Bonnie Liu

005300989

CS 32 Discussion 1F

Project 3 Report

**High-Level Description**

My base class for this project was Actor, which is derived from the given Graph Object class. Different Actors can have different image IDs, x-coordinates, y-coordinates, directions, and depths. I included a pointer to Student World, which I named m\_world, to call functions in the Student World class later. For that, I included a public, non-virtual, constant world() function to access m\_world. I chose to make isDead() a public, non-virtual, constant function because every object being in a dead or alive state allows the Student World class determine whether or not to destruct the object and delete the Actor pointer from the list. Of course, to do so, I also needed a public, non-virtual function called setDead() to alternate an Actor’s state from alive to dead. Additionally, I included the public, nonvirtual, constant function isOverlapping(int x, int y), which returns true if the Actor overlaps with the passed-in x-coordinate and y-coordinate. Similarly, getDistance(double x, double y) returns the Euclidean distance from the Actor to the point (x,y). Lastly, I included the following public virtual functions:

* takeDamage() returns true if this Actor can suffer damage and makes it do so. (This applies to all Agents.) Otherwise, it returns false.
* blocksBacteriumMovement() returns true if the derived subclass is Dirt. It returns false for all other derived subclasses. This function is constant.
* isEdible() returns true if the derived subclass is Food. It returns false for all other derived subclasses. This function is constant.
* preventsLevelCompleting() returns true if the derived subclass is Pit or Bacterium. It returns false for all other derived subclasses. This function is constant.
* doSomething() is a public, pure virtual function, so Actor is an Abstract Base Class. doSomething() is different for different types of Actors, so all derived classes must implement their own version of doSomething().

My Dirt class was derived from Actor. When constructed, my Dirt class starts with a hit points value of 1. This means that as soon as a Dirt pile gets hit by a projectile, it will be setDead() and disappear since its HP will drop to 0 or below. The Dirt class’s version of takeDamage() does indeed take away its hit points, but it actually returns false so that we can distinguish between Bacterium and Dirt when we use takeDamage() in our Student World class. When asked to doSomething(), it does nothing and simply returns. As mentioned before, Dirt does in fact block bacterium movement, so blockBacteriumMovement() returns true. Since blockBacteriumMovement() only returns true for our Dirt class, it is effectively a way for distinguishing if an Actor is a Dirt pile when we use polymorphism in our Student World class.

My Food class, also derived directly from Actor, is edible, so isEdible() returns true, as mentioned before. Since isEdible() only returns true for our Food class, it is effectively a way for distinguishing if an Actor is a Food item when we use polymorphism in our Student World class. Whn asked to doSomething(), it does nothing and simply returns.

My Pit class is directly derived from Actor. Each Pit starts with 5 Regular Salmonella, 3 Aggressive Salmonella, and 2 E. Coli, and in the constructor, I used integer variables to keep track of the number of each type of bacterium yet to be released from the Pit. In doSomething(), if the Pit is empty, I set the Pit to dead and returned immediately. Otherwise, there is a 1 in 50 chance per tick that the Pit released a Bacterium. If it does release a Bacterium, there is a 50% chance that that Bacterium is a Regular Salmonella, a 30% chance that it is an Aggressive Salmonella, and a 20% chance that it is E. Coli. To simulate the combined effect of these probabilities, I generated a random number between 1 and 500 using the provided randInt() function. If the generated number was between 1 and 5 (inclusive), I released a Regular Salmonella and decremented the Pit’s count for Regular Salmonella. If the generated number was between 6 and 8 (inclusive), I released an Aggressive Salmonella and decremented the Pit’s count for Aggressive Salmonella. If the generated number was 9 or 10, I released an E. Coli and decremented the Pit’s count for E. Coli. Finally, if the generated number was none of those, Pit does nothing for that tick. As mentioned before, Pit is one of the derived subclasses whose existence in Student World signals that the level has not yet been completed; hence, the preventsLevelCompleting() returns true for Pit. This, in combination with the preventsLevelCompleting() function for Bacterium, allows us to ensure the level has been completed in the Student World class.

Projectile is an Abstract Base Class derived from Actor. The constant, pure virtual void function beImplemented() ensures that a Projectile object can never be instantiated; instead, every instantiable class derived from Projectile (in this case, Flame and Spray) must implement its own beImplemented() method. Since there were no big differences between a single Flame object and a single Spray object aside from their abilities to damage and their maximum distance to travel, which I included as member variables in the constructor, beImplemented() simply returned without doing anything. beImplemented()’s only function was to ensure that the Projectile class could become an Abstract Base Class. In Projectile’s doSomething() function, if the Projectile was dead, it returned immediately. Otherwise, it tries to damage an actor by calling the damageOneActor(Actor\* a, int damage) function in Student World. If another Actor is successfully damaged, the Projectile has done its work and can now be damaged. Otherwise, it will continue moving forward in its original direction. If the Projectile has reached the maximum distance travelled, it will set itself to dead.

