**CS32 Project 3 Report**

**Akshay Smit**

**UID: 804641231**

1. High-level description of public member functions

Class Hierarchy:

Actor

EnergyHolder

Eater

Insect

Grasshopper

AdultGrasshopper

BabyGrasshopper

Ant

Anthill

Pheromone

Impactor

Poison

PoolOfWater

Pebble

Note that the Impactor class is basically the equivalent of the TriggerableActor class in the implementation provided on the class website.

The only other big difference between this hierarchy and the implementation on the website is the additional Eater class. This class was created because Anthill objects are not Insects, but eat in exactly the same way as, for instance, Grasshoppers. In the future, if more non-insect eater objects are added to the game, the existence of this class will save unnecessary code.

Class: **Actor**

Note that the destructor is pure virtual, but it is implemented. It has an empty body. The destructor was made virtual so that this will be an Abstract class (the doSomething function was not made pure virtual: it has an empty implementation, and that is used by the Pebble class)

All Actors are constructed so that they face right. Any Actors that need to face random directions do so in the body of their respective constructors.

1. StudentWorld\* getWorld() const

This function is called by an Actor to ask its StudentWorld object to perform some action or give some information about the 64-by-64 grid. It returns a pointer to the StudentWorld object the Actor is contained in.

ii) void setDead()

Sets the status of the Actor to dead.

iii) bool isDead() const

Returns true if the Actor is dead. Otherwise returns false.

iv) virtual void getPoisoned()

As implemented in the Actor class, this function does nothing. However, many Actors can indeed be poisoned, such as Adult Grasshoppers and Ants. But not all Actors are affected by poison, such as Pebbles and Adult Grasshoppers. So the function is declared virtual, so that those derived classes which are affected by Poison can override the base class implementation and get poisoned in their own way.

v) virtual void getStunned()

As implemented in the Actor class, this function does nothing. But some Actors like Baby Grasshoppers can get stunned, but others like Adult Grasshoppers cannot be stunned. So this function is virtual to allow derived classes which can get stunned to override this implementation, and get stunned in their own way.

vi) void setDoneSomething()

Calling this function sets a boolean variable so that the StudentWorld object knows this Actor has already done something on this tick.

vi) void setNotDoneSomething()

Calling this function sets a boolean variable so that the StudentWorld object knows this Actor has not done something on this tick (this is called by StudentWorld to prepare for the next tick, since no Actor has done something at the very beginning of a tick).

vi) virtual void doSomething()

As implemented in the Actor class, this function does nothing. Actors like Pebbles do not do anything when asked to do something. Other Actors like Poison objects and Adult Grasshoppers behave very differently when asked to do something. So this function is declared virtual to allow derived classes to override it, and do the actions they specifically need to do when asked to do something.

vii) virtual bool amIWalkable()

Returns true if Actor’s current square can be occupied by another Actor. Most Actors are walkable, so this function returns true as implemented in the Actor class. But some actors like Pebbles are not walkable, so this function is virtual to allow them to override it.

ix) virtual bool canIBeEaten()

Returns true if Actor can be eaten by other Actors so they can get more hitpoints/energy. This function, as implemented in Actor class, returns false since most Actors cannot be eaten. But some actors like Food objects can be eaten, so this function is virtual to allow edible derived classes to override it.

x) virtual bool amIEnemyOfAdultGrasshopper()

Returns true if the Actor is an enemy of an Adult Grasshopper. This function returns false as implemented in Actor class, as most Actors are not enemies of Adult Grasshoppers. But as with the previous two functions, it is virtual to allow enemy classes to override it.

Note that this function, if called by any Adult Grasshopper, will return true (see the overriding implementation below in the AdultGrasshopper class). It is the StudentWorld object’s job to prevent an Adult Grasshopper from biting itself.

xi) virtual bool amIEnemyOfAnt(int colonyNum)

Returns true if the Actor is an enemy of an ant of the specified colony. Returns false as implemented in Actor class. For the same reason as the previous function, it is virtual.

xii) virtual bool amIPheromone()

Returns true if Actor is a Pheromone object. Returns false as implemented in Actor class, but is virtual for similar reasons as the previous functions.

xiii) virtual bool canIBePoisoned()

Returns true if Actor is affected by Poison. As implemented in Actor class, this returns false. It is virtual for same reasons as previous functions.

xiv) virtual bool canIBeStunned()

Returns true if Actor can be stunned. As implemented in Actor class, this returns false. It is virtual for the same reasons as previous functions.

