**Descriptions of Public Member Functions**

**Actor: public GraphObject**

*Actor (StudentWorld\* world, int imageID, int startX, int startY, int depth, Direction direction = none):* This constructor initializes a GraphObject with the parameter specified, but also initializes the StudentWorld\* m\_world pointer to “world” and hasMoved boolean to false.

*virtual void doSomething() = 0:* I decided to make the doSomething() function pure virtual, as every non-abstract derived class has a different implementation.

*Virtual void getStunned():*  This function is used to stun actors that can be stunned. As only insects can be stunned I made the function virtual, and the base implementation simply return.

*virtual int returnHitPoints() const:* This function was designed as an accessor for my EnergyHolder class. I used this to access the hitPoints of actors in code that utilized a returned pointer to an actor from another function. Because of this, it had to be a base class function. I made the base class function simply return -1, as not all actors have hitPoints.

*virtual void changeHealth():* This function was created in a similar context as the previous function. I needed it to alter the health of EnergyHolders whose pointers were returned as Actors. The base class implementation simply returns.

*virtual void getPoisoned():* This function is used to poison all poisonable Actor objects in the StudentWorld. As essentially only Insects can be poisoned, I made the function virtual, and the base class implementation return.

*virtual bool isEnemy(int colony) const:* This function returns a Boolean regarding if an actor is an enemy to the colony specified in the parameter. As different actors have different enemy statuses, I made the function virtual and made the base class implementation return false, as more actors are not enemies than are.

*virtual bool blocksMovement() const*: This function essentially only returns if the actor is a Pebble, as Pebbles block movement. I made the function virtual, as all other actors return false. This is also the reasoning for my base class implementation returning false.

*virtual void getBitten(int amount):* This function is used to bite actors that can be bitten. As only insects can be bitten, this function is virtual and simply returns in the base class implementation.

*virtual bool isDangerous(int colony) const:* This function returns true if the actor is dangerous to the colony specified in the parameter (adult grasshoppers, poison, other ant colonies, etc.). As such, the function is virtual. As most actors are not dangerous, the function’s base class implementation returns false.

*virtual bool isEdible() const:* This function essentially returns true for Food objects, as it is used in functions that allow Insects to pick up/eat food. As such, this function is virtual and returns false in the base class.

*Virtual bool isMyAnthill(int colony) const:* This function returns true if the Actor object is the Anthill of the colony specified in the parameter. This function only returns true for specific Anthills, so it is virtual. The base class implementation return false, as most Actors return false.

*virtual bool isPheromone(int colony) const:* This function returns true if the Actor object is a Pheromone of the colony specified in the parameter. This function only returns true for specific Pheromones, so it is virtual. The base class implementation return false, as most Actors return false.

*studentWorld\* returnWorld():* This function returns the m\_world private member variable of the actor. This is useful for asking the StudentWorld to perform game changes, such as biting another actor, adding an actor to the simulation, etc.

*virtual bool isLiving() const*: This function returns true if the Actor is living, or should “doSomething” during the current tick. This implementation is changed in the EnergyHolder class, but is virtual due to it’s different implementations. The base class returns true, as even Pebbles are allowed to doSomething, even though they do nothing.

*void Moved():* This function changes the “hasMoved” boolean variable. This function was implemented to insure that actors who changed positions and moved to different parts of the 2D array of linked lists in the StudentWorld variable did not move twice in the same tick.

*bool hasNotMovedYet() const:* This function returns the opposite of the “hasMoved” member variable. This is used in the StudentWorld move() function.

*void resetMoved():* This function sets the “hasMoved” variable to false. This is used on every actor at the beginning of the StudentWorld move function.

*virtual ~Actor():* This destructor is virtual to prevent undefined behavior in the derived classes. It also utilizes the GraphObject’s setVisible function and sets setVisible to false.

**StudentWorld: public GameWorld**

*init():* This function initializes the game state. It sets the member variable “ticks” to 0 along with the number of Ants in each colony. It also initializes each value in the “timeOfIncrement” array to 5000. The reason for this value is that the array holds the tick count of the latest increment in ants. If the ant count for the colony is never incremented, then a tiebreaker must ensue; a value of 5000 would never win the tiebreaker. It also sets all of the antNames to empty strings. This function also sees of the LoadFile function returns true, and returns GWSTATUS\_CONTINUE\_GAME if it does. If not, it returns GWSTATUS\_LEVEL\_ERROR.

