Computer Science 32 Project 3 Report by Roy Lin <704-767-891>

1.) Class Hierarchy :

Actor

            Pool

            Poison

            EnergyHolder

                        AntHill

                        Ant

                        GrassHopper

                                    AdultGrassHopper

                                    BabyGrassHopper

ACTOR\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

StudentWorld\* getArena();

{

I chose this function so that I could get access to the StudentWorld pointer that made the Actor. This is for accessing functions in StudentWorld

}

virtual void doSomething() = 0;

{

I chose to define a pure virtual version of the doSomething() function in my base Actor class because all actors in Bugs! are able to doSomething(), and each type of actor does Something in a different way.

}

bool isDead();

{

bool to check if the actor is dead or not

This is for deallocating the dynamic memory if the actors died.

}

void setDead();

{

void that just sets the actor's status to dead

}

void set\_moved();

{

void to set the actor's status to moved

}

void set\_unmoved();

{

void to set the actor's status to unmoved

}

bool getMovedStatus()

{

returns the current status if the actor moved already or not. The main reason for this is so that when the data structure updates, it will not move the actor multiple times in one tick

}

Thus, I have 3 private member variables in this class: StudentWorld\* m\_swp;bool m\_dead;bool m\_moved; which help me implement my functions.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

POISON

virtual void doSomething();

{

virtual since we defined doSomething in the parent class; The poison do something calls a function in StudentWorld that poisons all the insects on the current square.

}

No members variables in Poison class since the poison just needs to update the actors inside of StudentWorld

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

POOL

virtual void doSomething();

{

virtual since we defined doSomething in the parent class; The pool do something calls a function in StudentWorld that stuns all the insects on the current square for 2 more ticks.

}

No member variables in Pool class since the Pool just needs to update the actors inside of StudentWorld

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PEBBLE

virtual void doSomething();

{

virtual since we defined doSomething in the parent class; The Pebble doSomething does nothing since it's a pebble.

}

No member variables in Pebble class since Pebble just needs to update the actors inside of StudentWorld

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FOOD

virtual void doSomething();

{

virtual since we defined doSomething in the parent class; The Food doSomething does nothing since it is a piece of food.

}

void addCarcassFood(int fmount);

{

This function will add to the food amount of food any amout that is passed in. This function is called by the StudentWorld class so that the StudentWorld won't need to dynamically allocate new foods, only need to add to the amount

}

int getFoodLeft();

{

returns the amount of food at the square currently

}

void getRidFood(int num);

{

gets rid of the amount of food at the square and then checks if the food is less than 0 - In which case the food would be set to dead

}

The member variables in the Food class is only m\_foodAmountLeft; which is only the amount of food at the square to be checked and updated

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

EnergyHolder

virtual void doSomething() = 0;

{

I chose to define a pure virtual version of the doSomething() function in my base EnergyHolder class because all Energyholders in Bugs! are able to doSomething(), and each type of EnergyHolder does Something in a different way.

}

virtual bool eat(int x, int y) = 0;

{

I chose to define a pure virtual version of the eat() function in my base EnergyHolder class because all Energyholders in Bugs! are able to eat(), and each type of EnergyHolder eat in a different way.

}

int getHP();

{

returns the amount of hit points that the energy holder currently posses

}

void addHP(int num);

{

adds number of hit points to the hit point counter based on the integer inputted into the function

}

void deHP();

{

decrements the hit points by one point each time the function is called

}

The member variable in the EnergyHolder class is only m\_hp which each EnergyHolder has. They can update or check the hp that the energyHolder has

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Pheromones

virtual void doSomething();

{

virtual since we defined doSomething in the parent class. The pheromone doSomething decrements the hit points. Then it checks if the hp is less than 0 - then it sets it to dead.

}

virtual bool eat(int x, int y);

{

virtual since we defined eat in the parent class. The Pheromone eat does nothing.

}

int getPherType();

{

returns what type of pheromone it is - whether it is part of Ant1, Ant2, Ant3, or Ant4

}

void addPheromones();

{

This functions add to the Pheromone hp everysingle time it gets called.

PSEUDOCODE:

amount = 256

if amount added to hp > 768

add hp up to 768

else

just add 256

}

The pheromone member variable is m\_pherType which is an identifier that tells what type of pheromone it is - whether it is part of Ant1, Ant2, Ant3, or Ant4

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Insect

virtual void doSomething() = 0;

{

I chose to define a pure virtual version of the doSomething() function in my base Insect class because all Insect in Bugs! are able to doSomething(), and each type of Insect does Something in a different way.

