Cody Hubbard - 004843389 - Proj 3 Report

1. CLASS DESCRIPTIONS:

**StudentWorld Class:**

canMoveTo is the function responsible for checking for almost all valid movement within the program. It is public because actors themselves cannot tell what other actors are where to this function was used to

addActor is used to place any actors created by other actors (new ants/pheromones/food) onto the field. It is public because all actors have no information about the field themselves so they call on this to handle the event for them.

getEdibleAt checks the square at position X, Y for any type of food and returns the pointer to that food item. This is public because there can only be one food object on any given square so this lets actors know if one already exists or not.

getPheremoneAt has the exact same purpose and functionality as “getEdibleAt” except that it is exclusively called on by ants because they are the only actors who use pheromones.

isEnemyAt checks if there are any enemy actors on a given square and returns a bool. This is needed because actors have no information about the rest of the field so if they are trying to estimate risk or act aggressively this function is used.

isDangerAt is much like “isEnemyAt” except return true if there are actors like poison or water at the given x,y location, actors deemed “dangerous”.

biteEnemyAt is an event handling function. Any time an actor wants to act aggressively this function is used to apply the action to the other actor(s) on the square

poisonAllPoisonableAt is an event handling function. Any time a Poison Actor wants to poison the actors on its square this function applies the action.

stunAllPoisonableAt is an event handling function. Any time a WaterPool Actor wants to stun the actors on its square this function applies the action.

increaseScore is used when an AntHill actor creates a new ant. This is necessary because the StudentWorld class tracks the scores of the different colonies so the AntHill cannot access this information. Therefore this function updates scoring information for the AntHill.

Actor Class:

doSomething is the backbone of actor events. Each tick this function asks its respective actor to perform its defined actions. This is in the Actor base class as well as is pure virtual because all actor classes need to be asked do things but many intermediate derived classes should not be able to be instanced. Also almost every class derived from object with do different things.

isDead is return if the actor is dead, in which case multiple action can be taken but most often the actor will simply be deleted function needed to be virtual.. Almost all actors will be asked if they are dead so this function needed to be in the basemost class.

blocksMovement tells if a specific actor prevents movement onto a spot or not. This function is virtual because some actors block movements while others do not, and all actors need to be asked if they block movement so this function is in the basemost class.

getBitten is the function that handles how different actors respond to in the event they are bitten. Some lose life points while others do nothing, and some bit back. Thus this function need to be virtual. Many actor can be bitten so this function in the basemost class.

getPoisoned is the function that handles how different actors respond to in the event they are poisoned. Some lose life points while others do nothing, thus this function need to be virtual. Also many actors will be asked how they handle poison so this function in the basemost class.

getStunned is the function that handles how different actors respond to in the event they are Stunned by a pool of water. Some sleep for several ticks while others do nothing thus this function need to be virtual. . Several derived classes have the ability to be stunned in thus this function need to be in a base class.

isEdible is used to tell is any actor can be eaten or not. Almost every actor will be asked if it can be eaten so this function needed to be in a base class, but only one actor, food, actually has this property so it needed to be virtual.

isPheremore is the same as inEdible except the only derived class returning true is pheremone.

isEnemy is used to tell if different actors are hostile towards each other. Almost every actor needs to be asked if it is an enemy so this needed to be in a base class. Additionally some actors are always hostile, some are never hostile and others are conditionally hostile so this needed to be virtual.

isDangerous is akin to isEnemy except also includes things like poison and water in its list of things deemed dangerous.

getWorld is the function that tells any derived actor what its studentworld object is. This is needed becasue every class derived from Actor will have a world and they need access to it to act within the simulation.

isMyAntHill is the function that will tell an Ant object if any actor is their anthill. All actors will be asked if they are an anthill so this needed to be in the base class, but only Anthill derived classes would return true so it needed to be virtual.

**Pebble Class:**

doSomething was needed here because objects from class perform different actions than other derived classes. In this case, pebbles do nothing.

blocksMovement is here because this derived class is actually the only derived class that redefines this behavior. Pebbles are used to limit the playing field and add obstacles so they must block movement.

