1. High-Level Descriptions of Public Member Functions in Classes
   1. StudentWorld class, I defined a few public member functions. All of these member functions were added to this class because it requires access to the field, or the lists in the field, or they were mandated by the spec.
      1. I defined addFood, which takes the x, y coordinates, and the amount to be added onto the field. It adds the amount to an existing Food object or creates a new one with the amount.
      2. I also defined deleteActor, which deletes the Actor object and removes it from the 2-D array of lists representing the field. It takes a pointer to the object, x, y coordinates, and an iterator indicating its place in the list.
      3. I also defined addActor, which takes in x, y coordinates, and a pointer to the Actor object. It adds the actor to the list at x, y.
      4. I also defined someIterator, which takes x, y coordinates, and a value to indicate if the iterator requested is the beginning or the ending iterator. It returns list.begin() or list.end() at the x, y coordinate.
      5. I also defined empty, which takes x, y coordinates and determines if the list at that position is empty.
      6. I also defined getFrontActor, which takes x, y coordinates and returns a pointer to the first actor in the list.
      7. I also defined createAnt, which takes x, y coordinates, the colony number, and a pointer to the colony Compiler object. It creates an Ant at the position indicated.
      8. I also defined dropPheromone, which is similar to addFood. It takes x, y coordinates, and the colony number of the Pheromone. It adds 256 to a pre-existing Pheromone of that colony (up to 768) or creates a new Pheromone object at that position.
      9. Lastly, I defined the init, move, and cleanup methods, which run the entire game as defined by the spec. init initializes the game and the field, move runs the game until 2000 ticks have passed, and cleanup removes all dynamically allocated object and cleans up the Field.
   2. Actor class – I defined the methods in this class because all Actor objects derived from this class need access to movement details, their StudentWorld details, and whether or not it is a Pebble.
      1. I defined doSomething as a pure virtual function since there shouldn’t be any declarations of a pure Actor object anyway. This method is therefore blank.
      2. I defined setMoved to set the bool m\_moved and getMoved which returns m\_moved to indicate if an actor has moved this tick. This prevents the Actor from moving multiple times in a tick.
      3. I defined setJustMoved to set the bool just\_moved and justMoved which returns just\_moved to mark that this actor moved. This allows StudentWorld to move the Actor object from one list to another.
      4. I defined setWorld which sets the StudentWorld\* m\_world and getWorld, which returns the m\_world pointer. This allows the Actor object to access the public functions of StudentWorld to understand its location and surroundings.
      5. I defined setPebble to set the bool notpebble to false and notPebble, which returns the bool notPebble. These methods allow an Actor to realize that it is/isn’t a Pebble.
   3. EnergyHolder class – I defined this class which extends Actor to classify an Actor that has some amount of energy. All of these methods were placed in this class because they all have something to do with energy and if the EnergyHolder object is dead or not.
      1. I defined doSomething as a pure virtual function since there shouldn’t be any declarations of a pure EnergyHolder object anyway. This method is therefore blank.
      2. I defined setDead to set the bool m\_dead to true, and isDead to return m\_dead or hp when it’s less than or equal to 0. These methods allow EnergyHolders to be classified as dead and marked for removal by the StudentWorld.
      3. I defined setHP to set the int hp, and getHP to return the int hp. These methods allow defining and accessing the energy of an EnergyHolder.
      4. I defined the start method, since all EnergyHolders perform similar behavior which is performed here. This method causes all EnergyHolders to lose 1 hp at the beginning of the tick, and check if they are dead. If they are dead, the method returns false, indicating that this EnergyHolder should not start its doSomething.
   4. Insect class – I defined this class which extends EnergyHolder to classify an Insect. All of these methods are placed in here because Insects move, can be stunned, and need random directions at times. This class defines the only organisms in the world shown through their ability to eat and move.
      1. I defined doSomething as a pure virtual function since there shouldn’t be any declarations of a pure Insect object anyway. This method is therefore blank.
      2. I defined randomDir since all Insects need to be able to pick a random Direction at times. This method returns a random Direction.
      3. I defined setStunned to set the int variable stunned and getStunned to return stunned. These methods indicate if an Insect has been stunned, and if so, for how many ticks.
      4. I defined setWaterStunned to set the bool water\_stunned and getWaterStunned to return water\_stunned. These methods indicate if the Insect has just been stunned by a WaterPool object, ensuring it is not stunned by the same WaterPool object until the Insect moves off.