}

virtual void move() = 0;

{

I chose to define a pure virtual version of the move() function in my base Insect class because all Insect in Bugs! are able to move(), and each type of Insect move in a different way.

}

//Insect Specialties

virtual bool bite(int enemy);

{

I chose to define a virtual version of the bite() function in my base Insect class because all Insect in Bugs! are able to bite(), and some insects bite in a different way

}

virtual bool eat(int x, int y) = 0;

{

I chose to define a pure virtual version of the eat() function in my base Insect class because all Insect in Bugs! are able to move(), and each type of Insect move in a different way.

}

int getStunStatus();

{

returns the current amount of stuns that the insect currently has;

}

bool stunnedState();

{

returns the current state of the stun status

}

void deStunned();

{

unstuns the Insect by one

}

void Stunned();

{

adds two to the currrent amount of stuns

}

void resetStun();

{

resets the current amount of stuns to two

}

void statusStun();

{

STATUS stuns the Insect (for pool and poison) - status modifiers

}

void deStatusStun();

{

un STATUS stuns the Insect (for pool and poison) - status modifiers

}

bool isStatusStunned();

{

returns the STATUS STUN of the Inset (for pool and poison) - status modifiers

}

int getEnemyNumber();

{

returns the enemyNumber identifier of the current Insect

}

The Insect class has 3 private member variables int m\_enemyNumber; which stores the enemy number of the Insect bool m\_stunnedByStatus; which stores if the Insect is status stunned or not int m\_stunCounter; which stores how many times the insect is stunned.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

AntHill

virtual void doSomething();

{

virtual because doSomething is virtual in the EnergyHolder and Actor Base classes. The Anthill do Something decrements the anthill hp. If the the hp falls below zero, set to dead. It will then try to eat any food on the current square. It checks if the hp is above 2000 to which it will allocate a new ant and decrement its own hit points by 1500.

}

virtual bool eat(int x, int y);

{

Eat function for the anthill function where if the current space has food, it will try to eat up to 10,000 units to add to its own hit points

}

The anthill class has m\_colony which helps determine what colony ants the anthill will produce and Compiler\* m\_compiler which passes in the compiler arguments to the ant that it creates;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Ant

virtual void doSomething();

{

virtual because doSomething is virtual in the EnergyHolder and Actor Base classes. The Ant do Something decrements the ant hp. If the hp falls below zero, set to dead and make food appear on the map. Then you have to check if the the ant is currently stunned to which case you would unstun it and return. Then reset the stun counter after it executes a movement. The ant would interpret one to ten commands (to be pseudocoded later). it will then update the instruction counter. IF the command changes the state of the simulation, return right away. Runs commands until it reaches the condition of changing state or it runs a total of ten commands

}

virtual void move();

{

virtual because move is virtual in the EnergyHolder and Actor Base classes. functions that moves the ant forward in one space in the direction it is facing

}

int getFood();

{

returns the amount of food that the ant is carrying

}

virtual bool eat(int x, int y);

{

virtual because eat is virtual in EnergyHolder. The Ant accepts to eat food at the ants current location

}

bool Interpreter();

{

Interprets the .bug file to get the commands

This is a trivial function because it just moves the ant around and makes the ant interact with the ant's surroundings

PSEUDOCODE IN THE SPEC

}

int getType();

{

returns the type of the ant

}

int generateRandomNumberUpTo(std::string operand);

{

returns a random number from 0 to the operand turned into an int

}

int convertToInteger(std::string operand);

{

convert the operand to an integer to which it returns

}

bool conditionTriggered(Compiler::Command cmd);

{

Called from the interpreter when it meets an "IF COMMAND" to which this function sees if the condition got met. CONDITIONS ARE TRIVIAL

}

void setBit();

{

sets the bit bool to true inside the Ant class for checking in the interpreter

}

void deBit();

{

sets the bit bool to false inside the Ant class for checking in the interpreter

}

bool pickUpFood();

{

Checks if there is food at the current space and attempts to pick up 400 and adds it to the food amount that the ant is carrying and checks if its over 1800

}

bool hasDangerInFront(int x, int y, int colony);

{

checks the direction that the ant is currently facing and analyzes the position in front of the ant. Sees if there are enemy ants, any type of grasshopper, poison, or pool are in front of it.