Class: **Energy Holder**

1. virtual ~EnergyHolder() = 0

This destructor was made pure virtual because this needs to be an Abstract class. No other function could be made pure virtual, so the destructor was made pure virtual. But the destructor is implemented, and the body of the destructor is empty.

ii) int getEnergy() const

Returns how much energy the Actor has.

iii) void gainEnergy(int amt)

Increase the amount of energy the Actor has by amt.

iv) void loseEnergy(int amt)

Decrease the amount of energy the Actor has by amt. If the resulting energy is <= 0, then

the Actor’s status is set to dead.

Class: **Eater**

1. virtual ~Eater() = 0

Destructor pure virtual for same reason as the previous two classes.

ii) virtual int Eat(int amt)

Asks the StudentWorld object to remove the specified units of food from this Actor’s square. The units of food actually removed from the square are added to the Actor’s energy, and returned by this function. Baby and Adult Grasshoppers, and Anthills eat in this way.

It is virtual because some Eater objects eat differently. Ants do not eat directly from the 64-by-64 grid. They first pick up food, and then eat it at a possibly later time. So the function is virtual to allow Ant class (and other future classes) to override this implementation and eat in their own ways.

class: **Insect**

i) Insect(int imageID, int startX, int startY, StudentWorld\* world, int energy)

This constructor indicates the Actor is not sleeping, has not been stunned on the current square, and makes the Actor face a random direction

ii) void FaceRandDir()

Actor faces a random direction

Pseudocode:

Generate a random number between 0 and 3 (inclusive)

If the number was 0

set direction to up

If the number was 1

set direction to right

If the number was 2

set direction to down

If the number was 3

set direction to left

iii) bool moveInDir(Direction dir)

Actor attempts to move on square in specified direction. If the move is successful, this function indicates the Actor has not been stunned on its current square and returns true.

If the move is unsuccessful, returns false.

Pseudocode:

For each direction:

Ask StudentWorld object if square in this direction is walkable

If not, return false

Otherwise move to square in this direction

iv) void Die()

Sets the status of the Actor to dead and asks StudentWorld object to place 100 units of food on its square.

v) virtual void getBitten(int amt)

Causes Actor to lose specified amount of energy. It is virtual because some derived classes like Adult Grasshoppers bite differently. They lose energy and then they have a chance of immediately biting a random enemy. So the function is virtual to allow Adult Grasshoppers (and other future classes) to override this implementation and bite in their own way.

vi) virtual void getPoisoned()

If the Insect is affected by Poison, this function reduces its energy by 150. It is virtual because this is an overriding implementation. This function was declared virtual in Actor class.

vii) void increaseSleepOrStunTicks(int ticks)

Increases ticks to sleep/be stunned by specified number of ticks.

viii) virtual void getStunned()

If the Insect can be stunned and wasn’t stunned previously, then this function increases the ticks to sleep/be stunned by 2. It is virtual because this is an overriding implementation. This function was declared virtual in Actor class.

ix) int initialInsectActions()

Causes Insect to lose 1 hitpoint and if it dies, take the necessary action when an Insect dies. If insect was sleeping/stunned, reduces the ticks to sleep/be stunned by 1. This function does the common actions of all Insects, when they are asked to do something.

This function returns 0 if the Insect died or is sleeping/stunned. Otherwise it returns 2.

Pseudocode:

Lose 1 hitpoint of energy

If energy is non-positive

set status to dead

ask StudentWorld object to create/add 100 units of food at current square

return 0

If Insect is sleeping or stunned

Decrement turns left to sleep/be stunned

return 0

return 2

x) virtual bool amIEnemyOfAdultGrasshopper() const

Returns true. This is an overriding implementation of the virtual function declared in Actor class.

xi) bool isSleepingOrStunned() const

Returns true if ticks to sleep/be stunned is greater than 0. False otherwise. This function is private because it is only called by the Insect class member functions.

xii) virtual bool biteRandomEnemy()

As implemented in the Insect class, this simply returns false. This is because not all Insects can bite (Baby Grasshoppers cannot bite). Also, Ants and Adult Grasshoppers do not bite in the same way, since they have different enemies, and also since they do different amounts of damage to their enemies. So, when Ants and Adult Grasshoppers bite enemies, they must tell their StudentWorld object to do different things.

Class: **Grasshopper**

1. Grasshopper(int imageID, int startX, int startY, StudentWorld\* world, int hitpoints)

The constructor body chooses a random distance between 2 and 10 (inclusive), to move in the direction the Grasshopper is facing.

