**CS 32 Project 3**

**High-Level Description**

***StudentWorld***

***virtual int init()***

This function was defined because it was required by the spec

***virtual int move()***

This function was defined because it was required by the spec

***virtual void cleanUp()***

This function was defined because it was required by the spec

***Actor\* getFoodAt(int x, int y) const***

This function will return a pointer to the food object at the indicated position, or a nullptr if no food object exists there. It was defined here because actors need to be able to ask StudentWorld where food exists. This function is not virtual because no classes are derived from StudentWorld.

***Actor\* getPheromoneAt(int x, int y, int colony) const***

This function will return a pointer to the pheromone object of the indicated colony at the indicated position, or a nullptr if no pheromone object of the indicated colony exists there. It was defined here because actors need to be able to ask StudentWorld where pheromone exists. This function is not virtual because no classes are derived from StudentWorld.

***void poisonAll(int x, int y)***

This function will poison all actors at the indicated position. It was defined here because poison objects need to be able to ask StudentWorld to poison all objects at their position. This function is not virtual because no classes are derived from StudentWorld.

***void stunAll(int x, int y)***

This function will stun all actors at the indicated position. It was defined here because waterpool objected need to be able to ask StudentWorld to stun all objects at their position. This function is not virtual because no classes are derived from StudentWorld.

***void biteEnemyAt(Actor\* me, int colony, int biteDamage)***

This function will randomly pick an actor at Actor me’s location to bite with the indicated biteDamage. It was defined here because insects need to be able to bite other insects. This function is not virtual because no classes are derived from StudentWorld.

***void jumpSomewhere(int x, int y, Actor\* jumper)***

This function will randomly find a location to jump to within a circle of radius 10 and then move Actor jumper there. It was defined here because adultGrasshoppers need to have a jumping functionality. This function is not virtual because no classes are derived from StudentWorld.

***void addFood(int x, int y, int amt)***

This function will either allocate a new food object at the indicated position with the indicated amount, or add the indicated amount to the food object already existing there. It was defined here because Actors need to be able to add food to various positions on the grid. This function is not virtual because no classes are derived from StudentWorld.

***void eatFood(Actor\* eater, int x, int y)***

This function will allow Actor eater to eat the food object at the indicated position. It was defined here because Actors need to be able to eat from food objects on the grid. This function is not virtual because no classes are derived from StudentWorld.

***void addGrasshopper(int , int y)***

This function will convert the babyGrasshopper at the indicated position into an adultGrasshopper. It was defined here because babyGrasshoppers need to be able to grow into adultGrasshoppers. This function is not virtual because no classes are derived from StudentWorld.

***void addAnt(int x, int y, int colony)***

This function will add an ant of the indicated colony at the indicated position. It was added here because anthills need to be able to give birth to new ants. This function is not virtual because no classes are derived from StudentWorld.

***void addPheromone(int x, int y, int colony)***

This function will add a pheromone from the indicated colony at the indicated position. It was added here because ants need to be able to create new pheromone objects. This function is not virtual because no classes are derived from StudentWorld.

***bool canMoveTo(int x, int y)***

This function will determine whether it is possible to move to the indicated position. It was added here because all insects need to determine whether it is possible to move to various positions. This function is not virtual because no classes are derived from StudentWorld.

***bool checkJumpSpace(int x, int y)***

This function will determine whether there is an empty position to jump to within a circle of radius 10. It was added here because adultGrasshoppers need to have a jumping functionality. This function is not virtual because no classes are derived from StudentWorld.

***bool isEnemyAt(int x, int y, int colony)***

This function will allow ants of the indicated colony to know whether there are any enemies at the indicated position. It was added here because ants will sometimes need this information for an if command. This function is not virtual because no classes are derived from StudentWorld.

***bool isDangerAt(int x, int y, int colony)***

This function will allow ants of the indicated colony to know whether there are any dangerous objects at the indicated position. It was added here because ants will sometimes need this information for an if command. This function is not virtual because no classes are derived from StudentWorld.

***bool isAntHillAt(int x, int y, int colony)***

This function will allow ants to know whether their colony’s anthill is at the indicated position or not. It was added here because ants need to know where their anthill is. This function is not virtual because no classes are derived from StudentWorld.

***Actor***

***virtual void doSomething() = 0***

This function was required by the spec. It allows each actor to execute its actions. This function was defined as pure virtual because each actor will have its own unique doSomething.