The Spray and Flame classes are both derived from Projectile.

Goodie is an Abstract Base Class derived from Actor, and Restore Health Goodie, Flamethrower Goodie, Extra Life Goodie, and Fungus are all derived from Goodie. Goodie’s doSomething() method returns immediately if Goodie is dead at the beginning of the call. If Socrates overlaps with the Goodie, getOverlappingSocrates(Actor\* a) will return a pointer to Socrates in Student World. From there, doSomething() will call playSound(). Note that the sound for all the real goodies are the same, but Fungus has no sound. This is why I chose to make playSound() a public virtual void function in Goodie. That allowed Fungus’s playSound() function to override the original playSound() from Goodie and play nothing instead. doSomething() will also call performSpecialAction(Socrates\* socrates), which is a pure virtual function. Some examples of functionalities derived Goodie classes may implement in performSpecialAction(Socrates\* socrates) include increasing the game score, decreasing the game score, restoring Socrates’s hit points, decreasing Socrates’s hit points, and adding an extra life to the game. Then the Goodie will be setDead(). If Socrates does not overlap with the Goodie, the Goodie’s lifetime will decrease by 1 for every tick that passes. If the lifetime reaches 0, the Goodie is setDead().

Agent is an Abstract Base Class derived from Actor. Both Socrates and Bacterium are derived from this class. Different types of agents have different maximum numbers of hit points. For example, Socrates is at full health when it has 100 hit points whereas E. Coli only have 5. Each Agent can takeDamage(int damage). In addition to subtracting damage from m\_hp, this function also calls playHurt(), which is a constant, pure virtual void function, since Socrates, Salmonella, and E. Coli all have different sounds to play when they get hurt. If m\_hp is less than or equal, the Agent must be setDead(), and playDead(), also a constant, pure virtual void function, is called. All Agents return true for takeDamage(int damage). This helps to distinguish between Dirt piles and Agents, particularly Bacterium, in our Student World class. numHitPoints() is a public constant function that returns the number of hit points the agent currently has. restoreHealth() is a public function that restores the Agent’s health back to its maximum level.

Socrates is derived from Agent. It starts off with 5 flames and 20 sprays. numFlames() and numSprays() are public constant functions that return the number of flames and the number of sprays Socrates has, respectively. These functions are called in Student World when the program tries to display the status bar. addFlames() is a public function that increases the number of flames by 5, and this is called by performSpecialAction(Socrates\* socrates) when Socrates lands on a Flamethrower Goodie. When asked to doSomething(), Socrates checks if it is dead. If it is, it returns immediately; otherwise, it tries to read user input. If the player tries to move left or move right, Socrates sets a new direction and tries to moveAroundCircle(int angle), which is a private function that adjusts Socrates’s position based on his angle. If the player presses a space bar, Socrates shoots a Spray in the direction it is facing, and if the player presses enter, Socrates shoots 16 Flames in evenly spread out directions. If the user provides no valid key input and Socrates’s number of Sprays is below 20, doSomething() increments Socrates’s Spray count by 1 every tick.

Bacterium is derived from the Agent class as well. Salmonella and E. Coli are both derived from Bacterium. doSomething() checks if Bacterium isDead(). If it is, return immediately. If Bacterium is Aggressive Salmonella and Aggressive Salmonella is close enough to chase Socrates, doSomething() sets the boolean chaseSocrates to true. Then we check if Socrates overlaps with the Bacterium. If so, make Socrates take the damage. Each type of bacteria has different damage power, so getDamage(), which is a constant, pure virtual function must be implemented by each instantiable subclass to determine how much damage to give Socrates. If Socrates does not overlap with the Bacterium and the amount of food eaten by the Bacterium equals 3, the Bacterium divides into two at the appropriate position. Otherwise, if Socrates does not overlap with the Bacterium and the Bacterium overlaps with a Food item, eat the food (increment food count) and set the Food to dead. If chaseSocrates, determined above, is true, the function stops there. Otherwise, doSomething() calls doMore(), a pure virtual function that must be implemented by the subclasses of Bacterium. doMore() details the specific abilities of each type of Bacterium. For example, E. Coli’s doMore() function aggressively hunts down Socrates. Salmonella’s doMore() function depends on its movement plan and the location of nearby food.

E. Coli and Salmonella are both derived from Bacterium. Salmonella is an Abstract Base Class because its getDamage() and addBacterium(double newX, double newY) functions are pure virtual. Regular Salmonella and Aggressive Salmonella are derived from Salmonella.

**Unfinished Functionalities**

n/a

**Other Design Decisions and Assumptions**

The spec didn’t specify what to do if the player obtains negative points. I made the decision to let the player continue playing. Instead of displaying “000-50”, however, I changed it so that the score bar would display “ -50” in the case of a negative score.