1. virtual void doSomething()

This function does the common actions of all insects (returning immediately if the Insect died or is sleeping/stunned), then the differing actions of all derived classes (returning immediately if Baby Grasshopper turned into Adult or Adult Grasshopper jumped/bit an enemy), and then does the following:

The Grasshopper tries to eat any food on the square. If it eats, there is a 50% chance it will go to sleep for two ticks.

Otherwise the Grasshopper tries to move in the direction it is facing. If it no longer has to move in that direction, a new direction and distance are chosen first, and then the move is attempted. If the move failed, the Grasshopper goes to sleep for two ticks. If the move succeeded, this function decreases the distance to be moved in the current direction by 1, and then the Grasshopper goes to sleep for two tick.

In the Pseudocode below, note that all of the actions from the point the Grasshopper starts to eat are performed by a private virtual function (of return type void) of the Grasshopper class called postEatActions(). The postEatActions() function performs actions that are common to all Grasshoppers.

Pseudocode:

Perform common actions of all Insects

If Insect died or is sleeping/stunned

return immediately

Perform specific actions of derived classes (calls private pure virtual function preEatActions())

If Baby Grasshopper turned into Adult or Adult Grasshopper bit enemy/jumped

return immediately

Try to eat any food on the square by asking StudentWorld object

If the Grasshopper ate

Generate random number between 0 and 1 (inclusive)

If number was 1, increase ticks to sleep by 2 and return

If move distance is 0

Face a random direction

Pick random move distance between 2 and 10 (inclusive)

Attempt to move in current direction

If the attempt failed

Set move distance to 0

Increase ticks to sleep by 2 and return

Decrement move distance

Increase ticks to sleep by 2

iii) virtual bool amIEnemyOfAnt(int colonyNum)

Returns true. This is an overriding implementation of the virtual function declared in Actor class.

Class: **BabyGrasshopper**

1. virtual bool canIBePoisoned()

Returns true. This is virtual as it is an overriding implementation of the virtual function declared in Actor class.

ii) virtual bool canIBeStunned() const

Returns true. Virtual for same reason as canIBePoisoned()

iii) virtual int preEatActions()

This is a private virtual function, which was declared pure virtual in the Grasshopper class. This function performs the specific actions of Baby Grasshoppers when asked to do something. These actions are totally distinct from the actions of Adult Grasshoppers.

The BabyGrasshopper checks if it has at least 1600 hitpoints of energy. If so, it asks StudentWorld object to create a new AdultGrasshopper at its location. The BabyGrasshopper then sets its status to dead, and asks StudentWorld object to replace it with 100 units of food.

If the BabyGrasshopper turned into an Adult, this function returns 0. Otherwise it returns 2.

Class: **AdultGrasshopper**

1. virtual void getBitten(int amt)

This is an overriding implementation of the function declared in Insect class. It calls Insect’s getBitten function and then there is a 50% chance the Adult Grasshopper will immediately bite a random enemy on its square.

Pseudocode:

Call Insect’s getBitten function

If the AdultGrasshopper is alive

Generate random number between 0 and 1 (inclusive)

If the number is 1

bite a random enemy on square (ask StudentWorld to do this)

ii) bool jumpToRandSquare()

This is a private function as it is only called by AdultGrasshopper’s member functions. It enumerates all open squares within a radius of 10 squares, and picks a random one to jump to.

Pseudocode:

Repeatedly:

If current square is walkable and within 10 squares, increment open square count

If there are no open squares

return false

Generate random number between 1 to open square count (inclusive)

Repeatedly:

If current square the open square indicated by the random number

move to the square

return true

iii) virtual bool biteRandomEnemy()

This is an overriding implementation of the private virtual function declared in Insect class. This function asks the StudentWorld object to attack a random enemy of the AdultGrasshopper on the same square.

Note that the StudentWorld object ensures the AdultGrasshopper does not attack itself. The StudentWorld object also causes the bitten enemy to lose 50 hitpoints of energy.

iv) virtual int preEatActions()

This is the implementation of a pure virtual function declared in the Grasshopper class. There is a 1/3 probability the AdultGrasshopper tries to bite a random enemy (by calling biteRandomEnemy() ). If it does bite an enemy, the function returns 1.

Otherwise there is a 1/10 probability the AdultGrasshopper tries to jump to an open square within a 10 square radius (by calling jumpToRandSquare() ). If it jumps, the function returns 1.

Otherwise the function returns 2.