}

bool hasPheromoneInFront(int x, int y);

{

checks the direction that the ant is currently facing and analyzes the position in front of the ant. Sees if there are pheromones of its own type in front of the ant in the direction that the ant is currently facing

}

The Ant member variables for ant are as follows Compiler\* m\_instructions; to pass in the instructions to the ant for the interpreter to follow

AntHill\* m\_home; to pass in what the home of the ant is to get the location of the anthill

bool m\_wasBit; stores if the ant was bit in the last turn

bool m\_wasBlocked; stores if the ant was blocked in the last turn

int m\_type; stores what colony the ant resides in

int m\_foodAmount; stores how much food the ant is carrying

int m\_lastRandomNumber; stores the last random number that the interpreter generated

int m\_instructorCounter; stores where the instructorCounter is

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

GrassHopper

virtual void doSomething() = 0;

{

I chose to define a pure virtual version of the doSomething() function in my base GrassHopper class because all GrassHoppers in Bugs! are able to doSomething(), and each type of GrassHopper does Something in a different way.

}

virtual void move();

{

virtual because base class Insect has virtual move functions. All grasshoppers move in the same way

}

virtual bool eat(int x, int y);

{

virtual because base class Insect has virtual eat functions. All grasshoppers eat the same way. It trues to eat 200 units of food at the current position.

}

int getCurrentDistance();

{

returns the amount of the distance that the grasshopper currently moved before the distance reset for changing direction

}

void loseDistance();

{

decrements the currentDistance that the ant has to move before it changes direction due to hitting a pebble or running out of the current distance to move in the current direction

}

void resetDistance();

{

sets the current distance to one

}

void randomDistance();

{

comes up with a rnadom number from one to ten to set the current distance to for the ant to move

}

The GrassHopper private member variable is m\_currentDistance which stores the distance that the grasshopper type still has to move before the distance is reset.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

BabyGrassHopper

virtual void doSomething();

{

virtual becuase the base class Insect and GrassHopperis virtual. The baby grasshopper loses one hit point. If the hp reaches less than 0 then add 100 units of food to the field and set itself to dead. Then it must check if the babygrasshopper is stunned or StatusStunned then return. Check if hitpoints are above sixteenhundred so that you can make an adult grassshopper, add 100 units of food to the field, and set itself to dead. Then, it will try to eat at the current square from the Insect's eat function. If the babygrasshopper did eat, there is a 50% chance that the grasshopper would be stunned right after eating. The babygrasshopper then attempts to move in the current direction until the distance runs out of the babygrasshopper runs into a pebble.

}

BabyGrassHopper does not have any private member variables since everything it needs is from the Insect class

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Adult GrassHopper

virtual void doSomething();

{

virtual because the base class Insect and GrassHopper is virtual. The adult grasshopper loses one hit point. If the hp reaches less than 0 then add 100 units of food to the field and set itself to dead. Then it must check if the adultgrasshopper is stunned or status stunned then return. There is a one in three chance that the grasshopper tries to bite another insect. The grasshopper will also have a 1 in 10 chance to jump. The grasshopper will try to eat the food at the square up to 200. If the adultgrasshopper did eat, there is a 50% chance that the adulthopper would be stunned right after eating. The adultgrasshopper then attempts to move in the current direction until the distance runs out of the adultgrasshopper runs into a pebble.

}

void jump();

{

PSEUDOCODE TIME:

random degrees;

random length;

convert degrees to radians;

move in X + rsin(rad) Y + rsin(rad)

check if there is a pebble or pos is valid

move to that pos

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

StudentWorld

virtual int init();

{

CAREY SAYS READ THE SPECIFICATIONS- HAS PSEUDOCODE INSIDE

}

virtual int move();

{

CAREY SAYS READ THE SPECIFICATIONS- HAS PSEUDOCODE INSIDE

{

virtual void cleanUp();

{

CAREY SAYS READ THE SPECIFICATIONS- HAS PSEUDOCODE INSIDE

}

//Helper Functions

std::string getWinningAntsName();

{

converts the private member position of the winning ant into the ants name to display onto the screen

}

Insect\* pickRandomActor(int x, int y, Insect\* other, int enemyNumber);

{

outputs an Insect pointer which returns a random enemy from the square on the field that would not be deemed an enemy by the insect passed into other.