      5. I defined moveActor to attempt to move an Insect in the Direction that it is currently facing, and return true if it does. If it runs into a Pebble then the method will return false, and the Insect will not move.
   5. Ant class – I defined this class which extends Insect to classify an Ant. An Ant is one of the primary Actors of the simulation as defined in the spec, and I placed these methods here to code its specific behavior and allow it to be bitten.
      1. I defined doSomething as a virtual method to enable polymorphism so that when the StudentWorld asks Actors to doSomething, the Ant class will have its own specific behavior to do. It firsts check if it can start, using EnergyHolder’s start method. Then it checks if it isn’t stunned. If so, it uses its Compiler pointer to access the next Command that it must perform. It enters this command into a switch, and executes the next command be it moving forward, changing direction, biting, etc. It calls other private methods to assist it in allowing a wide range of behaviors that an Ant can perform based on its Compiler Commands.
      2. I defined bitten to set the bool variable prevBitten to true, so when other Insects bite this Ant, it will be able to remember that it was previously bitten on the current square.
   6. GrassHopper class – I defined this class which extends Insect to classify a generic GrassHopper. GrassHoppers are specific insects with instructions to move a set distance, eat, and perform GrassHopper behaviors at the beginning of the turn, thus I placed these methods here.
      1. I defined doSomething as a pure virtual function since there shouldn’t be any declarations of a pure GrassHopper object anyway. This method is therefore blank.
      2. I defined setDistance to set the int distance, getDistance to return distance, and newDistance to set distance to a random int between 2 and 10. These methods allow a Grasshopper to know how much distance it has left to walk, adjusting this distance, and setting a new random one when it needs to.
      3. I defined eat to allow a Grasshopper object to check for a Food object, eat some/all of it, and sleep after based on a random token. If it doesn’t sleep however, this method describes how the GrassHopper must then move or change its Direction.
      4. I defined begTurn to call EnergyHolder’s start method to ensure it is alive, to check if it is stunned and decrement the counter, and setting stunned to 2, to ensure the object cannot do anything for another 2 ticks, as per the spec.
   7. AdultGrasshopper class - I defined this class which extends GrassHopper to classify an AdultGrasshopper. An AdultGrasshopper is one of the primary Actors of the simulation as defined in the spec, and I placed these methods here to code its specific behavior, and allow it to bite other Insects.
      1. I defined doSomething as a virtual method to enable polymorphism so that when the StudentWorld asks Actors to doSomething, the AdultGrasshopper class will have its own specific behavior to do. It firsts check if it can begin its turn, using GrassHopper’s begTurn method. Then it checks if it can bite, and if so it ends its turn. If not, it checks if it can jump, calling the private jump method, and if so it ends its turn. If not, it will then proceed to eat and move.
      2. I defined bite so the AdultGrasshopper can check if it is meant to bite a random Insect occupying the same spot. This method is public so that if an Ant bites an AdultGrasshopper, it has the opportunity to retaliate.
   8. BabyGrasshopper class - I defined this class which extends GrassHopper to classify a BabyGrasshopper. A BabyGrasshopper is one of the primary Actors of the simulation as defined in the spec, and I placed these methods here to code its specific behavior.
      1. I defined doSomething as a virtual method to enable polymorphism so that when the StudentWorld asks Actors to doSomething, the BabyGrasshopper class will have its own specific behavior to do. It firsts check if it can begin its turn, using GrassHopper’s begTurn method. Then it determines if it has enough energy to transform into an AdultGrasshopper, where the BabyGrasshopper object would die, and a new AdultGrasshopper object would be created to take its place. If not, it calls GrassHopper’s eat method to move and eat.
   9. Anthill class – I defined this class which extends EnergyHolder to classify an Anthill. An Anthill’s primary purpose is to produce additional Ant objects for the colony when it has enough energy. Thus, these methods code its specific behavior and allow its colony to be identified.
      1. I defined doSomething as a virtual method to enable polymorphism so that when the StudentWorld asks Actors to doSomething, the Anthill class will have its own specific behavior to do. It firsts check if it can start, using EnergyHolder’s start method. If so, it checks if it has enough energy to create an Ant object for the colony, and uses StudentWorld’s createAnt method to produce one.
      2. I defined getColony to return the int m\_colony. This method allows one to determine what colony this Anthill belongs to and act accordingly.
