Rahul Sheth

CS 32

March 2 2017

Report Project 3

**Part 1:**

Class Actor:

1. Virtual void doSomething() = 0;
   1. All of the actors in this project will be doing something but all of them will be doing something different hence it is pure virtual. Also, it is defined in actors so if we pass in a vector of Actor pointers we can use this.
2. Bool isDead();
   1. Checks if the actor is dead. This is better than say defining an m\_dead variable for every single class because it reduces risk for error. Returns the m\_dead variable of Actor class
3. Void setDead();
   1. Called when an actor is killed, either by hp or other reasons. This essentially just sets the m\_dead variable in Actor to true.
4. Void Stun();
   1. Called when an actor steps on a pool of water. This sets the m\_ticks private member variable up by two ticks. This signifies that the actor must sleep for an additional two ticks.
5. Int getTicks();
   1. Called to see whether an Actor is still sleeping or not. If this returns 0, then the Actor will proceed into doSomething. If not then it will reduce it’s ticks by 1.
6. Void reduceTicks();
   1. Called if getTicks is not 0, meaning that it is some positive number. This reduces the ticks by exactly one to show that the Actor has already slept another tick.
7. Virtual void updateHP(int hp) = 0;
   1. Updates the hp of an EnergyHolder actor by the value passed in. This was defined as purely virtual inside of the Actor class so if we are traversing an array of Actor pointers we will be able to access the update without dynamic casting to EnergyHolder.
8. Virtual int getColony() = 0;
   1. Gets the colony of an ant. Defined as a purely virtual function inside of Actor class so if we are traversing an array of Actor pointers, like we will in the Ant’s do something or in its bite, we will be able to easily tell if it’s of its colony or not.
9. Virtual bool isEnergyHolder() = 0;
   1. Returns whether the pointer refers to an energy holder or not. An energy holder is essentially anything other than a pebble, pool of water or poison. Returns true or false based on the condition and it is purely virtual because it must be defined within each class to return true or false.
10. Void checkIfStunned(int x, int y, StudentWorld\* sw);
    1. Checks whether an Actor has wandered onto a location where there is a pool of water as well. Only happens to actors that can be stunned. If there is a pool of water then we call the stun function.
11. Int checkFood(int x, int y, StudentWorld\* sw, int max);
    1. Checks if an Actor has wandered onto a location where there is food available. If so, it will update it’s hp by the units that are on the food and it will decrement the food’s hp as well, killing the food if necessary. If food units > max, return max.
12. Int checkPoison(int x, int y, StudentWorld\* sw);
    1. Checks if an Actor has wandered onto a location where there is poison. If so, it will update it’s HP by the amount the poisons decremented it by.
13. Bool getPoison();
    1. Returns whether an individual has just been poisoned. Ensures that that Actor does not get poisoned by the same poison object twice
14. Void setPoisonTrue();
    1. Set m\_poison to true if you encounter a poison object.
15. Void setPoisonFalse():
    1. Set m\_poison to false. This will occur if you are moving to a new location and therefore have already moved off the poison object.
16. Direction randomDirection():
    1. Resets the object into a random direction based on rand int of 1-4 assigned to NSEW. Useful for when distance goes to 0 and you need to reset.
17. Void move():
    1. Moves the Actor one position in the direction it is currently moving. If it is blocked by a pebble or is caused to move outside the bounds, it will have to set its distance to 0. Otherwise, it moves and resets the linked list

Class EnergyHolder:

1. Virtual bool isFood() = 0;
   1. This function will be redefined based on whether an object is or is not a food item. Purely virtual because all functions must have it and it makes it easier to identify food.
2. Bool isEnergyHolder();
   1. This function will return true because you are in the energy Holder class. Therefore, all inherited classes will do the same.
3. Int getUnits();
   1. Returns the units associated with that energy holder (Basically the health points, etc). Better to define in energyHolder so it doesn’t have to be redefined in all inherited classes.
4. Void updateHP(int hp)
   1. Update the health points of an object based on the value passed into it.
5. Virtual bool isInsect() = 0;
   1. This function will be redefined based on whether an object is or is not a insect object. Purely virtual because all functions must have it and it makes it easier to identify insects.
6. Virtual bool isPheromone() = 0;
   1. It is harder to distinguish between an anthill and pheromone based on the colony as they would return the same. Since we cannot parse int id’s, we will have to redefine a function inside each class based on whether it is a pheromone.
7. Virtual int getColony() = 0;
   1. This function will have to get redefined multiple times inside of the inherited classes of this class. You also need to know the colony to know the right ant to stun.
8. Void stun():
   1. This will increase the ticks it needs to sleep by 2.
9. Void setBitten():
   1. This will set if the insect has been bitten on the last turn

Class Poison:

1. Void doSomething():
   1. The poison object will do nothing in this instance. It’s just to stop it from being an ABC.
2. Bool isEnergyHolder():
   1. Returns false as Poison is not an energy holder.
3. Void updateHP(int hp):
   1. This itself will do nothing as well as poison is not an energy holder.
4. Int getColony:
   1. It will return not a colony as because it is not an ant.