*move():* This function is responsible for calling each actor’s doSomething function and furthering the simulation. First, it increments the ticks. After this, it resets each actor’s hasMoved variable using the Actor resetMoved() function. It then iterates through the array of linked lists and calls each actor’s doSomething function if it has not moved and is living. If the actor moves, it updates its position in the array. The function then removes all dead actors using the removeDeadActors() function. It then sets the display text using the setDisplayText() function. Finally, it checks if the simulation is over and returns the winning ant using return GWSTATUS\_PLAYER\_WON and the setWinner function or no winner using GWSTATUS\_NO\_WINNER if so. If not, it returns GWSTATUS\_CONTINUE\_GAME.

*void cleanup():* This function removes an dynamically allocated memory from the simulation by iterating through the data structure and deleting each pointer.

*bool loadFile():* This function is responsible for loading the field file and adding objects to the simulation. First it attempts to load the file and returns false if there was an error. It then attempts to compile the ant files, depending how many entrants enter. It also stored the ant names to the member variables in StudentWorld. If there are any errors, it returns false. It then goes through the field data file and adds objects/actors to the simulation. If it reaches this point, it returns true.

*Actor\* getEdibleAt(int x, int y):* This function returns a pointer to a Food object if it exists on the coordinate. If not, it returns nullptr.

*void addActor(Actor\* actor):* This function is used to add an actor to the simulation. It adds it to the data structure at the coordinate of the new actor.

*int getTicks() const:* returns the “ticks” variable.

*int returnWinningAntNumber(int antOne, int antTwo, int antThree, int antFour) const:* This function returns which colony has the most ants, aka the greatest out of the 4 variables passed through the parameters. If there is a tie, it sets it to the ant who was increments earliest.

*bool canMoveTo(int x , int y):* This function returns essentially if any objects block movement at the coordinate (using the blocksMovement() function on any actor).

*bool stunAllStunnableAt (int x, int y):* This function stuns all of the stunnable actors at the coordinate specified in the parameter. It calls the getStunned() function on each actor in the coordinate.

*bool poisonAllPoisonableAt (int x, int y):* This function poisons all of the poisonable actors at the coordinate specified in the parameter. It calls the getPoisoned() function on each actor in the coordinate.

*bool biteEnemyAt (Actor\* me, int colony, int biteDamage):* This function creates a vector with all actors that are enemies to the colony specified and are not the actor *me* and chooses a random actor in that vector to *getBitten,* the damage, being specified by the biteDamage parameter.

*void increaseScore (int antID):* This function increases the numAnts variable for the colony specified in the parameter. It also sets the array value for that ant to the current ticks value.

*bool isEnemyAt (int x, int y, int colony):* This function iterates through the list at the specified coordinate in the array and calls the isEnemy function for each actor, passing colony as the parameter. It returns true if any actor returns true, else it returns false.

*void setDisplayText():* This function essentially calls the stringFormatter function with all of the appropriate game variables (ticks, numAntsOne, numAntsTwo, etc.) and uses the setGameStatText to the string created by that function.

*string stringFormatter (int ticks, int ants0, int ants1, int ants2, int ants3, int winningAntNumber):* This function uses stringstreams to create the string that should be utilized in the setDisplayText() function. It sets the string to the specifications required in the project spec to be displayed at the top of the screen.

*void removeDeadActors():* This function iterates through the entire data structure and removes any actors who return false for the isLiving function. It deletes their pointers and erases them from the list.

*bool isAnthillAt(int x, int y, int colony) const:* This function iterates through the list at the specified coordinate and calls the isMyAnthill function with the specified colony as the parameter. If any actor returns true, it returns true. If not, it returns false.

*bool isDangerAt(int x, int y, int colony) const:* This function iterates through the list at the specified coordinate and calls the isDangerous function with the specified colony as the parameter. If any actor returns true, it returns true. If not, it returns false.

*~StudentWorld():* The destructor just calls the “cleanup” function.

**Pebble: public Actor**

*Pebble(int startX, int startY, StudentWorld\* world):* This constructor simply passes its parameters to the Actor constructor, but also passes IID\_ROCK for the image ID and sets the depth to 1 and the direction to right.

*virtual void doSomething():* This function simply returns, as Pebbles don’t do anything.

*virtual bool blocksMovement() const:* This virtual function returns true, as Pebbles block movement.

**EnergyHolder: public Actor**

*EnergyHolder(int startX, int startY, int imageID, int depth, Direction dir, int health, StudentWorld\* world):* This constructor passes the parameters to the Actor constructor, but also sets the member variable “m\_hitPoints” to health and “m\_food” to 0 as they start out holding no food.

*virtual int returnHitPoints() const:* This function returns m\_hitPoints.

*virtual void changeHealth(int amount):* This function adds the amount to m\_hitPoints (amount can be negative). It is virtual due to the fact that certain classes have max healths.