**EnergyHolder Class**

isDead is redefined here because this is the second level derived class where objects have life points and can actually die. isDead checks these life points and if the object has died calls kill me.

killMe is used to an actor's state to dead and then perform any actions that accompany that like leave food behind and the like, Energy Holder need to die so that's why it is public here.

updateEnergy is used to modify any class which is derived from energy’s energy value. Many classes use energy so it is public.

addFood first checks if there is already food on a square, if so it will add the given amount to it, otherwise it will create a new food object of that amount. Food is actually a specific use of energy and several derived classes interact with food so it is public here.

pickupFood is used to take energy from food object and place it in a specific variable, reducing the food energy in the process. Several classes derived from energy do this so it is public here.

pickupAndEatFood calls the pickUpfood from above and eats it in one go, this is public because a derived class need to do this,

becomesFoodUpodDeath is a specific property several classes derived food need to check for. All classes derived from food can die so they will be asked this so it is public, but only some actually become food so it is virtual.

updateHeldFood is used to modify the heldfood variable. This is public here because derived classes cannot modify this variable themselves.

getHeldFood gives the value held in the heldFood variable. This is public here because derived classes cannot access this information otherwise.

**Class Food:**

doSomething is a defined here because it was pure virtual, and food objects need to exist in the field, even though food does nothing. It is public because food objects will still be asked

To do things by the StudentWorld.

isEdible is overloaded here becasue food ois the only class derived from actor that will actually have this property.

**Class AntHill:**

doSomething is a defined here because it was pure virtual, and AntHill objects need to exist in the field. Anthills will try to make ants by calling a private function and lose one point of energy each tick. It is public because food objects will still be asked to do things by the StudentWorld.

isMyAntHill is overloaded here because this is the only class derived from actor that will actually change how this property is handled. It is public because the student world will ask this for ants.

**Pheremone Class:**

doSomething is a defined here because it was pure virtual, and Pheromone objects need to exist in the field. Each tick pheromones will lose a point of energy. This is public because pheromones will be asked to do things by the student world.

isPheremone is overloaded here because pheromones will be asked if they belong to a certain colony and need to return something other than false for this unlike other classes.

**TirggerableActor Class:**

isDangerous is the only function defined here because this is the only derived class which changed this property to true. It is public because they will be asked.

**WaterPool Class:**

doSomething is a defined here because it was pure virtual, and WaterPool objects need to exist in the field. This is public because WaterPools will be asked to do things by the student world.

**Poison Class:**

doSomething is a defined here because it was pure virtual, and Poison objects need to exist in the field. Each tick pheromones will lose a point of energy. This is public because WaterPools will be asked to do things by the student world.

**InsectClass:**

doSomething is only defined here because I wanted to put together common functionality several derived classes shared. The functionality could not be fully implemented so this just has a simple function, it decrements energy by 1.

getBitten is implemented here because classes derived from insects are the first classes that can be bitten and they share some common functionality. Different insects respond differently to being bitten so it is important that it is virtual.

getPosisned is redefined here because insects will take damage when poisoned. Some insects get poisoned differently so it is important that it is virtual

Same for getstunned as getpoisoned.

isEnemy is redefined here because different insects will belong to different colonies or even none at all, this is why it is important for this function to be virtual.

becomesFoodUponDeath is defined here because insects are the first derived classes which will leave corpses which are food.

All of the above classes for Insect are public becasue the student world will call upon them.

getXYInFrontOfMe is an important function for movement and all insects will move within the field.

It is public because StudentWorld will call is as well as the fact that almost all classes derived from insect need to use this function.

UpdateSleepTicks changes the amount of ticks an object derived from insect will sleep. This is public because insects can get stunned or rest and need this common functionality.

IsAlseep tells if a class derived from insect has sleep ticks great than zero, this is public because insect derived classes need to tell when it is time to wake up and they wouldn't have access to this information without this function.

randDirection will output a random direction by using the global randInt function. This is needed, and public, because there are several cases in which random directions are required so consolidating this common functionality for all classes derived from insect is important.

randAttack will choose and bite a random enemy at a square if one exists. Several classes derived from insects will want to do this so it is important for it to be defined and public here. Some classes derived from insect will bite differently so it is important for it to be virtual.

**Grashopper Class:**

getDist will return the distance a grasshopper is trying to walk. This is public because any class derived from grasshopper will want to walk a distance at any time and they wouldnt have access to this information otherwise.

updateDist changes the distance a grasshopper is trying to walk. Any class derived from grasshopper may want to do this and wouldn't be able to unless this function was public, so it is.

newPath gives any class derived from grasshopper a new direction and distance. This is usually called when a grasshopper derived object has been stopped from moving which may happen often and by different derived classes so I wanted to consolidate this functionality, thus this function is public.

tryEat will attempt to eat any food on the grasshoppers current square. This is public here because both classes derived from grasshopper will want to eat and I wanted to consolidate this functionality.

tryWalk will make a grasshopper try to move a square forward in its current direction, and reset what it wants to do if it could not. Both types of grasshopper will do this in almost the same way so having this common functionality publicly available here made sense.