Class: **Ant**

1. virtual int Eat(int amt)

This is an overriding implementation of the virtual function declared in Eater class. The ant attempts to eat any food it is carrying. If it is carrying food, it tries to increase its hitpoints by 100 (which corresponds to eating 100 units of food). If it is carrying less than 100 units of food, it increases its hitpoints by the amount of food it is carrying.

So if the Ant is carrying 0 units of food, the hitpoints are unchanged.

ii) int getColony() const

Returns this Ant’s colony number

iii) virtual void getBitten(int amt)

This is an overriding implementation of the virtual function declared in Insect class. This function first calls the Insect’s getBitten function and then sets a boolean variable so this Ant can remember it was bitten on the current square.

iv) virtual void doSomething()

This is an overriding implementation of the virtual function declared in Actor class. This function first perform the common actions of all Insects, returning immediately if the Ant died or is stunned.

Then the command specified by the instruction counter is executed (using a private member function called interpreter ). If any command attempts to change the state of the simulation (such as a failed attempt to move forward) then this doSomething function returns immediately.

A total maximum of 10 commands are executed.

Pseudocode:

Perform initial actions common to Insects (calls Insect::initialInsectActions() )

If dead or stunned, return immediately

repeatedly:

Get the command indicated by the instruction counter

Execute the command (by calling private function called interpreter )

If simulation state was changed, return immediately

Increment number of commands executed

If 10 commands have been executed, return

**NOTE**: The explanation and Pseudocode of the interpreter function are given below

v) virtual bool amIEnemyOfAdultGrasshopper() const

Returns true. This is an overriding implementation of the function declared virtual in Actor class.

vi) virtual bool amIEnemyOfAnt(int colonyNum)

Returns true if colonyNum is not equal to the Ant’s colony number. Otherwise false. This is an overriding implementation of the function declared virtual in Actor class.

vii) virtual bool canIBePoisoned() const

Returns true. This is an overriding implementation of the function declared virtual in Actor class.

viii) virtual bool canIBeStunned() const

Returns true. This is an overriding implementation of the function declared virtual in Actor class.

ix) int interpreter(Compiler::Command &cmd)

This private function takes the command specified by cmd’s opcode data member (see Compiler.h file) and executes it. The convention is that if any command changes (or attempts to change) state of the simulation, this function returns 1. Otherwise it returns 0. Here is the list of possible commands and how they are executed:

1. **moveForward** : Ant tries to move one square in specified direction. If the attempt fails, it sets a boolean to remember it was blocked from moving. Otherwise if move succeeded, it sets booleans to remember it was not bitten or blocked on the new square. Increments the instruction counter. Returns 1.
2. **eatFood** : Tries to eat 100 units of food (by calling the Eat function). Increments the instruction counter. Returns 1.
3. **dropFood** : Asks StudentWorld object to create or add units of food equal to the units of food the Ant is carrying, in the current square. Then the amount of food held by the Ant is set to 0. Increments the instruction counter. Returns 1.
4. **bite** : Tries to bite random enemy on square by calling biteRandomEnemy() . Increments the instruction counter. Returns 1.
5. **pickupFood** : If the Ant has picked up at most 1400 units of food, then the StudentWorld is asked to remove 400 units of food from the current square. The number of units actually removed from square are picked up by the Ant. If the Ant had more than 1400 units of food (say it has x units), then StudentWorld is asked to remove 400-x units. The units of food actually removed are picked up by the ant. Increments the instruction counter. Returns 1.
6. **emitPheromone**: Asks the StudentWorld object to create or add strength to a Pheromone object of the Ant’s colony on the current square. Increments the instruction counter. Returns 1 (Note that the StudentWorld object ensures a Pheromone does not have more than 768 strength)
7. **faceRandomDirection**: Faces a new random direction by calling FaceRandDir() . Increments instruction counter. Returns 1
8. **rotateClockwise**: Sets the direction of the Ant so it is rotated 90 degrees clockwise. Increments instruction counter. Returns 1
9. **rotateCounterClockwise**: Sets the direction of the Ant so it is rotated 90 degrees counterclockwise. Increments instruction counter. Returns 1
10. **generateRandomNumber**: Converts the operand1 data member of cmd to integer. If it is 0, sets the last random number to 0. Otherwise generates a random number between 1 and the number obtained from operand1, and assigns this to be the last random number. Increments the instruction counter and returns 0.
11. **goto\_command** : converts the operand1 data member of cmd to integer, and assigns it to instruction counter. This corresponds to fetching instructions from the location specified by operand1 (which will be executed the next time doSomething() calls interpreter). Returns 0.
12. **if\_command** : Checks whether the condition specified by operand1 is true, by calling a private function called commandTriggered (see below). If the condition is true, sets instruction counter to integer obtained by converting operand2 to integer. Otherwise increments the instruction counter. Returns 0.