*void setDead():* This function sets m\_hitPoints to zero and if the function becomesFoodUponDeath() returns true, adds 100 units of food to the simulation.

*virtual bool isLiving() const:* This virtual function returns true if m\_hitPoints is greater than 0, and false otherwise.

*int returnFoodHeld() const:* This function returns the value of m\_food.

*void dropFood(int amount):* If the getEdible function of the world of the Actor returns a nullptr, it creates a new Food object by using the addActor function in StudentWorld with the health being the amount specified, else it adds the amount specified to the health of the object.

*int pickUpFood(int amount):* This function picks up/removes the amount of food specified from the StudentWorld and returns that value. If there is not enough food it picks up what is there and returns that value. If there is no food, it returns 0. Also, it only picks up to 1800 units of food.

*int eatFoodFromStorage(int amount):* This function eats the amount of food specified and decrements the m\_food variable accordingly. It also increased m\_hitPoints accordingly by using the changeHealth function. If there is not enough food, it eats what is left and returns that value. If there is no food, it returns 0.

*int pickUpAndEatFood(int amount):* This function eats the amount of food specified from the point it is standing on and returns the food eaten. If there is not enough food, it eats what is left and returns that value. If there is no food, it returns 0.

*virtual bool becomesFoodUponDeath():* This function returns whether an object should drop food after dying. As such, it is virtual. This is essentially only true for Insects so the base class implementation returns false.

**Pheromone: public EnergyHolder**

*Pheromone(int startX, int startY, int ID, StudentWorld\* world, int imageID):* This constructor uses the parameters to construct the EnergyHolder class, but also uses the ID variable to construct the variable “antID” and initialize it to ID, and passes 256 for the health, 2 for the depth, and right for the direction.

*virtual void doSomething():* This virtual function only decrements the Pheromone’s health by 1.

*virtual bool isPheromone(int colony) const:* This function checks to see if its antID is the same as the colony and returns true if so, and false otherwise.

*virtual void changeHealth(int amount):* This virtual function does the same as its base class implementation but makes sure the Pheromone’s health does not go over 768.

*virtual bool isPheromone() const:* This virtual function returns true.

**Food: public EnergyHolder**

*Food(int startX, int startY, int health, StudentWorld\* world):* This function simply passes its parameters to the EnergyHolder constructor along with IID\_FOOD for the image ID, 2 for the depth, and right for the direction.

*virtual void doSomething():* This virtual function simply returns, as Food does not do anything.

*virtual bool isEdible() const:* This virtual function returns true.

**Anthill: public EnergyHolder**

*Anthill(int startX, int startY, int antNumber, StudentWorld\* world, Compiler\* compiler):* This function simply passes its parameters to the EnergyHolder constructor along with IID\_ANT\_HILL for the image ID, 8999 for the health, 2 for the depth, and right for the direction. It also initializes its Compiler\* m\_compiler variable to compiler and its antID variable to antNumber.

*virtual void doSomething():* This function first decrements the health of the Anthill by one. If it is not living any more, it returns. It then attempts to eat up to 10000 units of food and returns if it eats any. After this it sees if its hitpoints are at least 2000, and adds an ant to the simulation if so. It also decrements its health by 1500 and asks StudentWorld to increase the score of its colony using increaseScore.

*virtual bool isMyAnthill(int colony) const:* This function compares the colony variable to the Anthill’s antID variable and returns true if they are equal and false otherwise.

*int returnImageID():* This function returns the imageID for the ant it is creating in the doSomething() function by referencing it’s antID variable.

**TriggerablePool: public Actor**

*TriggerablePool(int imageID, int startX, int startY, StudentWorld\* world):* This constructor just constructs the Actor variable with its parameters but also sets the depth to 2 and the direction to right.

*virtual bool isDangerous(int colony):* This virtual function returns true.

**Water: public TriggerablePool**

*Water(int startX, int startY, StudentWorld\* world):* This constructor constructs the TriggerablePool and also passes IID\_WATER\_POOL as the imageID.

*virtual void doSomething():* This function attempts to stun all objects on the same coordinate using the StudentWorld’s stunAllStunnable function.

**Poison: public TriggerablePool**

*Poison(int startX, int startY, StudentWorld\* world):* This constructor constructs the TriggerablePool and also passes IID\_POISON as the imageID.

*virtual void doSomething():* This function attempts to stun all objects on the same coordinate using the StudentWorld’s poisonAllPoisonable function.