   10. Poison class – I defined this class which extends Actor to classify a dangerous Poison object. Poison’s primary purpose is to poison and reduce the energy of qualifying Insects that are on the same space as this object. Thus, I defined this behavior using this doSomething method.
       1. I defined doSomething as a virtual method to enable polymorphism so that when the StudentWorld asks Actors to doSomething, the Poison class will have its own specific behavior to do. This Poison object obtains beginning and end iterators from StudentWorld’s someIterator method, and checks if there are any Insects that are not AdultGrasshoppers occupying the same space as this Poison object. If so, it subtracts 150 hp from each Insect.
   11. WaterPool class - I defined this class which extends Actor to classify a dangerous WaterPool object. WaterPool’s primary purpose is to stun qualifying Insects that are on the same space as this object for 2 more ticks. Thus, I defined this behavior using this doSomething method.
       1. I defined doSomething as a virtual method to enable polymorphism so that when the StudentWorld asks Actors to doSomething, the WaterPool class will have its own specific behavior to do. This WaterPool object obtains beginning and end iterators from StudentWorld’s someIterator method, and checks if there are any Insects that are not AdultGrasshoppers occupying the same space as this WaterPool object. If so, it checks if they were just stunned by this WaterPool object using Insect’s getWaterStunned method. If this method returns false, it stuns each Insect for an additional 2 ticks.
   12. Pebble class – I defined this class which extends Actor to classify a Pebble object. A Pebble’s primary purpose is to block Actors from moving on to it. Thus, I defined this behavior by using Actor’s setPebble method to indicate that this Actor is indeed a Pebble and will block movement.
       1. I defined doSomething as a virtual method to enable polymorphism so that when the StudentWorld asks Actors to doSomething, the Pebble class will have its own specific behavior to do. However, since it is a Pebble, it doesn’t do anything.
   13. Pheromone class – I defined this class which extends EnergyHolder to classify a Pheromone object. Pheromones dropped by Ants for their own purposes and gradually die over time. Thus, I defined this behavior and allowed access to its colony ID through these methods.
       1. I defined doSomething as a virtual method to enable polymorphism so that when the StudentWorld asks Actors to doSomething, the Pheromone class will have its own specific behavior to do. This method only calls EnergyHolder’s start method, since a Pheromone does not need to do anything else.
       2. I defined getColony to return the int m\_colony. This method allows one to determine what colony this Pheromone belongs to and act accordingly.
2. Functionality I failed to finish
   1. There is no functionality that I failed to implement.
3. Design Decisions and Assumptions
   1. It was not specified if my program should be able to run with less than 4 ants. In fact, the spec seemed to indicate through its variable names such as (\*compilerForEntrant0, \*compilerForEntrant1, \*compilerForEntrant2, \*compilerForEntrant3) that this program should only run with 4 ant colonies. However, the simulation indicated otherwise since it could be run with fewer than 4 ant colonies. Thus, I followed the simulation’s added functionality, and allowed my project to run with fewer than 4 ant colonies.
   2. The simulation says “ ants” after the number of each colony’s ants in the status text at the top, but spec specifically says in that section to add 2 spaces after the colony’s ant number. In fact, the example in the spec is “Ticks: 1134 - AmyAnt: 32 BillyAnt: 33 SuzieAnt\*: 77 IgorAnt: 05” where there are no “ ants” after the score. Thus, I followed the spec and did not add the wording “ ants” after the number of each colony’s ants.
   3. The simulation’s ticks count down from 2000, whereas the spec says to count up to 2000. I followed the spec and counted up to 2000, ending the game when the ticks reach 2000.
   4. The spec did not say anything about decrementing the ant number in the status text if an Ant dies. Since the simulation doesn't decrement ants upon dying, and agrees with the spec, I followed this concept and do not decrement my ant numbers when they die.
   5. I realized that multiple Ants could drop Food objects on an Anthill, and this would make it difficult to appropriately add the energy to the Anthill object immediately. Thus, instead of coding that behavior for Food in the Anthill class, I coded it into the Ant class, where an Ant checks when it is dropping Food, if its Anthill is in the same location. If so, then it simply increments the Anthill’s energy by the amount of Food dropped.
