class Actor

virtual void doSomething()=0;

To iterate through the STL structure of Actors, every Actor needs to have a method to do something. Each type of actor does something different, and instances of Actors shouldn’t be created, so doSomething() is pure virtual.

StudentWorld\* getStudentWorld() const;

This method is not virtual because all Actors access its corresponding StudentWorld in the same way, and it is const because the method is only a getter. Most actors will need to communicate with the StudentWorld to get information about other actors.

bool moved() const;

This method checks whether an Actor has moved already. Every Actor uses this functionality the same way, so it is not virtual. Checking whether an Actor has moved ensures that Actors don’t move twice within a single tick.

void setMoved(bool b);

Used to set whether an Actor has moved. After an Actor moves, this allows the status to be set to moved. After a tick, every Actor is reset to a status of not moved. This is a simple setter method, so it is not virtual.

virtual void getBitten(int amount);

This method is in the Actor class so that bites can be applied easily to the STL structure of actors. Different actors respond to being bit differently (BabyGrassHoppers can be bit, Food can’t, etc). Therefore, the method is virtual. By default, Actors do not respond to being bitten (nothing happens). However, certain insects are affected, in which this method is overridden.

virtual bool blocksMovement() const;

Used to check whether a certain actor blocks movement. This is a property every Actor has (either does or doesn’t). It is virtual because some Actors block movement, and others don’t. By default, Actors do not block movement.

virtual void getPoisoned();

This method is in the Actor class so that poison can be applied easily to the STL structure of actors. Different actors respond to being poisoned differently (BabyGrassHoppers can be poisoned, Food can’t, etc). Therefore, the method is virtual. By default, Actors do not respond to being poisoned (nothing happens). However, certain insects are affected, in which this method is overridden.

virtual bool isEdible() const;

Used to check whether a certain actor is edible. This is a property every Actor has (either is or isn’t). It is virtual because some Actors are edible, and others aren’t. By default, Actors are not edible. So far, only Food is edible.

virtual bool isPheromone(int colony) const;

Used to check whether a certain actor is of the Pheromone class. This method makes it easy to find Pheromones from the STL structure of Actors. By default, classes are not pheromones. This method is virtual because the Pheromone class needs to override and signify that it is a pheromone.

virtual void getStunned();

This method is in the Actor class so that stuns can be applied easily to the STL structure of actors. Different actors respond to being stunned differently (BabyGrassHoppers can be stunned, Food can’t, etc). Therefore, the method is virtual. By default, Actors do not respond to being stunned (nothing happens). However, certain insects are affected, in which this method is overridden.

virtual bool isAntHill(int colony) const;

Used to check whether a certain actor is of the Anthill class and whether it is of a certain colony. This method makes it easy to find ally Anthills from the STL structure of Actors. By default, classes are not Anthills. This method is virtual because the Anthill class needs to override and signify that it is an Anthill.

virtual bool isEnemy(int colony) const;

Used to check whether a certain actor is an Enemy to a certain colony. All actors are either an enemy or not an enemy; this method is in the Actor class so that enemies can easily be found from the STL structure of actors. Some classes of objects are enemies, while some are not. By default, Actors are not enemies, but some are. Therefore, the method is virtual.

virtual bool isDead() const;

Used to check whether a certain actor is dead. This makes it easy for StudentWorld to sort through the dead actors after a tick. By default, Actors are not able to be dead. However, certain classes of objects can die, therefore the class is virtual.

virtual bool isDangerous(int colony) const;

Used to check whether a certain actor is dangerous to a certain colony. All actors are either dangerous or not; this method is in the Actor class so that dangerous actors can easily be found from the STL structure of actors. By default, Actors are not dangerous, but some are. Therefore, the method is virtual.

class Pebble:public Actor{

virtual void doSomething()

Pebbles do nothing during a tick. However, this method is needed because it needs to be defined in order for Pebble to be a non-abstract class.

virtual bool blocksMovement() const;

Overrides the default setting that Actors do not block movement. Pebbles block movement.