PSEUDOCODE TIME:

Iterate through list at the spot,

if list contains enemy(defined aboce)

add onto list

pick random from newly created list

then return

}

int getTicks();

{

return the current tickcount of the arena

}

int eatFieldFood(int x, int y, int num);

{

decreases the food at (x,y) by the num specified in the parameter

}

bool loadyField();

{

sees if the field is loaded. PSEUDOCODE IN THE SPEC

}

void dropFood(int x, int y, int fmount);

{

places food into the current location (x,y) with the current food amount as specified in the parameter

}

bool hasPebble(int x, int y);

{

checks if the current position (x,y) stores a pebble or not

}

bool hasFood(int x, int y);

{

checks if the current position (x,y) already as food on it or not

}

bool hasEnemyInsects(int x, int y, int enemyNumber);

{

checks if the current position (x,y) on the field has enemyInsects (other colony ants or any type of grasshopper) and returns true if it does. Falso otherwise.

}

bool theSimulationIsOver();

{

sets the condition to end the game. If the ticks ever reach 2000

}

bool dangerInFront(int x, int y, int colony);

{

Checks if there is danger in the square (x,y) by looking if there are enemy insects that are not in the same colony as the original insect

}

bool pheroInFront(int x, int y);

{

Checks to see if there is a pheromone in square (x,y) by looking if there are the pheromones of the specific ant colony type in the square

}

bool weHaveAWinningAnt();

{

returns true if any anthill loaded onto the screen ever makes more than 5 ants - to which a winner is capable of being determined by the time that the simulation ends

}

void allocateInsects(int x, int y, int colony, Compiler\* comp, AntHill\* home);

{

makes an ant at (x,y) - usually where the anthill is - with the parameters to make the any as the starting position, what colony the ant is supposed to be in, and what instruction the ant should read.

}

void stunInsects(int x, int y);

{

stuns every insect on the current square (x,y) to have it stopped for two turns

}

void poisonInsects(int x, int y);

{

poisons every insect on the current square (x,y) to have it lose a certain amount of health for each insect

}

void updateTickCount();

{

increments the tick counter by one everysingle time that the function is called

}

void createAdultGrassHopper(int x, int y);

{

Dynamically allocates an adult grasshopper at space (x,y) with health set at sixtenhundred

}

void addFood(int x, int y, int fmount);

{

adds an amount of food - taken from the parameter - to the current square. If there is no food object there, allocate a new food object with the amount of food taken from the parameter

}

void increaseColonyNumber(int colonyN);

{

increase the amount of ants produced for the colony specified by the parameter

}

bool biteAtSpot(int x, int y, int dam, Insect\* other, int enemyNumber);

{

bite a random insect at (x,y) that the insect deems to be an enemy. Subtract damage from it specified in the parameter and if the bitee was an adultgrasshopper then 50% bite back

RECURSIVE FUNCTION

PSEUDOCODE:

if enemy exists

pick a random enemy to bite

if hp is less than 0

setdead

see if bitee was adult grasshopper

` then 50% it bites back

}

void updateDisplayText();

{

format the updateDisplay text with the proper formatting as in string streams

PSEUDOCODE in stringstreams

}

void updateTheDataStructureThatTracksWhereEachActorIs(std::list<Actor\*>::iterator &li, int oldX, int oldY);

{

change the actor position to the new position in the datastructure using project3 warmup and project specifications

}

void pickUpFood(Ant\* ant, int x, int y);

{

at position (x, y) in the field, pick up food for the ant thtat is specified in the parameters

}

void placePheromones(int x, int y, int colony);

{

at position (x, y), dynamically allocate a new pheromone of type colony into that position

}

int winningAntColonyNumber();

{

returns the colony number which has the most ants produced

}

void setWinningColony(int w);

{

sets the private winning ant to the int specified by the parameter

}

int getWinningColony();

{

returns the colony number of the CURRENT

}

bool compareToWinner(int colony);

{

sees if the parameter colony passed in is the same as the current winner as described by the specifications.

}

I made many private member variables for StudentWorld std::list<Actor\*> m\_array[64][64]; which helped construct an array to keep my actor pointers so I could call it when I need the actor pointers

Field m\_field; which helps store the field that I created in the StudentWorld init function

int m\_tickCount; which stores the integer value of how many ticks went by in the simulator

std::vector<int> colonySize; stores in colonySize[0] how many ants were produced in anthill 0, colonySize[1] how many ants were produced in anthill, colonySize[2] how many ants were produced in anthill 2, colonySize[3] how many ants were produced in anthill 3

int m\_colonyCount; how many anthills (or different colonies) there are in the current run of the simulator

std::vector<std::string> compilerN; Stores string names for each ant bug file that was passed into the program.

