **bool** blocksPath() { **return** **true**; }

**virtual** **bool** isDamagable() { **return** **true**; }

Dirptile’s doSomething does nothing, and it’s doDamage just kills it straight up

Other objects can be placed on top of a dirtpile, they do not need to be destroyed to finish the level, they do block the path of enemies, and they can be damaged by projectiles

**Food public member functions**

**virtual** **void** doSomething();

**virtual** **void** doDamage(**int** damage);

**virtual** **bool** isOverlappable() { **return** **false**; }

**virtual** **bool** needsToBeDestroyedToFinishLevel() { **return** **false**; }

**virtual** **bool** blocksPath() { **return** **false**; }

**virtual** **bool** isDamagable() { **return** **false**; }

Food’s doSomething does nothing, as does doDamage. Other objects can not be placed on top of them, they do not need to be destroyed to finish the level, they don’t block any path, and they can’t be damaged by projectiles.

**Pit public member functions**

**virtual** **void** doSomething();

**virtual** **void** doDamage(**int** damage);

**virtual** **bool** isOverlappable() { **return** **false**; }

**virtual** **bool** needsToBeDestroyedToFinishLevel() { **return** **true**; }

**virtual** **bool** blocksPath() { **return** **false**; }

**virtual** **bool** isDamagable() { **return** **false**; }

**void** decNumEnemy(actorType type);

**bool** hasInventory();

actorType pickEnemy();

Pit’s doSomething does as detailed in the spec, and it’s doDamage does nothing.

Other objects can not be placed over a pit, it does need to be destroyed to advance to the next level, it doesn’t block the path of an enemy, and it is not damageable by projectiles.

decNumEnemy decrements the number of enemies of a certain type

hasInventory returns true if the inventory is not empty

the pickEnemy returns a random enemy that still is in capacity

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 aggressive salmonella doesn’t work correctly yet so I treat it like a regular salmonella right now.”

I finished all of the functionality  
   
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.”

For the Status bar, I didn’t know how many spaces could go in between the colon : and the item. ie Level: 1. I just used setw(k) where k was determined to produce a spacing where it looked the same as the playable game, and if the value changed to have more digits the whole bar would not be shifted. I assumed we should have a negative score look like -00250 instead of 00-250 as it looked in the playable game.

It was not specified what to do if the Aggressive Salmonella kept changing direction a lot of times when it reached Socrates. It looked like it got to Socrates and then twitched uncontrollably because it kept re-updating to get closer to him. I set those bacteria to instead not move if they were already within 2 pixels of Socrates, so that they would behave as they appear to do in the playable version.

The spray projectile was specified to be placed SPRITE\_WIDTH pixels from Socrates, but when I place it there, it starts much too far from Socrates when comparing to the playable game, and enemies weren’t even harmed by the spray if they were overlapping Socrates. I changed this distance to be SPRITE\_RADIUS/2. ( 2 pixels ) as to be consistent with the way that the playable game looked.

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

StudentWorld:

The main functions in my StudentWorld class were the init(), move(), and cleanup() functions. I first implemented a shorter version of init() and added to it, so I tested it by playing the game and making sure that the correct actors were brought to the screen as I gradually added functionality. I wrote the move function to make every actor do something, and I tested it by watching every actor do something as I added them to my game. I made sure that every actor did something and that the requirements of the spec were met by playing the game over and over throughout the process, always comparing it to the playable version of the game. Whenever there was something wrong, I looked at my StudentWorld class first, because that was where the higher-level logic was. I tested cleanup by making sure that all the actors were deleted using the debugger. (I set them to null pointer after deleting)

Actor:

The first child class of Actor I made was dirtPile. Since I wasn’t able to actually make an Actor object, I made sure the class was constructed properly and derived by my dirtPiles being able to show up on the screen. The fact that all other sub classes worked was proof enough to me that I had my Actor class working.

Sprite:

For the Sprite class, I knew it worked when all Enemy, and Socrates classes worked, as they were derived from Sprite and Sprite had to work in order for the sub classes to work.

Enemy:

I knew the Enemy class worked when each of the regular Salmonella, Aggressive Salmonella, and Ecoli classes worked. I introduced each of these subclasses into the game, one at a time in isolation so that I could make sure that they all worked on their own without any connection to the pit before I made them work with the pit. When all three enemies did what they needed too on their own, I knew the enemy class had to be working as I got no errors and its subclasses worked.