}

class EnergyHolder:public Actor{

virtual bool isDead() const;

Overrides the default setting that Actors do not die. Because energy holders have energy and lose energy every turn, an energy holder is dead when energy reaches 0.

virtual void doSomething()

Every tick, energy holders lose one unit of energy. Different types of energy holders do more things as well.

void addFood(int amount)

Allows an amount of food to be added to the current grid location of this energy. This is something that many energy holders do. However, this action is identically performed by energy holders, so the method is not virtual.

int getEnergy() const

Gets the amount of energy an energy holder currently has. Is not virtual, because it is just a simple getter method.

void updateEnergy(int amount);

Increases or decreases the amount of energy an energy holder current has. Is not virtual, because changing amount of energy is the same action for all energy holders.

int pickupAndEatFood(int amt);

Removes a certain amount of food from the grid location of this energy holder, and eats it, adding to its energy. Returns the amount of food picked up (if the amount of food requested exceeded amount of existing food, only existing amount of food eaten and returned). All energy holders do this the same way, so the method is not virtual.

    virtual bool becomesFoodUponDeath() const;

Indicates whether an energy holder drops food after its energy level hits 0. This lets the StudentWorld know whether to add food to the grid after this object’s death. By default, energy holders do not become food after death. However, some energy holders do, such as Insects. Therefore, the method is virtual so it can be overridden.

};

class Food:public EnergyHolder{

virtual void doSomething()

This method is overridden because unlike most energy holders, food does not decrease its energy every turn. It does nothing during a tick.

virtual bool isEdible() const

This method is overridden because unlike most actors, food is edible.

};

class Pheromone:public EnergyHolder{

virtual bool isPheromone(int colony) const;

This method is overridden because unlike most actors, this energy holder is a pheromone.

void increaseStrength()

Pheromone’s energy level is increased by 256, or until a capped energy level of 768. This method is not virtual because it is unique to the Pheromone class, which is also not a base class.

};

class TriggerableActor:public Actor

{

virtual bool isDangerous(int colony) const;

Overrides the default setting that Actors are not dangerous. TriggerableActors are dangerous by default.

};

class Poison:public TriggerableActor{

virtual void doSomething();

Every turn, objects of this class attempt to poison every Actor (other than itself) at its grid location. It does this by communicating with the StudentWorld. Some actors are affected by the poison, some are not. Actors can only be poisoned one time after stepping into the poison’s location; it can only be poisoned again after stepping off the location and stepping back in.

};

class Pool:public TriggerableActor{

virtual void doSomething();

Every turn, objects of this class attempt to stun every Actor (other than itself) at its grid location. It does this by communicating with the StudentWorld. Some actors are affected by the stun, some are not. Actors can only be stunned one time after stepping into the poison’s location; it can only be stunned again after stepping off the location and stepping back in.

};

class AntHill:public EnergyHolder{

virtual void doSomething()

Every turn, an anthill attempts to pick up any food at its location and eat it, up to 10000 units of food. An anthill also gives birth to a new ant if it has at least 2000 units of energy at the moment.

void giveBirth()

The anthill creates a new ant of its corresponding colony and puts it in the Student World. This process consumes 1500 units of the anthill’s energy. This method is not virtual because it is unique to AntHill, and because AntHill is not a base class.

virtual bool isAntHill(int colony) const;

This method is overridden because unlike most Actors, this energy holder is an AntHill. This method indicates whether an instance of AntHill is of a certain colony.