***virtual void getBitten(int amt)***

This function will reduce the actor’s health by amt. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because for actor’s that cannot be bitten the function will be redefined to simply return.

***virtual void getPoisoned()***

This function will poison the actor. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because for actor’s that cannot be poisoned the function will be redefined to simply return.

***virtual void getStunned()***

This function will stun the actor. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because for actor’s that cannot be stunned the function will be redefined to simply return.

***virtual void resetMovement()***

This function will reset an actor’s movement so that it will do something for the current tick. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because for actor’s that never move, the function simply returns.

***virtual void changeFood(int amt)***

This function will change the amount of food in the food object by the indicated amount. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because for actor’s that are not food, the function simply returns.

***virtual void updateEnergy(int amt)***

This function will change the amount of hit points by the indicated amount for any objects that have hit points. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because for actor’s that do not have hit points the function simply returns.

***virtual bool isEdible()***

This function will return whether the actor is a food object or not. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because the function needs to return false for all objects except food objects.

***virtual bool isPheromone(int colony)***

This function will return whether the actor is a pheromone object of the indicated colony or not. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because the function needs to return false for all objects except the correct pheromone object.

***virtual bool isEnemy(int colony)***

This function will return whether the actor is an enemy of an ant from the indicated colony. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because the function will return either true or false depending on the actor.

***virtual bool isDangerous(int colony)***

This function will return whether the actor is dangerous to an ant from the indicated colony. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because the function will return either true or false depending on the actor.

***virtual bool isMyAntHill(int colony)***

This function will return whether the actor is an anthill to an ant from the indicated colony. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because the function will return either true or false depending on the actor.

***virtual bool isPoison()***

This function will return whether the actor is a poison object. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because all objects will return false except poison objects.

***virtual bool didIMove()***

This function will return whether an insect already completed its move for the current tick. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because certain objects such as pebbles never do anything.

***virtual bool blocksMovement()***

This function will return whether the actor blocks movement, or simply put is a pebble. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because all objects except pebbles will return false.

***bool amIAlive()***

This function will return whether the actor is alive or not. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because certain objects such as pebbles never die.

***virtual int getColony()***

This function will return the colony number of the ant, or -1 if the actor is not an ant. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because different actors will return different values.

***virtual void getFood(int &amt)***

This function will set amt to the current amount of food that the object holds. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because for actor’s that never hold food the function simply returns.

***virtual int getEnergy()***

This function will return the current amount of hit points for the object. It was defined in Actor so that StudentWorld can call it on any actor. I made it virtual because for actor’s that do not have hit points the function simply returns some constant value.

***Pebble***

***virtual void doSomething()***

This function will execute the pebble’s doSomething, which will simply do nothing and return. It was defined in Pebble, so that Pebble would not be an abstract base class and the program could create instances of the pebble class. I made it virtual so that Pebble’s doSomething could be called through an Actor pointer.

***virtual bool blocksMovement()***

This function will return true because actors cannot move onto a pebble object. It was defined in Pebble, so that it would specialize the Actor’s version of the function which returned false. I made it virtual to enable polymorphism.

***EnergyHolder***

***virtual int getEnergy()***

This function will return the number of hit points of the object. It was defined in EnergyHolder, so that it would extend to all derived classes such as food, pheromone, ant, etc. I made it virtual to enable polymorphism and to specialize the Actor’s version of the function.

***virtual void getFood(int &amt)***

This function will set amt to the amount of food the object currently holds. It was defined in EnergyHolder, so that it would extend to all derived classes such as food, pheromone, ant, etc. I made it virtual to enable polymorphism and to specialize the Actor’s version of the function.

***virtual void updateEnergy(int amt)***

This function will change the object’s hit points by the specified amount. It was defined in EnergyHolder, so that it would extend to all derived classes such as food, pheromone, ant, etc. I made it virtual to enable polymorphism and to specialize the Actor’s version of the function.

***virtual void changeFood(int amt)***

This function will change the object’s food by the specified amount. It was defined in EnergyHolder, so that it would extend to all derived classes such as food, pheromone, ant, etc. I made it virtual to enable polymorphism and to specialize the Actor’s version of the function.

***void addFood(int amt)***

This function will add food to the object’s location by either allocating a new food object or adding the indicated amount to an existing food object. It was defined in EnergyHolder, so that it would extend to all derived classes such as food, pheromone, ant, etc. I did not make it virtual because only other EnergyHolder objects would ever need to call this function.