**Testing**

I tested my Socrates class by dynamically allocating a new Socrates at position (0, VIEW\_HEIGHT/2) in the Student World’s init() function. I made sure that Socrates was facing in the right direction (0 degrees) when it was first created. Then I tried to move Socrates around in a circle, making sure that the angle changed appropriately and that the player never went outside the bounds of the petri dish.

I tested my Dirt class by dynamically allocating a couple Dirt piles in the Student World’s init() function. Once I was sure the Dirt piles were properly constructed, I proceeded to generate them at random positions in the petri dish. Afterwards, I added the Food and Pit classes, and I went back to make sure that the Dirt piles did not overlap with the Food items and the Pit piles but that it was okay for the Dirt piles to overlap with one another. Later, when I added the Projectiles class, I shot at the Dirt piles to make sure they were destroyed properly.

I tested my Food class by dynamically allocating a few Food items in the Student World’s init() function. Once I was sure the Food piles were properly constructed, I proceeded to generate them at random positions in the petri dish in accordance with the spec. Later, when I added the Bacterium class and its subclasses, I checked the overlapping between the Bacteria and the Food items were being noted and that the Food was being deconstructed after overlapping. Additionally, I made sure that when a Projectile destroyed a Bacterium, there was a 50% chance that the dead Bacterium would dissipate and a Food item would appear in its place.

I tested my Pit class by dynamically allocating a few Pits in the Student World’s init() function. Once I was certain the Pits were properly constructed, I proceeded to generate them at random positions in the petri dish in accordance with the spec. Later, when I added the Bacterium class and its subclasses, I checked that the Bacterium were being properly generated and moving away from the Pit and that the Pit self-destructed once all of its Bacterium were released.

I tested my Spray class by dynamically allocating a few Sprays in the Student World’s init() function. I started by just letting the Spray generate along the borders without moving every time I pressed the space key, but once that started working, I implemented the functions that allowed the Spray to doSomething() and move forward appropriately. Then I checked if it overlapped with Bacteria and Dirt Piles properly, and I double checked that it had no effect on all other objects.

I tested my Flame class by dynamically allocating a few Flames in the Student World’s init() function. I started by just letting the Flames generate along the circumference of the petri dish without moving forward whenever I pressed the enter key, but once that started working, I implemented the functions that allowed all 16 Flames to doSomething() and move in all directions. Then I checked if it overlapped with Bacteria and Dirt Piles properly, and I double checked that it had no effect on all other objects.

I tested my E. Coli class by first setting m\_nRegularSalmonella and m\_nAggressiveSalmonella in the Pit class to 0. This allowed for more efficient observations of the behaviors of E. Coli objects. I made sure that the E. Coli followed around the player when the player moved, even when the player moved all the way around the circumference of the circle. I also made sure that the E. Coli got blocked by Dirt piles by purposely moving Socrates to an area with high concentration of Dirt piles so that the E. Coli would be more likely to be stuck behind the Dirt piles. When hit by a Flame or a Spray, my E. Coli would disappear and sometimes turn into a food item (50% of the time).

I tested my Regular Salmonella class by first setting m\_nEColi and m\_nAggressiveSalmonella in the Pit class to 0. This allowed for more efficient observations of the behaviors of Regular Salmonella objects. I made sure that the Regular Salmonella tried to get to the nearest Food item when it was close enough to sense it. I also made sure that unlike the Aggressive Salmonella, the Regular Salmonella do not get blocked by Dirt piles; instead, they change directions and continue along another path. This means that Regular Salmonella should never get stuck. After running the program many times, I am fairly confident that the Regular Salmonella in my program do not get stuck. When hit by a Flame or a Spray, my Regular Salmonella would disappear and sometimes turn into a food item (50% of the time).

I tested my Aggressive Salmonella next by first setting m\_nEColi and m\_RegularSalmonella in the Pit class to 0. This allowed for more efficient observations for the behaviors of Aggressive Salmonella objects. I moved the player within 72 pixels of the Aggressive Salmonella, and I waited for the Aggressive Salmonella to come chase Socrates. I also made sure that the Aggressive Salmonella got blocked by Dirt piles by purposely moving Socrates to an area with high concentration of Dirt piles so that the Aggressive Salmonella would be more likely to be stuck behind the Dirt piles. When hit by a Flame or a Spray, my Aggressive Salmonella would disappear and sometimes turn into a food item (50% of the time).

I tested my Goodie class last. By running my program many times, I made sure that all the Goodies, including Fungus, only appeared on the circumference of the petri dish. During testing, I moved Socrates over each Goodie/Fungus and observed the effect it had on Socrates’s hit points and the game score, displayed in the status bar. I made sure that the status bar was updated regularly, including the game score, level, number of lives left, hit points, number of sprays left, and number of flames left.

For the Student World class, I tried my best to trace through the code and look for any memory leaks that may have occurred. A big problem I faced was deleting items from the list. I learned that list.erase() returns an iterator to the next item in the list, so if we try to increment the iterator by writing something like it++, an out-of-bounds error will occur as iterating beyond the end of a list is illegal.