Class Food:

1. Void doSomething():
   1. The food object does nothing in this instance because it doesn’t move or do anything.
2. Bool isPheromone():
   1. This will return false as it is not a pheromone. This is just a check to see if we have hit a pheromone.
3. Int getColony():
   1. Return not a colony as food is not a colony.
4. Bool isFood():
   1. Returns true as this is a food item. This will help if you are going thru a vector and you are trying to find a food item.
5. Bool isBitten():
   1. Returns false because food cannot be bitten
6. Bool isInsect():
   1. Returns false as food is not an insect.

Class PoolOfWater:

1. DoSomething():
   1. The pool of water will not do anything in this.
2. Bool isEnergyHolder():
   1. This will return false.
3. Bool isBitten():
   1. Pools of water cannot get bitten so it will return false.
4. Void updateHP(int hp):
   1. This will do nothing as well as pool of water has no hp.
5. Int getColony():
   1. This will also return Not A Colony as it is not an Ant.

Class Pebble:

1. DoSomething():
   1. The pebble will not do anything in this.
2. Bool isEnergyHolder():
   1. This will return false.
3. Bool isBitten():
   1. pebbles cannot get bitten so it will return false.
4. Void updateHP(int hp):
   1. This will do nothing as well as pebble has no hp.
5. Int getColony():
   1. This will also return Not A Colony as it is not an Ant.

Class babyGrasshopper:

1. Void kill():
   1. This is a helper function that will be called when a baby grasshopper is dead. This will cause a 100 unit food to be dropped and for setDead to be called.
2. Bool isFood():
   1. This will return false as it is an insect.
3. Int getColony():
   1. This will return not a colony as baby grasshoppers are not ants.
4. Bool isPheromone():
   1. This returns false as it is an insect.
5. Void resetDirection:
   1. This will be called when the distance equals 0. The direction will be randomized and the distance will be too and it will be returned.
6. Void doSomething():
   1. This will check if the ticks have ran out and then increment by 2. Then it will move based on the direction if it can (if it doesn’t hit a pebble). If the ticks have not ran out then it will decrement the ticks.
7. Bool isInsect():
   1. Returns true. We will be able to identify insects better.

Class AntHill:

1. Bool isFood():
   1. This will return false as anthill is not food. This is useful in identifying food at a position.
2. Void doSomething():
   1. This will reduce hit points by 1 and check if it was dead. Then it would check the food and then it would add ants if the anthill hp is greater than or equal to 2000.
3. Bool isPheromone:
   1. Returns false
4. Bool isInsect():
   1. This will return false as anthill is not an insect. Makes it easier to identify insects
5. Int getColony():
   1. Returns the colony that the ant represents.
6. Void addAnAnt():
   1. Adds another ant onto the square. Helper function for the call in doSomething.

Class adultGrasshopper:

1. Void doSomething():
   1. This will do all the functionality that the original doSomething from baby grasshopper did in addition to jumping across a radius 10 circle and biting others.
2. checkIfSolidLocation(int x, int y):
   1. This will adjust the vector of coordinates will all the coordinates that are in the radius 10 circle so doSomething can choose from them.
3. Void biteOthers(int x, int y):
   1. It will bite the enemy objects that are on the same location as the current object we are analyzing.

Class Ant:

1. Void doSomething():
   1. This will first run through all the normal checks of an insect doSomething. Then it will go into the interpreter commands and find the proper one to analyze.
2. Bool isFood():
   1. Returns false because it is an insect.
3. Void setBlockedTrue:
   1. Sets whether the ant was previously blocked on movement by a pebble. This will help it with other commands, especially with the if command.
4. Bool isPheromone:
   1. Returns false as it is an ant.
5. Bool getBlocked:
   1. This will check whether an ant object was blocked on the last turn or not. This is to help it analyze whether
6. Bool checkIfCommand():
   1. Depending on the values that were fed into the if command line, it will either: check if the last random number was 0, if am carrying food, if I am hungry, if I am standing with an enemy, If I am standing with food, if I am standing on my anthill, If I smell a pheromone, if I smell danger, If I was bit, and if I was blocked from moving. Then it will change the ic based on that.
7. Bool isInsect:
   1. Returns true as ants are in fact insects.
8. Void rotateCounterClockwise:
   1. This will rotate your direction counterclockwise.
9. Void rotateclockwise:
   1. This will rotate your direction clockwise.
10. Void dropSomeFood:
    1. Helper function if the command is called to drop all of the food of the ant.
11. Void pickUpSomeFood:
    1. Helper function if the command is called to pick up food on the current square. First checks for food then checks it against the maxes.
12. Int getColony:
    1. Returns the colony of the ant.
13. Void eatSomeFood:
    1. Eats 100 units of the food or all if less than 100 units of food is available.
14. Bool checkEnemies():
    1. Checks if there are enemies to bite on the current square.
15. Void setBlockedFalse:
    1. This will be called if you moved to a new square successfully, meaning that you were not blocked. This will be useful in the if commands.
16. Void bite:
    1. This will bite all of the ants that are not of the current ant’s colony and all the grass hoppers.