***int pickupFood(int amt)***

This function will pickup the indicated amount of food from the food object at the object’s current location. It was defined in EnergyHolder, so that it would extend to all derived classes such as food, pheromone, ant, etc. I did not make it virtual because only other EnergyHolder objects would ever need to call this function.

***int pickupAndEatFood(int amt)***

This function will pickup and eat the indicated amount of food from the food object at the object’s current location. It was defined in EnergyHolder, so that it would extend to all derived classes such as food, pheromone, ant, etc. I did not make it virtual because only other EnergyHolder objects would ever need to call this function.

***Food***

***virtual void doSomething()***

This function will executes Food’s doSomething, which simply checks whether the food object has reached 0 capacity and needs to be deleted or not. It was defined in Food because each object needs to have their unique version of doSomething. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual bool isEdible()***

This function will return true because food is a food object. It was defined in food to specialize Actor’s version of isEdible, which simply returns false. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***AntHill***

***virtual void doSomething()***

This function will executes AntHill’s doSomething, which reduces its hit points by one, checks whether it has died, eats any food placed on it, and creates new ants if enough energy. It was defined in AntHill because each object needs to have their unique version of doSomething. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual bool isMyAntHill(int colony)***

This function will return whether the AntHill is an anthill of an ant from the indicated colony. It was defined in AntHill to specialize Actor’s version of isMyAntHill, which simply returns false. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***Pheromone***

***virtual void doSomething()***

This function will execute Pheromone’s doSomething, which reduces its hit points by one and checks whether it has died. It was defined in Pheromone because each object needs to have their unique version of doSomething. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual bool isPheromone(int colony)***

This function will return whether the Pheromone is a pheromone of an ant from the indicated colony. It was defined in Pheromone to specialize Actor’s version of isPheromone, which simply returns false. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***TriggerableActor***

***virtual bool isDangerous(int colony)***

This function will return whether the object is dangerous to an ant from the indicated colony. It was defined in TriggerableActor because almost all classes derived from TriggerableActor are dangerous, except for ants from the same colony. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***WaterPool***

***virtual void doSomething()***

This function will execute WaterPool’s doSomething, which stuns all objects on the same position as it. It was defined in WaterPool because each object needs to have their unique version of doSomething. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***Poison***

***virtual void doSomething()***

This function will execute Poison’s doSomething, which poisons all objects on the same position as it. It was defined in Poison because each object needs to have their unique version of doSomething. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual bool isPoison()***

This function will return true, since poison is a poison object. It was defined in poison in order to specialize Actor’s version of isPoison, which simply returns false. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual void resetMovement()***

This function will reset Poison’s action tracker, which will allow it to perform an action during the current tick. It was defined in Poison in order to specialize Actor’s version of resetMovement, which simply returns. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual bool didIMove()***

This function will determine whether Poison already executed its doSomething during the current tick or not. It was defined in Poison in order to specialize Actor’s version of didIMove, which simply returns false. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***Insect***

***virtual void getBitten(int amt)***

This function will reduce the insect’s health by the indicated amount. I defined it in insect in order to specialize Actor’s version of getBitten, which simply returns. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual void getPoisoned()***

This function will poison the insect object. I defined it in insect in order to specialize Actor’s version of getPoisoned, which simply returns. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual void getStunned()***

This function will stun the insect object. I defined it in insect in order to specialize Actor’s version of getStunned, which simply returns. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual void resetMovement()***

This function will reset insect’s action tracker, which will allow it to perform an action during the current tick.I defined it in insect in order to specialize Actor’s version of resetMovement, which simply returns. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual bool isEnemy(int colony)***

This function will determine whether the insect is an enemy of an ant from the indicated colony. I defined it in insect in order to specialize Actor’s version of isEnemy, which simply returns false. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual bool isDangerous(int colony)***

This function will determine whether the insect is dangerous to an ant from the indicated colony. I defined it in insect in order to specialize Actor’s version of isDangerous, which simply returns false. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual bool didIMove()***

This function will determine whether the insect has already performed an action for the current tick. I defined it in insect in order to specialize Actor’s version of didIMove, which simply returns false. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual int getColony()***

This function will return the insect’s colony number, which is -1 for any non-ants and the colony number for any ants. I defined it in insect because all insects need to be able to get their colony number. I made it virtual because ant’s implementation of getColony is different from grasshopper’s.