**Ant Class:**

doSomething is defined here because it was pure virtual and Ant objects need to exist in the field. The exact procedure is heavily detailed in the spec, but what should be known is that i've almost entirely delegated the actions to the public function in the grasshopper class this is derived from.

Getbitten is defined here because ants need to keep track of when they have been bitten, unlike other insects. So this functionality is implemented in this redefinition.

isEnemy is redefined here because ants belong to a colony and they must be able to compare their colony to other Insect derived objects to determine if they are friendly or hostile. This functionality is implemented here.

moveForwardIfPossible will attempt to move an Ant forward one step in its current direction. If it could not it will keep track of this as well for use elsewhere.

These ant functions are public because the student world will call upon them, ant has no derived classes.

**BabyGrassHopper class**

doSomething is defined here because it was pure virtual and babygrasshopper objects need to exist in the field. The exact procedure is heavily detailed in the spec, but what should be known is that i've almost entirely delegated the actions to the public function in the grasshopper class this is derived from.

**AdultGrassHopper Class:**

doSomething is defined here because it was pure virtual and adultgrasshopper objects need to exist in the field. The exact procedure is heavily detailed in the spec, but what should be known is that i've almost entirely delegated the actions to the public function in the grasshopper class this is derived from as well as several complex private functions

getPoisoned and getStunned are defined here because adultgrasshoppers are immune to these things so these functions needed to be overloaded here.

2. BUGS/ UNIMPLEMENTED FUNCTIONALITY

I could not implement the adult grasshopper jumping in a circle correctly so instead it randomly jumps to any point in the 11x11 square centered at the grasshoppers position (i.e. 5 spaces up down left and right, like the circles radius).

There could be bugs in my ant compiler, it was not as thoroughly tested as I would have liked.

Using the class hierarchy provided it can be inferred that all Insect objects need to start with a

random direction they are facing, which will ultimately be passed to the GraphObject. To accomplish this i had to use a function in my initializer list as follows. For reasons I do not understand.

Here's the code,

Insect::Insect(StudentWorld\* world, int startX, int startY, int energy, int imageID)

:EnergyHolder(world, startX, startY, **randDirection()**, energy, imageID, 1),

sleepTicks(0), wasStunned(false)

{}

Where the bolded underlined part is a function of the Insect class which returns a random direction. For reasons even further beyond my current understanding this gives me the following error in g++,

Actor.cpp:254:52: runtime error: member call on address 0x60b00000ae30 which does not point to an object of type 'Insect'

However the program still outputs “\*\*\*\*\*\* SUCCESS! Passed the sanity check. \*\*\*\*\*\*” so i’m leaving it.“

3. DESIGN DECISIONS

dynamic\_cast was used sparingly to access member functions of classes derived from actor class. I the cases in which it was used the pointer was already known to be what it was being casted as though methods other than dynamic\_cast.

I wanted to simply call on the inherited doSomethigthing() functions for classes that shared functionality but i couldn't because I couldn't find a work around for the return statements which the common code shares.

I do not know how atoi or .c\_str() work but they were need to convert to a string to an int.

4 TESTS

Student World Class:

StudentWorld::Init was tested by incrementally increasing the objects of the field it initialized. I checked that each Actor was being place properly by using the debugger and looking into the 3 dimensional vector it uses as a field data structure as well as visual confirmation by running the Bugs! Program and seeing that my output matches the field text document used by the program.

Later upon implementing the anthill Actor i made sure my compiler was working correctly by compiling incrementally more ant colonies up to the max, and looking for each’s spawned hill in the field.

StudentWorld::Move was tested visually with the program output, I then looked in the debugger to be sure field actors were being place in the correct x,y places in the m\_field data member. I also used the debugger to make sure that my checkPosition function was putting actors who moved places during their tick into the correct place in the field data structure. Lastly I used the debugger to ensure dead actors were being erased properly.

Pebble Class:

This class was thoroughly tested during part 1 of the assignment. I used babygrashoppers to ensure movement was being correctly blocked.