};

class Insect:public EnergyHolder{

void inFront(int& x, int& y);

This method updates the given x and y paremeters to hold the x,y location one space ahead in the insect’s direction. This method reduces code redundancy and is used to check whether an insect can or should move forward. This method is not virtual because every insect moves forward in its direction the same way.

int getSleep() const;

Returns how many more turns an insect is stunned/sleeping. All insects spend some of their time sleeping/stunned. This is a simple getter method, so it is not virtual.

virtual bool isEnemy(int colony) const;

This method is overridden because unlike most Actors, insects can be enemies. By default, insects are enemies. However, some insects, such as Ants, are only sometimes enemies, depending on colony number. Therefore, this method is virtual.

bool moveInDirection()

Moves the insect in the direction that it is currently facing. This method is not virtual because all insects move forward in the same way.

void incrementSleepBy(int turns);

Updates the amount of sleep/stun turns left for an insect. This method is not virtual because sleeping/stunned for a certain amount of turns is a shared property among insects.

virtual bool isDangerous(int colony) const;

This method is overridden because unlike most Actors, insects can be dangerous. By default, insects are dangerous. However, some insects, such as Ants, are only sometimes dangerous, depending on colony number. Therefore, this method is virtual.

virtual void getPoisoned();

This method is overridden because unlike most Actors, insects can be affected by poison. By default, insects lose 150 energy points from being poisoned. However, not all insects are affected the same way. Therefore, this method is virtual.

virtual void getStunned();

This method is overridden because unlike most Actors, insects can be affected by stuns. By default, insects sleep for two turns when stunned. However, not all insects are affected the same way. Therefore, this method is virtual.

virtual void setTriggerStatus(bool b);

Checks whether an insect is affected by a TriggerableActor on a current square. Unlike most Actors, insects are capable of being affected by poison and waterpools. This method is used to make sure a waterpool does not stun and poison does not poison the same insect more than once while it is on the square. This method is not virtual because trigger status (whether trigger affected the insect) is a property shared by insects.

virtual void getBitten(int amount);

Applies a bite from another insect. All insects can be bit. This inflicts the indicated amount of damage, reducing this insect’s energy level by amount. This method is virtual because not all insects are affected by bites in the exact same way (ex: AdultGrassHopper may bite back).

virtual bool becomesFoodUponDeath() const;

This method overrides the default that energy holders do not drop food upon death. When insects die, 100 units of food are dropped at the location of death.

};

class Ant:public Insect{

public:

virtual void doSomething()

Every turn (in addition to decrementing energy), Ants run commands from its compiler, unique to its colony, using runCommand() and conditionTriggered() when the command is conditional. The Ant either runs up to ten commands, quitting this method once the Ant has attempted to do something (anything other ifs, gotos, and genenrating random int). This is unique to Ants; only ants behave based on the compiler.

bool conditionTriggered(Compiler::Command cmd);

This method checks whether the condition specified by the given command is true during this tick (standing on food, enemy in front, etc..). This method is not virtual because checking conditions from a compiler command is unique to Ants, and Ant is not a base class.

bool runCommand(const Compiler::Command cmd);

Runs the given command. This method returns true if the command prompted the ant to do something (other than ifs, gotos, generating random int). If the command is an if statement, the condition is processed by conditionTriggered, and then the Ant acts accordingly.

virtual bool isEnemy(int colony) const;

This method is overridden because unlike the default that Insects are enemies, Ants are only enemies to an Actor if the given colony differs from the Ant’s colony. (Grasshoppers are considered colony -1).

virtual bool isDangerous(int colony) const;

This method is overridden because unlike the default that Insects are dangerous, Ants are only dangerous to an Actor if the given colony differs from the Ant’s colony. (Grasshoppers are considered colony -1).

virtual void getBitten(int amount);

This method is overridden because in addition to being inflicted amount of damage, reducing the energy level, the Ant has to make indicate to itself that it has been previously bitten, because the compiler may ask for this in a command.

int pickupFood(int amt)

Picks up a certain amount of food from its current location and holds onto it (doesn’t eat it). This functionality is unique to Ants; other Insects do not do this (they pick up and directly eat the food). Therefore, this method is in the Ant class, and the method is not virtual because Ant is not a base class.