**NOTE:** Some of the work for the above commands is done by private helper functions. But in all these cases, the working of the helper function has been fully described above.

x) bool commandTriggered(Compiler::Command &cmd)

This private function returns true if the condition indicated by operand1 of cmd is true. Otherwise returns false.

Here is a list of possible conditions and how they are checked:

1. **i\_smell\_danger\_in\_front\_of\_me**: Asks StudentWorld if there is an enemy in the square the Ant is facing
2. **i\_smell\_pheromone\_in\_front\_of\_me**: Asks StudentWorld if there is a Pheromone of this Ant’s colony in the square the Ant is facing
3. **i\_was\_bit**: Returns the boolean variable storing whether or not the Ant was bit previously on its current square
4. **i\_am\_carrying\_food**: Returns true if Ant is carrying a positive number of units of food. Otherwise returns false
5. **i\_am\_hungry**: Returns true if Ant has at most 25 hitpoints of energy. Otherwise returns false.
6. **i\_am\_standing\_on\_my\_anthill**: Asks StudentWorld for the location of the Ant’s Anthill, and returns true if Ant is at same location. Otherwise returns false
7. **i\_am\_standing\_with\_an\_enemy**: Asks the StudentWorld object to count enemies of the Ant at the current square, and returns true if there are enemies present. Otherwise returns false
8. **i\_am\_standing\_on\_food**: Asks StudentWorld if there are any eatable Actors at the current square. Returns true if there are, otherwise returns false.
9. **i\_was\_blocked\_from\_moving**: Returns a boolean variable storing whether or not the Ant was blocked from moving during its previous attempt to move from the current square.
10. **last\_random\_number\_was\_zero**: Returns true if the last generated random number was 0. Otherwise returns false.

Apart from the above functions, the Ant class contains a few more private virtual functions which manipulate boolean variables. These boolean variables are used to remember whether the Ant was blocked from moving or was bitten on its current square. These functions have trivial implementations.

Class: **Anthill**

1. virtual void doSomething()

This is an overriding implementation of the virtual function declared in Actor class. The Anthill loses 1 hotpoint of energy, setting its status to dead if resulting hitpoints are <= 0 and returning immediately.

Then the Anthill eats any available food on its square, and if it eats, the function returns immediately.

If the Anthill has >= 2000 hitpoints, it creates a new Ant of its colony.

Pseudocode:

Lose 1 hitpoint of energy

If the hitpoints <= 0

set status to dead

Attempt to eat any food on square

If food was eaten

return immediately

If Anthill has at least 2000 hitpoints of energy

Create a new Ant of this colony

Ask StudentWorld to add the Ant to simulation

**NOTE:** The checking for at least 2000 hitpoints and creation of the new Ant is done by a private function (of return type void) called createAnt(). The working of this function has been described in the above pseudocode.

Class: **Food**

1. virtual bool canIBeEaten() const

Returns true. This is an overriding implementation of the virtual function declared in Actor class.

Class: **Pheromone**

1. virtual void doSomething()

This is an overriding implementation of the function declared virtual in Actor class. This function causes the Pheromone to lose one hotpoint of energy, setting its status to dead if resulting hit points <= 0.

ii) int getColony() const

Returns the colony number of the Pheromone.

iii) virtual bool amIPheromone() const

Returns true. This is an overriding implementation of the virtual function declared in Actor class.

Class: **Impactor**

i)virtual void doSomething()

This function does the specific actions of the derived classes. It actually calls a private pure virtual function called impactActors() which performs the actions of specific derived classes.

The specific actions taken by the derived classes are as follows:

**Poison**: asks StudentWorld to poison all actors at its square

**PoolOfWate**r: asks StudentWorld to stun all actors at its square

Note that not all Actors are capable of being poisoned or stunned. Those Actors which cannot be stunned/poisoned will not be affected.

Class: **Pebble**

1. virtual bool amIWalkable() const

Returns false. This is an overriding implementation of the function declared virtual in Actor class.