**Insect: public EnergyHolder**

*Insect(int imageID, int startX, int startY, int depth, Direction direction, int health, StudentWorld\* world):* This constructor calls the EnergyHolder constructor with th parameters but also sets the variable m\_stunCount to 0 and the boolean wasStunned to false.

*bool doSomethingInsect():* This function decrements hitPoints by one, checks to see if the Insect isLiving and if not, returns true. Also if the Insect is stunned, it lowers the insects m\_stunCount and returns true. If the function reaches the end, it returns false.

*virtual void getBitten(int amount):* This virtual function decrements the health of the Insect by amount. It is virtual because ants do something else when bitten as well.

*virtual void getPoisoned():* This function decrements the Insect’s health by 150. It is virtual because adult grasshopper’s do not get poisoned.

*void lowerStun():* This function lowers the m\_stunCount by 1.

*virtual bool becomesFoodUponDeath() const:* This virtual function returns true.

*void increaseSleepTicks(int amount):* This function increase m\_stunCount by the amount specified.

*virtual bool isEnemy(int colony) const:* This virtual function returns true, as all insects are enemies besides ants of the same colony.

*virtual void getStunned():* This function makes sure the wasStunned variable is false, and if so, adds 2 to the m\_StunCount and sets wasStunned to true.

*void noLongerStunned():* This function sets wasStunned to false.

**Grasshopper: public Insect**

*Grasshopper(int imageID, int startX, int startY, int health, StudentWorld\* world):* This constructor constructs the Insect using its parameters but also sets the depth to 1, and the direction to a random direction using the randomDirection() function. It also sets the stepsLeft variable to a random number between 2 and 10.

*virtual void doSomething():* This function first calls doSomethingInsect and returns if it returns true. It then calls the grasshopperAction function and increases the sleep ticks by 2 and returns if it returns true. It then attempts to eat up to 200 units of food from its coordinate and has a 50% of sleeping immediately after. If not, it sees if it has any stepsLeft. If not, it changes its direction to a random direction and resets the stepsLeft to a random int between 2 and 10. It then attempts to move forward in its direction and if it does, it lowers stepsLeft and calls noLongerStunned. If it can’t move, it calls setStepstoZero. If it reaches the end of the function, it increases the sleep ticks by 2.

*virtual bool grasshopperAction() = 0:* This pure virtual function is used so that the baby grasshoppers and adult grasshoppers can do different things in the same part of the grasshopper’s doSomething function. It is pure virtual as the implementations have almost nothing in common.

*int chooseSteps() const:* This function returns a random int between 2 and 10.

*int returnStepsLeft() const:* This function returns the stepsLeft variable.

*void lowerStepsleft():* This function decrements stepsLeft by 1.

*void setStepstoZero():* This function sets the stepsLeft variable to 0.

*void setSteps():* This function sets stepsLeft to a random int between 2 and 10.

**adultGrasshopper: public Grasshopper**

*adultGrasshopper(int startX, int startY, StudentWorld\* world):* This constructor constructs the Grasshopper using the parameters but also passes IID\_ADULT\_GRASSHOPPER as the image ID and 1600 as the health.

*virtual bool grasshopperAction():* This function starts off by initializing an int variable to a random int between 0 and 1. If it is 0, it attempts to bite any enemies on its coordinate. If it does, it returns true. Else, it initializes another int variable to a random number between 0 and 9. If this number is 0, it tried 10 times to jump to an open coordinate within a radius of 10 of itself. If it does jump, it returns true. If the function reaches the end, it returns false.

*virtual void getStunned():* This virtual function simply returns, as adult grasshoppers cannot be stunned.

*virtual void getPoisoned():*This virtual function simply returns, as adult grasshoppers cannot be poisoned.

*virtual void getBitten(int amount):* This function used the Insect class’s implementation of the function, but then bites a random actor on its coordinate with a bite of 50 damage if a random int between 0 and 1 is initialized to 0.

**babyGrasshopper: public Grasshopper**

*babyGrasshopper(int startX, int startY, StudentWorld\* world):* This constructor constructs the Grasshopper using the parameters but also passes IID\_BABY\_GRASSHOPPER as the image ID and 500 as the health.

*virtual bool grasshopperAction():* This function checks to see if the grasshopper has at least 1600 hitPoints. If so, it adds an adult grasshopper to the simulation at its position, sets the baby Grasshopper to dead and adds 100 units of food to the simulation. The function also returns true. If not, the function returns false.

**Ant: public Insect**

*Ant(int antID, int imageID, int startX, int startY, StudentWorld\* world, Compiler\* compiler):* This constructor constructs the Insect using its parameters but also sets the depth to 1, the direction to a random direction, and the health to 1500. It also sets the member variable Compiler\* m\_compiler to compiler, the booleans wasBitten and wasBlocked to false, and the ints lastRandom and instructionCounter to 0.