};

class GrassHopper:public Insect{

virtual void doSomething();

Grasshoppers sleep two out of three turns (or more if effectively stunned). Every turn, its energy is decremented. On turns that grasshoppers are awake and alive, they first attempt to perform actions unique to the type of grasshopper (doMore()). Then, if grasshopper can act after doMore(), it attempts to pick up and eat 200 units of food. If successfully eats, it has 50% chance of immediately resting. Otherwise, it attempts to move one unit in its direction. If the grasshopper has finished walking in its direction, or is blocked, the grasshopper first sets a new random direction and desired distance to move in that direction.

virtual bool doMore()=0;

Every grasshopper has certain actions that are unique to it, specified in this class. This method is pure virtual to ensure that instances of GrassHopper cannot be created; they can only be a base class for a type of grasshopper with more actions than specified in doSomething(). The method then returns whether the grasshopper should finish doSomething() based on whether the actions in this method have been successfully executed (returning true means finish turn).

};

class BabyGrassHopper:public GrassHopper{

virtual bool doMore();

Baby grasshoppers are unique in that once they reach an energy level of 1600 or greater, they are replaced by an AdultGrassHopper. This baby is set to dead (energy level of 0), 100 units of food are dropped because of death, and adult grasshopper is added with energy level of 1600 in the death’s location in StudentWorld.

};

class AdultGrassHopper:public GrassHopper{

virtual void getPoisoned();

This method is overridden because unlike most Insects, Adult grasshoppers are immune to poison. This means that when poison attempts to poison an adult grasshopper, nothing happens.

virtual void getStunned();

This method is overridden because unlike most Insects, Adult grasshoppers are immune to stuns. This means that when waterpools attempts to stun an adult grasshopper, nothing happens.

virtual void getBitten(int amount);

This method is overridden because in addition to being inflicted amount of damage, reducing the energy level, the adult grasshopper also has a 1 in 2 chance of biting back, randomly choosing an Insect in its StudentWorld location to bite, inflicting 50 units of damage.

virtual bool doMore();

Adult grasshoppers are unique in that they attempt to bite or jump. There is a 1 in 3 chance that the adult grasshopper will attempt to bite another random Insect in the same square. If the grasshopper successfully bites another Insect, the method returns true, indicating that an action was successful and that the adult grasshopper should not do more this tick. Otherwise, there is a one in ten chance that the adult grasshopper will jump(). If the jump is successful, the method returns true, indicating that an action was successful and that the adult grasshopper should not do more this tick. Otherwise, the method returns false; no action successfully performed, and grasshopper should attempt more actions during the tick.

bool jump();

The adult grasshopper attempts to jump to another open square (one without an Actor that blocksMovement()) within a 10 square radius. If there is no open square, this method returns false to indicate that it could not jump. Otherwise, it returns true to indicate a successful jump. This method is not virtual because only adult grasshoppers jump, and adult grasshopper is not a base class.

};

class StudentWorld : public GameWorld

//None of StudentWorld classes are virtual (except the ones derived from GameWorld) because no classes are derived from StudentWorld.

virtual int init();

This method loads the ant programs, compiles them, loads the field file, and dynamically allocates, based on the field file, Actors to the array of STL lists that represents the 64\*64 grid. The anthills correspond with the ant programs, taking in the corresponding compiler as a parameter.

virtual int move();

This method prompts every Actor to doSomething(). If a colony produces another ant, its score is updated. If an Actor dies, it is removed and depending on whether it dropsFood(), drops 100 units of food at spot of death. Tick number is updated. If it reaches 2000, the method returns the winner (most ants, or if there is tie, first to number of ants, or no winner if no colony reached 5 ants).

virtual void cleanUp();

Deallocates any memory (Actors) that has not been deallocated yet. Called after game is over.

Actor\* getEdibleAt(int x, int y);

Returns an edible at given x,y location. Returns nullptr if no edible at x,y location.