Compiler \*compilerForEntrant0, \*compilerForEntrant1,

\*compilerForEntrant2, \*compilerForEntrant3; declares the compiler pointers so that we can delete them in the destructor

int m\_winningAnt; returns the current winning anthill that the simulation sees as the anthill which produced the most ants.

2.)  I finished Everything for Project 3

3.)  I just tried to follow as close to the example program that I was provided with as I could

4.)  DESCRIPTION ON HOW I TESTED THE CLASSES

AS FAR AS I KNOW, NO FAILED FUNCTIONS

ACTOR:

I tested the actor class by making sure that an actor cannot be made. To further test the actor classes, I made sure that the functions in the actor class can be accessed in the derived classes, like the bools for instance. I tested the m\_dead accessors and mutators for actor by creating a derived version of the actor class and seeing if setting the state to dead would work or not. As well, I tested the m\_moved accessors and mutators to see if it would change the m\_moved status each time that the actor moved. Since Actor was a base/ superclass there was not much to test it - the later derived classes are way better. Making some functions pure virtual makes sure that I cannot declare that class.

POISON:

I tested the poison class by making sure that a poison will be created at the poison position specified by the field class. I made sure that the poison will poison every insect on that field by isolating the poison and one baby grasshopper first. I surrounded them with pebbles and stopped the program so I could trigger movement using the enter key. It worked as I saw that the health got lower as i printed the health onto the console screen.

I then proceeded to check if poison worked with the insects were stunned. The poison would sting multiple times to a babygrasshopper or ant that decided to rest at that spot. I then proceeded to add multiple babygrasshoppers and ants to see if the poison will poison if there are multiple insects on the same spot.

POOL:

Pool is pretty similar to poison: I tested the pool class by making sure that a pool will be created at the pool position specified by the field class. I made sure that the pool will stun every insect on that field by isolating the pool and one baby grasshopper first. I surrounded them with pebbles and stopped the program so I could trigger movement using the enter key. It worked as I saw that the health got lower as i printed the health onto the console screen.

I then proceeded to check if pool worked with the insects were stunned. The pool would stun multiple times to a babygrasshopper or ant that decided to rest at that spot. I then proceeded to add multiple babygrasshoppers and ants to see if the pool will stun if there are multiple insects on the same spot.

PEBBLE:

How did I test pebble? It's a pebble. Just kidding haha I made sure that the insects would not move onto the pebbles. I had to make sure that the insects would change directions if it moved onto the pebble. The pebble's primary purpose was to keep the actors contained in the field class that was loaded. I also made sure that NO ACTORS could ever go onto the pebble or that would ruin the pebble implementation.

FOOD:

I tested Food by making sure that food was created at the food poistion as the field specified. Then I wanted to make sure that food was allocated with the right amount of hit points by printing out the hit points every time that the food does something. I saw that a food was aloocated every time an insect would die - sent 100 units of food.

I made sure that there were never two food objects in the same space. I printed out how much food the ant took from the food to make sure the right amount of food was taken. I also checked if the food would be deleted after there was no food left at the designed space.

ENERGYHOLDER:

EnergyHolder is a bit like actor in that it is a Base Class in which pure virtual functions are defined so it is a bit harder to test the functionality. I did test what was special for energyholders though in the case of hit points that I would properly access and change them with accessor and mutator functions in my classes.

Since there were two pure virtual functions, I made sure that I could not create an Energyholder object by itself.

PHEROMONES:

Pheromones was a bit harder to test since I had to go into the BUG file to change the code to emit Pheromones. I tested multiple conditions in which to place pheromones after the ant walks or if the the ant was on a food. The pheromones would follow the ant around.

I would test the hitpoint of a single pheromone and follow it throughout its life in the simulation. The hit points would decrease every tick and the hit points would increase if an ant decided to increase it. The pheromone would be set to dead and would be removed from the field if the hit points ever fell below zero.

INSECT:

Insect was like EnergyHolder and Actor in the sense that it is a Base Class in which pure virtual functions are defined so it is a bit harder to test the functionality I did test what was special for Insects though in the case of being stunned and seeing if it was an enemy to other insects or not. Every time it ran over a pool or poison, I would see if the derived classes of Insect would sting for 2 turns or lose health due to poison. I had multiple stuns in the movement and if the insect was affected by status effects such as poison and pool - in which it cannot be pool stunned unless it leaves or comes back again.