Class Pheromone:

1. Void doSomething():
   1. Simply reduces the hp on every tick and kills the pheromone if it’s hp is 0.
2. Bool isPheromone:
   1. Returns true as it is a pheromone.
3. Int getColony:
   1. Returns the colony of the ant that the pheromone is associated with.
4. Bool isFood:
   1. Returns false.
5. Bool isInsect:
   1. Returns false.

**Design Issues:**

I assumed that we could stun inside of the actor classes as opposed to the pool of water and poison classes. This was a design solution that was backed up by Prof. Smallberg in the FAQ’s.

**Problems:**

I Implemented the ticker text on the top of the file at the last minute and did not know how to test it on terminal so that is a variable in my program that may be incorrect.

**Classes**

Class Actor:  
 To test the Actor class, I would simply add new actors into the program and ensure that they don’t return any nullptr exception and correctly work through all the functions. Since most of my functions in the Actor class are generalized across all of the classes in the function, that provided a good foundation for test as I could see the behavior of different classes which have different requirements with them. When I ran it against the USC Ants program, it ran fairly similarly to the simulation so I assumed it was fine.

Class EnergyHolder:

There really isn’t much to test in EnergyHolder as it mostly served as a middle man in dynamic casting and a base class for some inheritance in this project. Therefore, I ran the functions that were meant to be generalized for the entire program, such as updateHP and stun. I saw that the bugs would in fact get stunned at the proper times and the baby grasshoppers would turn into adult grasshoppers, proving that the update HP function was working. I believe this was fairly thorough as the material to cover in this class was minimal.

Class Poison:

Poison was a class that could not be judged by its own functionality rather than the behavior of other objects as they interacted with poison. Most of the time, I would put in certain print statements to print when an object landed on a poison object and would also print out the hp before and after. As the hp was modified with what the spec entailed, I assumed the poison class was correct.

Class Food:

Food was also a class that could be judged by its own functionality so I put in separate print statements to see when an object would hit a food object when working through. Then I would print out the units on the object and then checking the transfer, carefully taking note of the individual maximums. As objects would pick up food normally, I could see no real errors with my implementation of food.

Class poolOfWater:

PoolOfWater was yet again another class that could not be judged by its own functionality and was judged based on the same mechanism that I judged poison. As I saw that it interacted in a way that was specified in the spec, I assumed that poolOfWater was also correct.

Class Pebble:

The pebble class was truly judged in the move functions of each of the objects. I would carefully take note of the grasshoppers and see when they would be blocked by a pebble, whether it was walking or during jumping. In addition, I would use print statements to assess the value of m\_blocked in the ant class to make sure that it was working properly with the pebbles.

Class babyGrasshopper:

The babyGrasshopper class was a fairly tough class to test for. The first thing that I looked for was whether the grasshoppers would die and disappear when the hp hits 0 by using print statements. In addition, I would take into account whenever a new adult grasshopper was created and I would make the sure the units was greater than 1600 hp. Since this all worked fine, I assumed that baby grasshopper was working fine.

Class Anthill:

The only thing I had to judge for anthill was whether it would properly create the ants. This was the case as it would make the proper amount of ants in the beginning. Since I saw no issue with this, I assumed everything was fine.

Class Ant:

I believe that my testing for ant was fairly insufficient. I simply ran the simulation for Professor Smallberg’s test simulation against my own and saw that the ants were working in the proper direction. I didn’t really know how to twist around the code or the field to make the ants move in a particular way and judge their behavior.

Class Pheromone:

My testing for pheromone consisted of seeing whether it was properly created and incremented whenever it encountered an ant of the same class as it. I believe that this was the case and therefore believe that it was fine.

Class StudentWorld:

My StudentWorld implementation was intertwined with the implementation of all of my Actors. Additionally, I would test the new location of the objects versus their positions in the linked lists to ensure the location was updating. Aside from that, I didn’t see any major problems with StudentWorld that might mess up the code.