4. How I tested each Class
   1. StudentWorld class – I compiled and ran the code, while in another window I ran the simulation given to us. They run nearly identically, so I was able to determine that my StudentWorld class functions. The game status at the top of the screen updated appropriately as additional ants were born and the asterisk was updated next to the leading ant colony’s name. The ticks decremented appropriately and the field was successfully generated in accordance with the text file. Each Actor was called to do something each tick, and dead Actors were promptly removed in the same tick. A winner was declared at the end of 2000 ticks if a colony was in the lead, and otherwise the simulation ended with no winner. All of this was in accordance with the spec and mirrored the simulation given to us, thus passing these tests.
   2. Actor class – Since I did not need to create objects of this class, I tested this class by calling its methods, and seeing if it did the appropriate behaviors. The getWorld and setWorld methods appropriately return and set the StudentWorld object that each Actor has a pointer to. Similarly, Actors that were not Pebbles returned true for notPebble, and Pebble objects returned false due to setPebble being called. After an Actor moved, the setMoved actor was appropriately called and the getMoved prevented it from moving again in the same tick. Lastly, the justMoved method allowed an Actor to be moved to a different list in the 2-D array reflecting its movement. Clearly, the Actor class passed all these tests.
   3. EnergyHolder class – Since I did not need to create objects of this class, I tested this class by calling its methods, and seeing if it did the appropriate behaviors. Upon calling the setDead method, the EnergyHolder was considered dead, and StudentWorld calling its isDead method allowed it to be removed from the game. As the EnergyHolders lost energy each tick, the setHP and getHP method appropriately changed and reflected this object’s energy. Lastly, the start method appropriately decremented the energy by 1 each tick, and called setDead when the energy was less than or equal to 0, returning immediately so it is removed. Thus, the EnergyHolder class passed all these tests.
   4. Insect class – Since I did not need to create objects of this class, I tested this class by calling its methods, and seeing if it did the appropriate behaviors. Upon calling randomDir and setting its Direction to randomDir, the Insect appropriately changed its Direction. Similarly, the getStunned and setStunned allowed the Insect to change its stun count and access it. The same thing occurred with getWaterStunned and setWaterStunned where the variable was appropriately changed and accessed. Lastly, moveActor moved the Insect 1 step in the direction that it was facing and returned true if there was no Pebble in the way. Otherwise, it returned false, as it should have. Thus, the Insect class obviously does what is supposed to and passes all these tests.
   5. Ant class – I compiled the programs with USCAnt.bugs and compared their behavior with that of the simulation. Since they run identically, I knew I had a good start. However, I needed to check the other commands and if\_statements that the USCAnt did not call. Thus, I created the UCAnt.bug, UCLAAnt.bug, and UCBAnt.bug. The UCAnt simply moves away from the Anthill in one direction until it encounters food, rotates 180 degrees when it does, picks up the food, proceeds in a straight line back to the Anthill, deposits the food after eating some, rotates, and does the same thing all over again. This Ant allowed me to test picking up food, eating food, dropping food, rotate methods, Anthill conditions, food conditions, moving, goto commands, and if\_commands.

The UCLAAnt goes out on a murderous rampage where it faces random directions, moves, encounters other Insects, is bitten, and bites a random one. This Ant allowed me to test the bite method, faceRandomDirection command, and the generateRandomNumber to determine which Insect to bite. Lastly, the UCBAnt is a cautious Ant that follows its colony’s pheromones to determine a safe path, avoiding danger, and being blocked. This Ant allowed me to test the emitPheromone command, the danger in front of me condition, the pheromone in front of me condition, and if the Ant was previously blocked condition. Thus I was able to successfully check and confirm that my Ant class works for all Commands and Conditions.

* 1. GrassHopper Class – Since I did not need to create objects of this class, I tested this class by calling its methods, and seeing if it did the appropriate behaviors. When I called getDistance, I appropriately received the distance, and I could similarly set it using setDistance. I also received a new random distance when I called newDistance. Lastly, upon calling the eat method, the GrassHopper appropriately checked if there was a Food object in the same spot. If so, it ate 200 (or however much was left) of the Food’s energy, changing the hp for both appropriately. It then randomly determined to sleep, ending the method if so. Otherwise, it successfully moved in the direction it was facing, decrementing the distance, or appropriately changing its distance/direction. Clearly, the GrassHopper class successfully implements all of these methods and thus passes these tests.