Since there were two pure virtual functions, I made sure that I could not create an Insect object by itself.

ANTHIlL:

Anthill was pretty integral to the simulation in that it makes the ants and are the home base for the ants as well. I had to make sure that the proper amount of anthills were allocated for how many BUG files were entered into the program.

I made sure that the anthill produced an ant everytime it had over two thousand hit points. I did that by printing out the anthill health everytime do something was called onto the anthill.

Then I proceeded to make sure that 1500 hit points were taken away from the anthill hitpoints. I made sure that the anthill would eat food by seeing if the hp would go up everytime an ant would drop food onto the anthill. In this case, I boxed the anthill, food, and one ant so that the ant would be able to make it onto the anthill in high amount of chances.

ANT:

With the most integral part of the program being the ants program, I proceeded to test as much as I could with the ants. I knew that exactly five ants would be made in the first five ticks so I made sure that that happened. Each ant would appear and follow the instructions that accompanied it with the BUG file. I had to make sure the interpreter worked so I made commands as if on food and I would cout to the console log every time that happened. I did that for every instruction and every if command to see if the conditions got met and if the ant did the correct thing as the instruction got met. I checked the dying conditions as well to make sure that the ant got deallocated when the hp went to zero by printing the hit points to the console log.

It was also made sure that the ant could eat and pick up food by printing out the amount of food it carries and the amount of hit points that the ant currently has. I check if the ant can sense danger and pheromones in front of the ant by seeing the bool in front of the ant when the ant has danger or the pheromone in front of the ant.

GRASSHOPPER:

GrassHopper was just like Insect and EnergyHolder in that it is a Base Class in which a pure virtual functions are defined so it is a bit harder to test for functionality. I checked the current amount of the distance that the grasshopper has done by making sure that the accessor and mutator funtions work. I wanted to make sure that the distances would be reset if the distance goes down to 0 or if one of the grasshoppers runs into a pebble. I tested the move function side by side with these functionalities since the move function would be used for both the babygrasshopper as well as the adult grasshopper. I made sure that the grasshoppers would move in the direction that they were facing.

BABYGRASSHOPPER:

Since this was one of the first classes I created and it was for Project Part 1, I tested to make sure the babygrasshoppers would move only every 2 ticks by stopping the program and stepping through them. The biggest idea for babygrasshopper was to transform into an Adult grasshopper at hp 1600. I made sure that happened by printing to the console log the hit points and making sure that an adult grasshopper appeared, food was allocated to the field, and the babygrasshopper was removed from the screen by setting it to dead.

The only way babygrasshopper is differnt from adult grasshopper is in its doSomething. I made sure that the babygrasshopper could onyl get stunned by pools of water and that it would want to sleep on food if it eats it. I tested it by making it sleep every time, then every other time, then random time.

ADULTGRASSHOPPER:

The adultgrasshopper was the culmination of the grasshopper programming. The hardest part to test in this function would have to be BITE and BITE BACK. I tested for this by putting two adult grasshoppers right away surrounded by pebbles such taht when they land on each other, I would print out the hit points. The grasshoppers would keep on biting back at each other until one grasshopper died. Then I would check with an adult grasshopper and one ant while printing out the hit points for each insect.

The next thing to test was jump, to make sure that the grasshopper jumps to random squares. First I tested to make sure that the grasshopper always jumps. Then, I made sure that the grasshopper only jumps in 1 to 10 chance - also that it moves any spot 10 spots away, even diagonals. Then I would test to make sure the adult grasshopper faced the right direction after it jumped in that direction.

STUDENTWORLD:

I made sure that every dynamic allocation has a delete. I coutted when I made a new object and coutted when I deleted the object and made sure that they matched up with each other. I had to follow the data structures and made sure that each actor woudl be put in the right place. I did that by testing for nullptrs or if the actors were moved, I tested to see if the actors existed in the place that they moved to. I made sure that the colony count for the ants were correct by coutting colonyCount.

I watched the screen and made sure everything was working properly just like Carey Nachenburgs example bug program. I made sure that the text formats were all correct and that a \* would be put next to the winning ant. I made sure that the winning ant did get chosen. I then tested to make sure that the actors died when they were supposed to by coutting to the console log when they died. I made sure that every push back had an erase. I made sure sure all my functions worked. I made sure that every doSomething would doSomething except for pebble. I am going to talk about how I did the set move to make sure that an acotor would not move multiple times in one tick. I also made sure exactly the amount if BUG Files were made, compilers were compiled.