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

Returns pheromone of given colony at given x,y location. Returns nullptr if no instance at x,y location.

bool isAntHillAt(int x, int y, int colony) const;

Returns whether there is an anthill of given colony at x,y location.

void increaseScore(int colony);

Increments the score of given colony.

void poisonAllPoisonableAt(int x, int y);

Attempts to apply poison (call getPoisoned) to all Actors at the given x,y location.

void stunAllStunnableAt(int x, int y);

Attempts to apply stun (call getStunned) to all Actors at the given x,y location.

bool biteEnemyAt(Actor\* me, int colony, int biteDamage);

Attempts to apply bite to all enemies of Actor me of its given colony, inflicting biteDamage. Grasshoppers are considered to be in colony -1. Returns whether there was an enemy to bite.

bool canMoveTo(int x, int y);

Returns whether there is an Actor in x,y that blocksMovement() (a Pebble).

bool isDangerAt(int x, int y, int colony) const;

Returns whether there is an Actor that isDangerous() to the Insect of given colony at x,y location.

bool isEnemyAt(int x, int y, int colony) const;

Returns whether there is an Actor that isEnemy() to the Insect of given colony at x,y location.

void add(Actor\* ac, int x, int y);

Adds actor ac to x,y location.

void setDisplayText();

Updates the display to current number of ticks and colony scores.

#endif // ACTOR\_H\_

2) Design decisions

The biggest decisions were whether an action should be dealt by the StudentWorld or a certain actor class. For example, I originally wanted Poison to retrieve Actors from the StudentWorld and then individually apply the poison to each, but eventually decided to make the StudentWorld deal with the entire action through a method that stuns all actors at a certain square. I originally also had methods in StudentWorld that returned a std::vector of Actors that met a certain condition, and have the Actor calling the method sift through the list (like choosing a random bug to bite). However, I ultimately decided to let the StudentWorld deal with all of these actions (like choosing a random bug and applying the bite).

I assumed that after a BabyGrassHopper dies and an AdultGrassHopper is created, that AdultGrassHopper may immediately doSomething() during that tick.

I assumed that if two AdultGrassHoppers are in the same square and one bites another, the retaliation mechanism allows the two AdultGrassHoppers to keep on biting back and forth, if probability allows.

I assumed that Pools are considered dangerous, even though the spec does not explicitly say so.

Although most Actors cannot be poisoned, I allowed them to have a virtual method getPoisoned() that does nothing, to make applying poison easier within the StudentWorld. Same applies for pools.

3)

Actor

I did not explicitly test for Actor because it is an abstract class. However, the functionalities of Actor are used in its derived classes. I assume that if the derived classes work properly with its data members and member functions, then Actor is without a problem.

StudentWorld

StudentWorld functionality was tested through the tests described in the individual classes below. For example, if an Ant can properly bite an enemy, walk around Pebbles, and pick up or drop food, I knew that the StudentWorld methods of detecting an enemy, biting a random enemy, detecting blocked movement, detecting food, adding Actors worked. If I created a certain field or programmed an Ant and they showed up in the simulation and the display responded, I knew that the field loading, Ant compiling and display functionalities work. Therefore, I did not perform explicit StudentWorld tests. The tests listed below add up to a comprehensive test of StudentWorld.

However, one thing I had to make sure was that the winner status worked. I tweaked Anthill so that only 4 ants would be created, and put no food on the field. I made sure that the end result was no winner. Then, I put food only near one Anthill, so it eventually outputted one more Ant. I made sure that that Anthill won. Then, I put food near two Anthills, so that each eventually outputted one more Ant. I made sure that the Anthill reaching 5 ants first won. Also, I made sure that the program ended properly once I pressed enter, which indicated that the cleanup method worked.