***Ant***

***virtual void doSomething()***

This function will execute Ant’s doSomething, which reduces its hit points by one, checks whether it has died, checks whether it is stunned, and then processes the commands. I defined it in Ant because each object needs to have their unique version of doSomething. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual bool isEnemy(int colony)***

This function will determine whether the ant is dangerous to an ant from the indicated colony. I defined it in ant in order to specialize Insect’s version of isEnemy which simply returns true. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual int getColony()***

This function will return the colony number of the ant. I defined it in ant in order to specialize Insect’s version of getColony which simply returns -1. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual bool moveForwardIfPossible()***

This function will determine whether the ant can move one spot forward based on the direction it is currently facing. I defined it in ant in order to specialize Actor’s version of moveForwardIfPossible, which simply returns false. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual void getBitten(int amt)***

This function will reduce the ant’s hit points by the indicated amount. I defined it in ant in order to allow the functionality where the ant can keep track of whether it was bit or not. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***Grasshopper***

No public member functions for this class

***BabyGrasshopper***

***virtual void doSomething()***

This function will execute BabyGrasshopper’s doSomething, which reduces its hit points by one, checks whether it has died, checks whether it is stunned, checks whether it has enough energy to transform into an AdultGrasshopper, attempts to eat food, move, and then go to sleep. I defined it in BabyGrasshopper because each actor needs their own version of doSomething. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***AdultGrasshopper***

***virtual void doSomething()***

This function will execute AdultGrasshopper’d doSomething, which which reduces its hit points by one, checks whether it has died, checks whether it is stunned, attempts to bite any enemies, attempts to jump to a square, attempts to eat food, moves, and then go sleep. I defined it in AdultGrasshopper because each actor needs their own version of doSomething. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual void getPoisoned()***

This function will simply return because AdultGrasshopper cannot be poisoned. I defined it in AdultGrasshopper in order to specialize Insect’s version of the function, which actually reduced the actor’s health. I made it virtual to enable polymorphism and allow Actor pointers to access it.

***virtual void getStunned()***

This function will simply return because AdultGrasshopper cannot be stunned. I defined it in AdultGrasshopper in order to specialize Insect’s version of the function, which actually stunned the actor. I made it virtual to enable polymorphism and allow Actor pointers to access it.

**Unfinished Functionality**

As far as I know, my program exhibits all functionality required by the spec.

**Design Decisions and Assumptions**

1) I decided that should any actors be created during a tick, that they would only doSomething during the next tick and not during the current tick.

2) I decided that all actors would have a colony number that could be retrieved, but for any actors that were not ants the colony number would be -1.

3) I decided that every actor would maintain a pointer to the StudentWorld object.

4) I decided that StudentWorld would maintain an array of all Compiler objects and that any new ants allocated would be given a pointer to their respective compiler object.

5) I decided to use a 2D array of linked lists as the internal data structure.

6) It was not specified when to remove dead actors, so I decided to remove any dead actors after all objects were given a chance to doSomething and included a check to make sure no dead actors would be able to doSomething.

7) The amount of degree precision necessary for properly finding all spaces to jump to in a circle of 10 radius was not specified, so I decided to check every 6 degrees across all 360 degrees.

8) It was not specified whether any ant commands would only be called if all necessary prerequisites were met; for example, if pickupFood would only be called when the ant is actually standing on food. Therefore all my functions include a check to ensure no bad access will happen.

8) I decided to give any actors whose hit points never changed a constant health of 100.

9) I decided to set the initial winning colony to -1, to indicate that no colony had met the prerequisites to be considered a winer yet.

**Class Testing**

***StudentWorld***

Since StudentWorld is essentially responsible for all the major functionality of the simulation, I basically tested the class by checking to make sure that there were no runtime errors throughout the duration of the simulation. I tested the simulation by running various scenarios such as having varying amounts of different actors, making sure the game ended properly, and checking that any initialization errors were properly reported.

I also tested the class by making sure that each function had any checks in case bad values were passed in to it or in case it was asked to do something invalid. For example the biteEnemy function has to make sure it will not bite the actor that called the function, and any find functions will properly iterate through the entire linked list at any point. Thus I basically tested this class by running the simulation a bunch of times.

***Actor***

Actor is responsible for allowing polymorphism, and ensuring that StudentWorld can get any necessary information it may need from any actor. I tested the class by making sure that it had all the necessary functions for StudentWorld to have proper functionality, but since Actor is an Abstract Base Class I could not test the class itself. I just had to make sure that it had all the functions that I would need for later reference, and then proceed to specialize those functions in the appropriate derived class.

