1.)

Actor

Most of the functions in my public class are there to allow an Actor pointer to call the function. Many of these functions are redefined in derived classes. The actor, and each of its children, has a direction parameter so a randDir() function is made in this class in order to allow all other derived classes to set their direction to random if need be; this isn’t virtual because the function won’t change for any derived classes. AreThereRocks() is in Actor because no other derived actor can be on the same square as a pebble. It’s not virtual since the function won’t change for specific derived clases. IsitDead, isInsect, isFood, isPebble, isPheromone, isDanger, are there to help identify Actors in the data structure for other functions, simply returning a true or false. IsItDead returns false in Actor but will be redefined later in EnergyHolder by whether or not the Boolean variable “isDead” is true or false. Each of these are obviously virtual since they need to be redefined. All are set to false and when defined by the appropriate derived classes, will spit true instead. The gotPoisoned, gotStunned, gotBit and getColony are all virtual functions because other derived Insect classes need to redefine these to make sure they work properly according to the specs. They return nothing and do nothing in this base class. DoSomething in Actor is a pure abstract function since no object will be made that is simply an actor. There is no need to define the function in this class.

EnergyHolder

The EnergyHolder class is the parent class of any class that has hitpoints or life. SetDead, IsItDead are both defined in this class and virtual since every class derived from EnergyHolder can be “dead” or “alive” depending on the class bool variable “isDead”. The functions might need to be redefined by derived classes, hence the virtual. EnergyHolder’s DoSomething is defined but doesn’t do anything meaningful since the class EnergyHolder won’t technically be constructed on its own. getHealth() returns the hitpoints member in the class. dropHealth decreases the hitpoints member by 1 unit. setHealth takes in a parameter and replaces hitpoints value with the parameter value. minusHealth and addHealth take in a parameter and respectively subtracts it from and adds it to the existing hitpoint value. The functions dealing with health aren’t virtual since they’re all relatively simple getters or setters that don’t need to be redefined by derived classes. Finally, eatFood increases the caller’s hitpoints by the amount of food eaten and also deals with detracting the amount of food from the food pile that the calling actor ate from. This isn’t virtual either since the function doesn’t need to be redefined for other classes that eat food off the field.

Insect

Insect class is the shared class by all the bugs. The moveInsectFrom function removes the caller’s pointer in the data structure slot and moves it to the destination data structure slot and updating the array of vectors. The function doesn’t need to be redefined so it remains unvirtual. isStunned checks whether or not the insect is stunned by evaluating whether or not the int stunned member of Insect is 0 or not – if it’s nonzero, it returns true, it is stunned. DropStun decreases the length of stun by one, getStun returns the amount of time left of being stunned stored in the stunned member, and setStun(int amt) sets the stunned member to an amount specified in the parameter. The virtual void functions gotPoisoned and gotStunned don’t do anything meaningful in Insect but are redefined in derived classes since insects react differently in the situation of being poisoned or stunned. gotBit(int amt) is defined however to do something meaningful, taking away the hitpoints from the Insect and checking to make sure the Insect isn’t dead. gotBit is set to be virtual since some insects will react differently to being bit such as the ant and adult grasshopper. biteBug(int amt) is a void function that when activated, calls the gotBit function for other insects that were targeted, inflicting damage depending on the parameter value. IsInsect is redefined here in this class to ensure a convenient way of recognizing whether or not a class is an insect or not by simply having the function return true in this class.