Pebble

The only thing to test for Pebble was whether it successfully blocked movement. I created a field that surrounded an Insect and ran the simulation to make sure it never escaped. I did this with a BabyGrasshopper, AdultGrasshopper, and Ant. To test if AdultGrasshopper jumps took into account Pebbles, I made sure to surround the AdultGrasshopper with multiple layers of Pebbles, to make sure it never escaped. Then, I left holes in the barrier to see if the AdultGrasshopper would attempt to jump into them.

EnergyHolder

I ensured that each derived class lost a unit of energy every turn (except Food), by using the debugger to keep track of it. This showed that doSomething() works properly. I made sure pickAndEatFood works by creating a field with just an Insect and food. Then, I used the debugger to keep track of the energy levels of the food and the insect to make sure that energy was conserved. I made sure that when the amount of food requested was more than the amount of food that existed, only the existing amount of food was picked up and that food energy level was at 0, I also placed food one after another to make sure Insects could pick up food two squares in a row. By creating a field with just a single Ant, one with a single BabyGrasshopper, one with a single Anthill, etc, I made sure that only those that becomeFoodUponDeath become food upon death.

Food

I made sure that StudentWorld detected Food through the isEdible method by creating a field with one grasshopper and one food items, setting a breakpoint every time isEdible returned true, to make sure every Food item was detected, and only Food was detected. I also made sure that Food didn’t decrement energy level, keeping track with the debugger as well. Other interaction tests were described in EnergyHolder. Other food interactions are described in other sections. For example, an Anthill picking up food is described in the Anthill section. If those interactions with food worked, I assumed that Food was without a problem.

Pheromone

To test pheromones, I created an Ant program that simply emitted pheromones and moved and turned in random directions and set random integers when it smelled pheromone. I created a field with just an Anthill surrounded by rocks, and tweaked the Anthill energy levels so it would only create one Ant. Then, I kept track of the energy levels of the pheromones to make sure it used EnergyHolder’s doSomething(). I set a breakpoint to catch the program every time the Ant program smelled a pheromone, making sure that it only smelled pheromones in front of it when the pheromone actually existed. To make sure that pheromones properly disappear after its energy is depleted, I created a field in which I manually added a pheromone object (through StudentWorld init()), surrounded by pebbles to make sure Ants didn’t interfere with it, making suring it disappeared after 256 turns.

Then, I created a field with the same Ant program and another dummy Ant program that did nothing except move and turn and detect whether it smelled pheromone. I made sure that it never smelled pheromone because it is not of the same colony. Then, I created a field just the Anthill surrounded by rocks, but tweaked the Ant program so it wouldn’t move, emitting pheromone in the same location repeatedly. This was to make sure that when Pheromone strength was increased, it never reached past its cap of 768.

Poison

To test poison, I created a field with a single row (trapped by pebbles), in which a babygrasshopper was surrounded by poison objects in every other square. I used the debugger to keep track of the when the grasshopper was successfully poisoned. I made sure that while the grasshopper was sleeping on the poison, it did not get poisoned again, but that other poison objects could poison it after it stepped off. Then, I created a field with only two open squares: the baby grasshopper and a poison object. I kept track of when poison was successfully triggered. This was to ensure that the same poison object only poisoned a grasshopper once on the stay, but if it moved off and moved on again, the poison would trigger once again. Then, I created a field with a single row, in which the babygrasshopper was surrounded by poison objects in every square. This was to ensure that after a babygrasshopper was poisoned, slept 2 turns, and moved immediately onto another poison object, the second poison object would trigger (the status of poisoned from the first instance wouldn’t carry over after moving off of it). I repeated these tests with an ant that behave like a baby grasshopper. Then, I made a field surrounding an ant with poison objects, in which the ant program stopped moving and turned another direction whenever it smelled danger in front of it, to make sure that poison is registered as danger. I made sure that in this simulation, the ant never escaped the confines of the poison walls. Then, I created a field with a single AdultGrassHopper surrounded everywhere by poison. I made sure that the AdultGrassHopper’s energy levels were not affected, and that it moved around the field as if nothing was there.