Regular Salmonella:

To test the regular salmonella, I used init() to spawn a single bacteria to the center of the screen. I commented out sections of the code to make sure that each specific part of the code made the regular salmonella act the way it should. I watched the way it moved, reacted to food, and behaved similarly to the playable version. With one bacteria in the dish, I was able to see that when it ate three food it divided properly as well. After the regular salmonella worked from a hardcoded spawn, I made sure it worked as it came out of the pit.

Aggressive Salmonella:

I tested the aggressive salmonella much the same way as the regular salmonella. I placed it down in isolation. The fact that I used a lot of the same functions that the regular salmonella did let me mainly focus my testing on the extra behaviors of the aggressive salmonella. I purposely tried to get Socrates as close as possible to the Aggressive salmonella over and over to make sure that Aggressive Salmonella never went off the game screen, went through dirt, and tracked Socrates as he moved around. I basically just played the game a bunch of times with Aggressive Salmonella in it, and then I made aggressive salmonella come out of the pit to make sure that worked.

E coli:

I tested Ecoli by isolating it from the other bacteria first. I made sure it met the requirements of the spec in the way it tracked Socrates specifically, and I made sure it behaved the way it did in the playable game as well. I then let it spawn out of the pit to make sure that it interacted with the pit and other bacteria correctly as well.

Socrates:

I first added the move functionality. This was easy to test because either he moved correctly or he didn’t. It was easy to see this by just playing the game. I then added functionality and tested as I went along. I made sure each key Press did the right thing as I added it, and I made sure his sprays and flames changed accordingly when they were added. One of the first things I did, in order to test Socrates, was I implemented the status bar at the top, so I could see when Socrates’ values changed accordingly. I first added goodies, and made sure that Socrates’ score and health changed, and then I added the enemies to make sure Socrates’ health changed and that he died when he needed to.

Goodie:

I tested the functionality of my Goodie class by playing the game a lot to make sure that each of my specialized Goodie and Fungus classes behaved the way they should. When all the subclasses did the right thing, I knew Goodie had to be working.

RestoreHealth Goodie:

I tested this Goodie by changing the spawn rate to a lot more and making the RestoreHealthGoodie the only one to spawn. I made sure that Socrates’ health was properly restored and that the points were added correctly. I also made sure the Goodie went away after some time. I then had the goodie spawn with the other goodies too to make sure they behaved the way they appeared too in the playable game.

Flamethrower Goodie:

I tested the Flamethrower Goodie by isolating it on the game screen so that it was the only goodie to show up. I made sure that the flames correctly changed in the status bar, and the score updated correctly. I then let the flamethrower goodie spawn along with the other goodies to make sure it behaved the same way the playable game looked.

Extra Life Goodie:

I tested the Extra life Goodie the exact same way as the previous two goodies. I made sure that it worked on its own using the status bar, and then I made sure it worked in conjunction with the other goodies.

Fungus:

I tested the fungus the same way as the Extra Life Goodie, and I made sure that a negative score displayed correctly on the screen. When the fungus acted the same way it should be compared to the playable version, I called it a day.

Projectile:

I tested the projectiles individually the same way, by first implementing them, and then letting Socrates space or enter bar fire them so I could see what they did. I used the f command to freeze the game and make sure that they looked the same as the playable version on each game tick.

Spray:

To test the spray, I watched how Socrates sprayed. I held down the space bar in order to get the distance right and make sure that a spray acted the same way it should as in the playable game. At this point, I had implemented everything else besides the flame, so I made sure it interacted with each object the way it should be spraying the sprayer at each object when it spawned.

Flame:

The flame was the last thing I implemented, so when I implemented it, I made sure it interacted with all other game objects well. I used the f command to make sure that the same type of ring shape was created as the playable version, and I ran multiple cases of stepping through the frames to make sure that flames properly died when it interacted with damageable objects.

Dirtpile:

The dirt pile was the second thing I made after Socrates, so I just made sure that they were able to spawn correctly and prove that my Actor class could work. I just had to make sure that other objects interacted with the dirt as they should, and that dirt was created in the correct spot. When I noticed that everything was working properly I called it a day.

Food:

I made sure the food disappeared when eaten, and that it spawned correctly not overlapping any dirt piles or pits. To do this I uppped the number of food to be created so I could more easily see that no food was overlapping. I made sure that the enemies properly ate the food when I tested one enemy at a time so I could watch one enemy on the screen eat food to make sure it did the right thing.

Pit:

I was convinced pit was working when it correctly spawned all of the enemies and then disappeared. I made sure that the right count of enemies was spawned by removing the food objects so that the Enemies could not multiply. I made sure the pit objects did not overlap with any other objects by upping the number of pits to be quite high so I could clearly see that Pits were created with no overlap.