*virtual void doSomething():*This function first calls doSomethingInsect and returns if it returns true. It then attempts to follow up to 10 commands, which are fetched from the compiler. If it cannot fetch a command it dies and returns and 100 units of food is added to the simulation. If the command changes the state of the game, it immediately returns. The instruction counter is changed according to the command. If the ant moves, wasBitten and wasBlocked are set to false. If it is blocked from moving, wasBlocked is set to true.

*virtual bool isEnemy(int colony) const:* If colony is equal to the antID variable, it returns false. Else, it returns true.

*virtual void getBitten(int amount):* This virtual function performs the Insect implementation but then sets wasBitten to true.

*int returnPheromoneImageID():* This function refers to its antID variable and returns the imageID for the pheromone it may create during the doSomething function due to an appropriate command.

**How I Tested My Classes**

**Pebble**

My Pebble class was essentially tested in the fact that no Actors moved through the pebble during the simulation. The pebble also did not move, as it was not supposed to.

**Water/Poison**

I tested these classes, otherwise known as the TriggerablePool derived classes by having the compiler output “STUNNED!” or “POISONED!” whenever an appropriate object stepped on it along with the coordinate it was stunned at. This was useful in verifying that they were stunning or poisoning appropriate objects and not adult grasshoppers or other objects.

**Anthill**

I tested my Anthill class by having it print out its health at every tick in the doSomething function. This ensured that its health was being decrements by one each tick and by 1500 when it birthed an ant. I also had it print out “DEAD!” when it died, where I saw that it disappeared from the simulation. I also ensured that it was birthing the correct ants and eating food each tick by having the ants die after being birthed. I saw that a food object was created and disappeared the next tick.

**Food**

I tested the Food object by having it output its health at every tick, where it was constant unless an Insect ate it or picked it up. I also had it print out its location at every tick so I knew that there was no more than one Food object at each coordinate. I also went through the simulation tick by tick to make sure dying Insects were dropping food.

**Baby Grasshopper**

I tested the baby grasshopper class in many different ways. I made sure that no grasshoppers were moving through pebbles in any way. Also I went through tick by tick by using the F key and the space bar to make sure they were moving once every three ticks. When they stepped on a pool of Water I made sure that they were on the same spot for 4 ticks but then moved off and were not infinitely stunned. I also had the getPoisoned function print out “OUCH!” and the grasshopper’s health, where I saw that it was decremented by 150. I temporarily made the grasshopper able to eat 1600 units of food to make sure it was molting into adult grasshoppers and saw that food was being dropped and eaten by the resulting adult grasshopper. I also noticed that after stepping on food, sometimes the baby grasshoppers would rest, which is good as they have a 50% chance of sleeping after eating.

**Adult Grasshopper**

I tested the Adult Grasshopper class with similar methods as the baby Grasshopper. I made the grasshopper print out “OUCH!” when bitten, but also print out “TAKE THIS!” when it bit back, to which I saw that around every other time it would bite back and not bite itself. I also saw that it would change positions dramatically every once in a while, due to it jumping. I also noticed it would never be poisoned as the getPoisoned function never printed out “OUCH!”. The same goes for being stunned as it never halted when stepping on water. I also noticed that after stepping on food, sometimes the adult grasshoppers would rest, which is good as they have a 50% chance of sleeping after eating.

**Ant**

Testing the Ant class was complex. Most of it consisted of making output statements and corresponding with the bug file to make sure it was going in order and smoothly. In terms of testing it being bit, poisoned, or stunned. I used the previous methods as for the other insects where I used output statements to ensure the code was entering certain conditions and flowing correctly. As the command decipherer is very long and drawn out, I had to use the various ant compilers to test them out thoroughly, not just the USCAnt.bug. After realizing many critical errors, the Ant class works smoothly no matter what compiler is used.

**Pheromone**

I tested the Pheromone class out by using different ant compilers. I noticed that different ants output different colors of pheromones, which is behaving as it should. I also noticed that ants going near a pheromone and facing it always stepped on the pheromone grid point (according to the compiler) unless it was another colony’s pheromone. Other than this, I noticed that the pheromones never moved or output anything. They were unable to be poisoned and were not dangerous. The main testing of the pheromone’s was confirming their presence and relativity to their colony ants.

**Assumptions**

Most of my questions/concerns were answered by Carey, the TAs, or the spec. Thus, I believe I made very little to no assumptions.

**Unfinished Functionality**

None. (At least I hope so)