***Pebble***

All pebble has to do is stop actors from moving onto it and not die throughout the duration of the simulation. Thus it was very easy to test and I simply ran the simulation with various actors and checked that none of them were able to move on to any pebbles.

***EnergyHolder***

EnergyHolder is responsible for making sure all classes derived from it have hit points and can hold food. I tested it by making sure that it properly handled the health functionality for all classes and allowed classes to manipulate their food levels. I did this by placing two actors within a box of pebbles and observing whether the expected behavior occurred or not. For example a poison and ant should eventually lead to the ant dying, a food and ant should eventually lead to the food being completely eaten, and so on.

***Food***

I tested food by making sure that it was allocated whenever insects died, it was depleted whenever insects walked over it, and then new food allocation either led to the creation of a new food object or the addition of food to the current object. I did this by creating 3x3 squares within the field using pebbles and then using those squares to try and force certain behaviors. For example I would place an insect by itself to make sure that when it died food was created, I made sure food would be completely eaten, and so on.

***AntHill***

I tested AntHill by making sure it exhibited all the functionality as required by the spec. I did this by freezing the simulation and running one tick at a time to make sure the AntHill worked as it was supposed to. I also used the small squares surrounded by pebbles method to force certain behaviors to happen and then making sure those behaviors were executed properly. Additionally I used breakpoints to make sure that the AntHill’s energy was being properly modified for each situation.

***Pheromone***

I tested pheromone by making sure each ant correctly created a pheromone from its colony, and that pheromone would disappear after its health was completely depleted. I did this by modifying the bugs files to have the ants move in random directions and constantly drop pheromones, so that I would be able to observe the behavior of the pheromones. I then simply watched the simulation to make sure everything was working properly.

***TriggerableActor***

This class was not responsible for anything too important in of itself, and was created more for easier hierarchal comprehension of how the classes worked. Therefore by simply making sure that the classes derived from TriggerableActor worked, I was able to ensure that TriggerableActor worked.

***WaterPool***

I tested WaterPool by simply making sure that it successfully stunned insects in a manner according to the spec. I did this by placing different insects and a waterpool in a square surrounded by pebbles, and made sure that the waterpool was stunning the insects. I also tested that the waterpool would not stun the same insect over and over again.

***Poison***

I tested poison in a very similar manner to poison, by making sure that it successfully poisoned insects in a manner according to the spec. Once again I placed a poison in a square with insects and made sure that after enough times walking over the poison the insect would eventually die. I also included a check to make sure that poison would only poison insects once per tick.

***Insect***

In general for all insects, I had to use a variety of methods to test them. I first ran a full simulation using Smallberg’s version and mine side by side to first determine if there were any obvious differences. I did this multiple times for various fields. Then I created multiple squares within the grid in order to force certain scenarios such as biting or picking up food, to make sure they were occurring correctly as well. Finally for more obscure scenarios I created instances of the classes and manually forced their behavior to make sure it was working properly.

I used this three prong analysis to try and catch any bugs, since there are many different ways bugs could emerge. Specifically for the insect class I was testing to make sure that all the behavior I abstracted to this class was working properly, so that when I would test the individual classes I would just have to check their doSomething.

***Ant***

I tested ant in a very similar process to insect, but the majority of my focus was on ant’s execution of doSomething. Furthermore since the unique part of Ant’s doSomething is its command processing, that is what I was testing. I did this by creating instances of the ant class and manipulating the instance myself. I attempted to test every command, and also tried to include checks for any bad values that could be passed in.

***Grasshopper***

I also tested Grasshopper in a similar way to insect, but again my focus was on behavior unique to grasshopper such as moving and eating food. I tested this by running simulations with grasshoppers, and checking each tick to make sure everything was working properly. I also spent time going through the code myself and trying to think if there were any points where bugs could emerge and proactively patching them.

***BabyGrasshopper***

I tested BabyGrasshopper in a similar way to ant, where the majority of my focus was on its doSomething function and the unique behavior within it. For babyGrasshopper this includes the transformation to adultGrasshopper functionality. I tested this by simply creating a field with many babyGrasshoppers and food and making sure that they correctly transformed.

***AdultGrasshopper***

I tested AdultGrasshopper in a similar way to ant, where the majority of my focus was on its doSomething function and the unique behavior within it. For AdultGrasshopper this includes the biting capability and the jumping functionality. I tested this by simply running the simulation and specifically for biting I also created instances of the AdultGrasshopper