Ant

Ant’s getCompiler function returns the compiler pointer that was passed into the class constructor. Ant’s doSomething() function first checks its health and stunned status, then calls the simpleInterpreter a total of 10 times to complete the Ant’s turn in a tick. The simpleInterpreter gets commands from the compiler compiling the inputted bug file. The function evaluates these commands by using a switch statement that accounts for each possible command possibility. Each case ends with a commandCount++ to allow the next command in the file to be read. It can be commanded to generate a randomNumber. There is also a series of possible if statement commands that the file can read. Each command that alters the ants position or state returns true, counting as one of the ten iterations of the simple interpreter in each ant’s move during a tick. The ant can be commanded to moveForward according to its faced direction and also reset to being considered unbitten and unblocked; this returns true. emitPheromones creates a pheromone object on the field (or adds to an existing pheromone) depending on the ant’s colony – this returns true. faceRandomDirection changes the direction of the ant to a random direction, done by the randDir function in Actor class; this returns true. rotateClockwise and rotateCounterClockwise do exactly what their name implies, rotating the direction of the ant clockwise or counterclockwise by one and returning true. The bite bites a random bug on the same square as the ant by using the function “checkInsect” in the studentWorld class and also returning true. pickedUpFood tells the ant to pick up (400 at a time) and hold its food, stored in the food member variable – it also subtracts the picked up food amount from the food pile’s food unit; it returns true. dropFood does the opposite of pickedUpFood, dropping all of the value of its food member variable onto the floor at its respective location, either creating a new Food object at that location or adding to an existing food object, returning true after finishing the command. eatFood commands the ant to eat some of its food that it’s holding, 100 at a time. This returns true. The command “invalid” returns false, which subsequently kills the ant.

Ant’s getColony function returns the colony number that it’s apart of; it’s redefined from Actor so hence it’s virtual. The gotBit(int amt) function is redefined, hence virtual, in ant since the ant needs to remember where on the field it was bitten. The getBit() function returns the bool variable prevBitten, which determines whether or not the ant was bitten on that field location. Ant’s resetBit() is only called when it moves away from the square of when it was bitten, such as when it’s commanded to moveForward. It resets the prevBitten bool to be false and the bitX and bitY coordinates of where it was bitten to be -1. The spec didn’t specify the default unbitten coordinates (or specify using coordinates) for ant, so I made it to be -1. getBlocked() returns the prevBlocked bool member variable, seeing whether or not the ant was blocked at the specific location. resetBlock makes prevBlocked false.

Adult Grasshopper

The doSomething of adult grasshopper first checks the health and stunned status of the grasshopper before doing something meaningful in the tick turn. If the grasshopper shares the space with another bug, it’ll have a 1/3 chance of biting it – achieved using the randInt function. Otherwise, there’s a 1/10 chance of the grasshopper using the jump function, which utilizes a calculation taking a random radius and random theta and obtaining a jumpX and jumpY value for the grasshopper to jump to. The grasshopper checks for rocks before successfully making the jump. If it doesn’t jump, it’ll check for food at the square and eat. If it doesn’t do that either, it’ll check if the random direction and distance of walking is finished, and either change direction if it finished walking or if it was blocked by a rock, or walk forward one.

gotBit is virtual and redefined in grasshopper since adult grasshoppers have a likelihood of retaliating when bit. Because adult grasshoppers can’t be poisoned, the redefined gotPoisoned and gotStunned functions don’t do anything meaningful. Adult grasshoppers are also considered dangerous to ants, so I wrote a isDangerous function that returns true.

Baby Grasshopper

Baby grasshopper doSomething is very similar to adult grasshopper’s doSomething except lacks the biting portion and also does not jump to a spot 10 units away. Baby Grasshopper’s gotPoisoned function is defined to take away 150 hitpoints and check to make sure the Baby Grasshopper is dead or not. Baby Grasshopper’s gotStunned sets the stunned length to two of the baby grasshopper. Baby grasshopper’s are also considered dangerous to ants.

Anthill

Anthill’s getCompiler does the same as the Ant’s getCompiler, returning the compiler the user passed into its constructor. Anthill’s doSomething checks its health if it hit 0 or not, and then checks for food on its square, eating 10,000 at a time. If the Anthill has over 2000 health, it’ll create a new ant depending on what colony the anthill is labeled. Anthill’s getColony returns the colony member variable.

Pebble

Pebble’s doSomething doesn’t do anything meaningful. Pebble’s isPebble function simply returns true.

Poison

Poison’s doSomething calls studentWorld’s poisonAll function, which triggers the gotPoisoned function of each insect. There’s also a virtual isDanger function redefined from Actor that returns true.

Pool

Pool’s doSomething calls studentWorld’s stunAll function, which triggers the gotStunned function of each insect. There’s also a virtual isDanger function redefined from Actor that returns true.

Pheromones

Pheromone is an energy holder. It has a isPheromone function that returns true. Its doSomething class simply just depletes its health and sets to dead when health is 0.