Pool

To test pools, I created a field with a single row (trapped by pebbles), in which a babygrasshopper was surrounded by pool objects in every other square. I used the debugger to keep track of the when the grasshopper was successfully stunned. And made sure it rested exactly four turns on the stun before moving off. I made sure that while the grasshopper was sleeping on the pool, it did not get stunned again, but that other stun objects could stun it after it stepped off. I repeated this test except with the original m\_distance of babygrasshopper set to one, so that after the stun it would turn instead of move off. I made sure that the stun didn’t trigger during this turn. Then, I created a field with only two open squares: the baby grasshopper and a stun object. I kept track of when the pool was successfully triggered. This was to ensure that the same pool object only stunned a grasshopper once on the stay, but if it moved off and moved on again, the pool would trigger once again. Then, I created a field with a single row, in which the babygrasshopper was surrounded by pool objects in every square. This was to ensure that after a babygrasshopper was stunned, slept 4 turns, and moved immediately onto another pool object, the second pool object would trigger (the status of stunned from the first instance wouldn’t carry over after moving off of it). I repeated these tests with an ant that behave like a baby grasshopper. Then, I made a field surrounding an ant with pool objects, in which the ant program stopped moving and turned another direction whenever it smelled danger in front of it, to make sure that poison is registered as danger. I made sure that in this simulation, the ant never escaped the confines of the pool walls. Then, I created a field with a single AdultGrassHopper surrounded everywhere by pools. I made sure that the AdultGrassHopper’s motion was not affected, and that it moved around the field as if nothing was there.

Anthill

To make sure that the Anthill properly picked up food every turn (if there was food), I created an Ant program that would pick up food and drop it if it was on the Anthill. Using the debugger I put a breakpoint in Ant whenever it dropped food, and tracked the food to make sure that the Anthill ate the proper amount. I created a field in which there was a food object of greater than 10000 units, so that when an Ant picked it up and dropped it on the Anthill, I could make sure that only 10000 units were picked up and eaten. An ant dropping a food on an Anthill also indicated that the isAntHill() method was successfully overridden to recognize an AntHill, and that the StudentWorld method for recognizing anthills also works. I also did tests to make sure that giveBirth() worked properly, creating the right amount of ants and making sure that the score of the specific colony was updated. I created a field with an Anthill and many food objects, and used the debugger to set a break point during every birth. I made sure that after the initial five births that occur in the first five ticks, births occurred only when the AntHill had 2000 or more units of energy.

Insect/Grasshopper

I did not explicitly test these class because instances of these classes were not meant to be created. However, their functionalities were tested when testing the derived classes. I assumed this worked if its derived classes functioned properly.

Ant

This class was harder to test because there were so many commands and conditions. To make sure that the logic of interpreting the compiler worked, I first created a very simple Ant program that just moved and turned and tested it in a field with just an Anthill. Then, I created an Ant program that could also pick up food and tested it in a field with an Anthill and food. I kept on adding on like this until I tested each functionality. Often, I used emit pheromone to indicate that an if statement was met. For example, to test if the Ant could be bit properly, I programmed it to only emit a pheromone and move forward when it was bit (to get out of the way so I can see it). Then, I could place an Ant and an AdultGrasshopper in a small field (bounded by pebbles). Every time the grasshopper was on the same location as the Ant and bit it, I made sure that an Ant can be detected and properly bit, and that the Ant’s prevBitten functionality worked. And to make sure that the biting functionality worked as well, I programmed an Ant to only check whether it is standing with an enemy and bite if it does. Then, I placed the Ant with a BabyGrasshopper, so that every time the grasshopper walked over the Ant, it would be bit. I used a debugger to keep track of every time the baby grasshopper was bit. I also did this with an AdultGrasshopper and an enemy Ant program that just walked around. This way, I made sure that Ants can properly detect enemies that are on the same square. To test whether Ants can property detect enemies that are in front of it, I created a field in which an Anthill was surrounded by BabyGrassHoppers, and tweaked the BabyGrassHopper code so that they didn’t move. In the Ant program, I made the Ant turn away whenever it sensed enemies in front of it, treating it like a rock. I made sure that in the simulation, Ants never escaped the confines of the BabyGrassHopper walls. I repeated this with Ants and AdultGrassHoppers. Other tests included creating an Ant that just spun clockwise to make sure that functionality worked; same thing for counterclockwise. Furthermore, the tests I described with TriggerableActors and Pheromones showed that Ants can detect danger and pheromones as well. Tests with the Anthill showed that ants can properly detect Anthills as well as drop, detect and pick up food.