Waterpool Class:

A simple grasshopper was used to test the water pool class. The grasshopper only moved and could be stunned. I restricted the StudentWorld init method to only initialize rocks and pools in the field for this test. I used the visual output the program provides to be sure that when a grasshopper stepped on a pool of water it correctly rested for 4 ticks. I also check a couple grasshoppers being stunned in in the debugger as well, making sure their sleeptimers were being incremented and decremented correctly as well as to make sure grasshoppers were not being stunned multiple times on the same water pool.

Due to the way I setup my class inheritance I was effectively stunning an Insect, not a grasshopper, so the test ran will hold for any class which is derived from an insect, eliminating the need to test ants/other insects.

Poison Class:

A simple grasshopper was used to test the poison class. The grasshopper could only move and be poisoned. I used a restricted student world int() method which only initialized grasshoppers and poison. I then ran the program and used the visual output to be sure some grasshoppers were dying earlier than others due to the poison. I confirmed the visual cues with the debugger, checking that a grasshopper energy was being correctly reduced when it stepped on a poison block.

Due to the way I setup my class inheritance I was effectively poisoning an Insect, not a grasshopper, so the test ran will hold for any class which is derived from an insect, eliminating the need to test ants/other insects.

Food Class:

The food class and the eating of food was in three steps. I first made sure that food actors had their energy reduced when eaten, I did this using a simple baby grasshopper which only moved and ate, checking with the debugger that energy values were being correctly adjusted when food a grasshopper shared a square with food. I then disabled the eating functionality and made sure that dying grasshoppers correctly left food behind when they died.

Grasshopper Class:

Grasshopper class was tested incrementally, I first made sure that its movement was working correctly as this was crucial to testing almost all of my other classes. I tested movement against a field of only rocks and used the visual output to be sure grasshoppers were not traveling onto rocks, and that they had pseudo random movement. I then tested that grasshoppers could be stunned/poisoned correctly at the same time I was testing the waterpool/ poison classes. I next tested the grasshoppers eating and dying mechanics as they both involved the food class, I first tested eating food by restricting the field to only food, rocks and grasshoppers, and checking with the debugger that energy values were being correctly adjusted when food was eaten. I then made sure death was working by removing everything except grasshoppers and rocks and checking that new food objects were being created upon tick 500. Lastly I tested if molting by spawning baby grasshoppers with 1600 energy and checking that they were dying and adult grasshoppers as well as food were spawning on the first tick.

As you'll see I used the grasshopper class to test most basic actor/insect interactions so almost all tests performed on the grasshopper are akin to testing for almost all classes derived from insect

.

AdultGrasshopper Class:

The Adult grasshopper class is derived from the Insect class and thus most functionality not explicitly tested here was already tested by the baby grasshopper class. The Adult Grasshoppers bite function was tested by spawning only rocks and baby grasshoppers with 1601 energy so they immediately turned into adults and removed the ⅓ chance so they would always bite. Then using the visual aid to tell when a grasshopper was supposed to bite another as well as a breakpoint in the debugger to make sure the bite was carried out as expected. This test effectively bites an insect, not another grasshopper so this test should work for all insects, as well as Adult Grasshoppers getbitten functionality. Jump was tested visually by looking at the field and in the debugger by making sure they were changing position correctly.

AntHill Class:

The anthill class was first tested by compiling ant programs and initializing their respective hills upto 4 colonies. To Test ant creation and correct scorekeeping I limited the field to only anthills and rocks and gave the ants basic move forward functionality and used the debugger/visual field to ensure 5 ants were made in the first 5 ticks of the simulation. I tested the Adult Grasshoppers immunity to poison and water by putting a breakpoint within the overwritten methods to make sure they trigger appropriately.

Due to my use of inheritance functions like consuming food or dying have already been tested by my basic grasshopper class.

Ant Class:

Due to my use of inheritance The Ant class only needed functions that it overrode or newly implemented to be tested because all other functions were tested by the basic grasshopper class. The biggest of these function to be tested was the ant interpreter method, but I was not able to test it as thoroughly as one would want. This was incrementally tested by changing the logic of the single ant colony being spawned each time a new command of the interpreter was implemented, and using the debugger to make sure the specific commands were being executed correctly.

Pheromone Class:

The peremone class was the last to be tested as it relied upon the ant class to have any function. I tested it by spawning ants that only walked, released pheromones, and “smelled” pheromones and then used the debugger to ensure that the pheromones were dying, dissipating, and being interpreted correctly.