Food

Food’s doSomething doesn’t do anything meaningful. However it has a pickUpFood function and dropFood function that respectively takes away or adds to a food pile on the field. Food’s isFood function returns true.

StudentWorld

* + removeDeadActor iterates through a specific vector in the 2D array field and erases and deletes all actors with the bool isDead as true.
  + removeNull iterates through a specific vector in the 2D array field and erases all nullpointer values. I used this to debug since I had a difficult bug in my move() function that showed nullpointers showing up in my vectors.
  + storeObject(Actor\* temp) adds an actor object into a specific vector of a slot in the 2D array. Used usually after making a new object of a class.
  + checkInsect iterates through a vector in one of the 2D array slots for actors with isInsect returning true.
  + checkFood iterates through a vector in one of the 2D array slots for actors with isFood returning true.
  + checkPebble iterates through a vector in one of the 2D array slots for actors with isPebble returning true.
  + checkPheromone iterates through a vector in one of the 2D array slots for actors with isPheromone returning true. It also returns the actor that has such a function returning true.
  + checkDanger iterates through a vector in one of the 2D array slots for actors with isDanger returning true.
  + checkAntOnHill iterates through a vector that an ant is standing on to see if an anthill of the same colony is sharing the same lot, returning either true or false.
  + getActor iterates through a vector in the array depending on the location and passes by reference the target actor (found by checking ID).
  + updateArray removes an insect that just moved from its old space from its vector location and puts it in the new vector that corresponds with the new coordinates that it stands on.
  + updateJump does the same as updateArray except takes in consideration only jumping grasshoppers.
  + setDisplayText takes in the tick count and prints the message displayed on top of the game screen.
  + poisonAll and stunAll are called to call every gotPoisoned and gotStunned functions of insects.
  + addFood adds a Food object onto an empty location or adds more food to an existing food object.
  + chooseRandomInsect iterates through a vector and puts all the insects in the vector into a new vector of insects, then takes a random integer using randInt from 0 to the amount of insects minus one.

2.)

Though I finished typing up the Ant class, doSomething, and interpreter, I don’t know how to get the USCBug to properly be utilized by the game and to have its commands read. Though every other actor works and appears on screen, my code never has shown ants on my screen. I also cannot get my Field to load without typing in the exact address of the field.txt file on my computer into the loadfield parameter. I also didn’t have time to implement smaller details such as ants knowing not to bite other ants that share the same colony as it.

3.)

There was a lot of freedom in what kind of a data structure is used but no guidelines to how to iterate through data structures the best. I used an iterator to navigate through some of my vectors if I knew I was going to delete/erase objects in the vector (so the size of the vector wouldn’t mess up my for loops if I used brackets). I used simple for loop statements for other more straightforward functions.

4.) For the actor class, I indirectly tested it by using it as a base class for all my other classes. I made sure the boundaries weren’t crossed in the actor 2D array for each actor, and additionally made sure that each class derived from Actor was able to compile correctly.

EnergyHolder was tested by making sure the functions dealing with hitpoints and death of insects were working properly, since that was the main purpose of creating the EnergyHolder class. I also made sure each virtual function called was the correct redefined function if appropriate.

Similar to Insect class, I made sure that the important functions such as the stunned and poisoned functions were working properly. I also made sure they were redefined appropriately if necessary.

Ant was one of the most difficult classes to implement – however, I was unable to test it thoroughly since I was unable to see the ants on screen. I relied on making sure that the functions I used in each portion of ant worked well in the grasshopper classes, so I could have a little hope that they worked fine in the ant class too.

Testing the grasshoppers, I made sure that first they moved without doing anything more such as eating or jumping. Then, I added each new feature and made sure that they also worked in addition.

Unfortunately I was not able to test Anthill much other than making sure it appeared on the map. I wasn’t able to produce any ants to say it worked well otherwise.

Pebbles were tested simply by making sure that the bugs don’t walk through the pebbles.

Poison and Pool was tested simply by running the code and seeing if the grasshoppers get poisoned and die on top of the poison or stunned and stayed for 2 ticks, which they did.

I am unable to test Pheromones since the ant class was not able to run since I wasn’t able to read in the ant files properly.

I tested the food class by allowing the grasshoppers to eat the food and checking if they disappear when their units are depleted completely, and also if they give health to the grasshoppers, which they did as well.