  2. AdultGrasshopper class – I tested the AdultGrasshopper class by placing it in different fields. I first placed it in an empty field, where once every 3 ticks, it appropriately moved and changed direction, in accordance with its instructions. However, it would also occasionally jump in a random direction inside a radius of 10. Eventually it died due to a lack of food. I then placed it in a field of Pebbles. It appropriately could neither move nor jump. I then placed it in fields of WaterPool objects and Poison objects. Neither object had any effect on the AdultGrasshopper. I then placed it in a field full of Food. The AdultGrasshopper appropriately survived the duration of the simulation by consuming Food, occasionally not moving due to falling asleep from the Food coma. Lastly, I placed the AdultGrasshopper in a field of Ants, BabyGrasshoppers, and AdultGrasshoppers. In this vicious field, all of the Actors died much more rapidly, indicating that the bite method for the AdultGrasshopper works appropriately. Thus, the AdultGrasshopper class clearly behaves appropriately and passed all of these tests.
  3. BabyGrasshopper class – I tested the BabyGrasshopper class by placing it in different fields. I first placed it in an empty field, where once every 3 ticks, it appropriately moved and changed direction, in accordance with its instructions. I then placed it in a field with Pebbles. The Pebbles successfully blocked the BabyGrasshopper from moving. I then placed it in a field surrounded by Food. It appropriately ate the food and eventually transformed into an AdultGrasshopper upon gaining enough energy. I then placed it in a field surrounded by Poison objects. It appropriately died after a few ticks, placing a Food object where it died. Lastly, I placed it in a field surrounded by WaterPool objects. It was appropriately stunned and moved once it was no longer stunned (4 ticks instead of the usual 2). Thus, the BabyGrasshopper class passed all these tests and behaves appropriately.
  4. Anthill class – I tested this class by changing its initial energy, and in conjunction with the Ant class. After changing its initial energy to 16000, it appropriately created 10 Ant objects of the same colony, and died after 990 additional ticks. After changing its initial energy to 15010, it created 9 Ant objects, and died after 1501 additional ticks (instead of incorrectly creating 10 Ant objects and dying immediately). Lastly, in conjunction with the UCAnt.bug as indicated above, the Anthill was repeatedly brought Food objects, increasing its energy, and causing it to produce additional Ant objects. All of this is in accordance with the spec, so the Anthill class passed the tests.
  5. Poison class – I tested this class by creating 3 fields. In the first field, a BabyGrasshopper is surrounded by Poison objects. It appropriately dies in a few ticks. Similarly, I placed an Ant surrounded by Poison objects in the second field. It died in a few more ticks. Lastly, I placed an AdultGrasshopper surrounded by Poison objects in the third field. It died much later due to a lack of food in the field. Thus, the Poison objects appropriately acted on the BabyGrasshopper and the Ant, and not the AdultGrasshopper.
  6. WaterPool class – I tested this class by creating 3 fields. In the first field, a BabyGrasshopper is surrounded by WaterPool objects. It is appropriately stunned for 2 ticks after each move. Similarly, I placed an Ant surrounded by WaterPool objects in the second field. It is also appropriately stunned for 2 ticks after each move. Lastly, I placed an AdultGrasshopper surrounded by WaterPool objects in the third field. It did not affect the movement of the AdultGrasshopper because the WaterPool objects did not stun it. Thus, the WaterPool objects appropriately acted on the BabyGrasshopper and the Ant, and not the AdultGrasshopper.
  7. Pebble class – I tested this class by creating 3 fields. In the first field, a BabyGrasshopper is surrounded by Pebble objects. It appropriately cannot move. Similarly, I placed an Ant surrounded by Pebble objects in the second field. It also cannot move. Lastly, I placed an AdultGrasshopper surrounded by Poison objects in the third field. The AdultGrasshopper was also unable to move. Thus, the Pebble objects appropriately acted on the BabyGrasshopper, the Ant, and the AdultGrasshopper by preventing them from moving onto it.
  8. Pheromone class – I tested this class in conjunction with the Ant class. I first had an Ant emit a Pheromone each time it moved, quickly covering much of the field in Pheromones. Many of these Pheromones survived for a long time, acting appropriately. Then, I had the Ant only drop 1 Pheromone after it moves off an Anthill. This Pheromone died appropriately after 256 ticks. Lastly, I had an Ant emit a Pheromone on the same spot 5 times before moving off. This Pheromone appropriately died after 768 ticks, indicating that the Pheromone class functions as indicated by the spec.