BabyGrassHopper

To make sure that the basic movements of the BabyGrassHopper worked, I placed the a BabyGrassHopper by itself in an empty field and observed its motion. I made sure that it turned or moved only two out of three turns, ensuring that the sleep functionality from its parent class worked. I watched the BabyGrassHopper die on the 500th tick, being replaced with food, showing that the EnergyHolder functionalities still worked. Then, I made a maze using pebbles to test its ability to detect pebbles. I made sure that the BabyGrassHopper turned every time it was blocked and tried moving in the new direction. The tests described in the TriggerableAction classes ensured that BabyGrassHoppers responded properly to poison and stuns. Furthermore, Tests described in Ant indicated that BabyGrassHoppers could be bit properly bit. These both showed that inheritance from the Insect class worked. To make sure that BabyGrassHoppers properly turned into AdultGrassHopper, I created a field in which a single BabyGrassHopper interacted with many food objects. Using the debugger, I kept track of how BabyGrassHoppers interacted with food, ensuring that the picking-up-and-eating functionality was properly inherited from the EnergyHolder class. Once the BabyGrassHopper reached 1600 energy points, I made sure that a new AdultGrassHopper class was created, that the StudentWorld detected the BabyGrassHopper as dead, removed it, and that in place a food object was created.

AdultGrassHopper

The main things to test about AdultGrassHopper were response to being stunned, poisoned, and bit and the jumping mechanism. To test its (in)ability to be poisoned, I created a field with a single AdultGrassHopper surrounded everywhere by poison. I made sure that the AdultGrassHopper’s energy levels were not affected, and that it moved around the field as if nothing was there. To test its (in)ability to be stunned, I created a field with a single AdultGrassHopper surrounded everywhere by stunned. I made sure that the AdultGrassHopper’s movements were not affected, and that it moved around the field as if nothing was there. To make sure that AdultGrassHoppers bite back, I created a field with wo adult grasshoppers surrounded by pebbles so that they would run into each and bite. I tweaked the code so that AdultGrassHoppers always bite and also always bite back. When the two AdultGrassHoppers ran into each other, they kept biting back and forth until one of them died, ensuring that the biting back mechanism works and happens within a single tic. To make sure that the AdultGrassHopper inherited GrassHopper’s doSomething() properly, I commented out AdultGrassHopper’s doMove() code and did the tests described in BabyGrassHopper, because without doMove(), adult grasshoppers move around and pick up food in the same way as BabyGrassHoppers. The tests described in Ant also ensured that AdultGrassHoppers properly detect and bite ants.

To test the jumping mechanism, I set a breakpoint every time an adult grasshopper jumped. Then, I made sure that the new location was within a 10 square radius. I used Pythagorean theorem to measure the distance between the new and old locations. Then, I created a field in which an adult grasshopper was surrounded by a circle of pebbles, ten pebbles deep on each side. I made sure that the adult grasshopper never jumped out. Then I created a field in which the adult grasshopper was surrounded by a circle of pebbles, nine pebbles deep on each side. I made sure that the adult grasshopper sometimes jumped out. This ensured that the adult grasshopper only jumped within a ten square radius and only to open spots. By iterating through the simulation tick by tick and using the debugger to step through individual lines of code, I also ensured that the grasshopper didn’t do anything more after jumping or after biting.