Class: **StudentWorld**

**The Actors are contained in a two-dimensional array of linked lists. The linked lists contain pointers to all Actors at that location. The array has dimension VIEW\_WIDTH by VIEW\_HEIGHT. So the first index is the column, the second index is the row.**

**Compilers of the Ant colonies are contained in a vector<Compiler\*> called m\_compilers.**

**The StudentWorld object stores column numbers of Anthills in an array and the row numbers in another array.**

1. bool isSquareWalkable(int x, int y) const

Returns true if a particular square does not contain any Pebbles (or any Actors that block movement of other Actors). Otherwise returns false.

This function simply iterates through all Actors at the specified location, returning false if any Actor is not walkable (it asks each Actor whether it is walkable).

ii) bool checkEatable(int x, int y) const

Returns true if the specified square contains non-dead Actors that can be eaten (the only one being Food objects). As with the isSquareWalkable() function, this function iterates through all Actors at the location, returning true if any Actor is edible.

iii) bool PheromoneAheadOfAnt(const Ant\* caller) const

Returns true if the square in the direction the caller is facing contains a Pheromone of the relevant Ant colony. This function first determines the location of the square which the caller is facing, and then iterates through all Actors at that square, asking each one whether it is a non-dead Pheromone of the correct colony.

iv) bool DangerAheadOfAnt(const Ant\* caller) const

Returns true if the square in the direction the caller is facing contains an enemy Actor of the caller. Otherwise returns false. Note that the Actor class has a public amIEnemyOfAnt function (as explained previously) which takes as its parameter the colony number.

The DangerAheadOfAnt function determines the location of the square caller is facing, then iterates through all Actors at that square, asking each one if it is an enemy of the caller’s colony. If any Actor is an enemy, this function returns true.

v) int countAntEnemies(const Ant\* caller) const

Returns the number of enemy Actors present at the same square as caller. This function iterates through all Actors at the caller’s square, incrementing the count if any one is an enemy of the caller’s colony. It finally returns the count.

vi) int getColonyX(int colonyNum) const

Returns the column number of the specified colony’s Anthill. The StudentWorld stores column numbers of all Anthills in an array.

vii) int getColonyY(int colonyNum) const

Same as previous function, but with row numbers instead of column numbers.

viii) bool biteRandAdultGrasshopperEnemy(AdultGrasshopper\* caller)

This causes a random enemy of the caller to get bitten, causing 50 hitpoints of damage. It returns true if there was at least one enemy at the same square as the caller (in which case a randomly chosen enemy was bitten). Returns false if there were no enemies present.

**This function does not consider the caller to be an enemy of itself.**

Pseudocode:

Calls countAdultGrasshopperEnemies to determine number of enemies at the caller’s location (shown below)

If no enemies present

return false

Generate random number between 1 and enemy count (inclusive)

Repeatedly for each Actor at caller’s location:

If current Actor is not dead and is enemy of caller (and is not the caller itself)

increment counter

If the counter equals the random number

Cause the enemy Actor to get bitten, doing 50 hitpoints damage

**NOTE:** If, for instance, there are 5 enemies and the random number is 2, then the above code will attack the 2nd enemy at the square (counting from beginning to end of linked list using an iterator)

ix) bool biteRandAntEnemy(Ant\* caller)

This causes a random enemy of the caller to get bitten, causing 15 hitpoints of damage. It returns true if there was at least one enemy at the same square as the caller (in which case a randomly chosen enemy was bitten). Returns false if there were no enemies present.

Pseudocode:

Calls countAntEnemies to determine number of enemies at caller’s location

If no enemies present

return false

Generate random number between 1 and enemy count (inclusive)

Repeatedly for each Actor at caller’s location:

If current Actor is not dead and is enemy of caller

increment counter

If the counter equals the random number

Cause the enemy Actor to get bitten, doing 15 hitpoints damage

x) int eatOrPickUpFood(int x, int y, int amt)

Returns the units of food removed from the specified location, which is equal to amt if the food object at that location held more than amt units. Otherwise the food object dies, as all its units are eaten/picked up. In either case, this function returns the units actually removed from the location.

**Note that the StudentWorld class does not allow multiple non-dead food objects to be at the same location (see addOrCreateFood function below).**

Pseudocode:

Nothing to do if amt is 0

Repeatedly for each Actor at specified location:

If Actor is a non-dead edible object

If Actor has at least amt units of food

The Actor loses amt units of food

else

The Actor loses all units of food it held

Return units of food removed from specified location

xi) void poisonActorsAt(int x, int y)

This function simply iterates through all Actors at the given location and causes them to get poisoned. Not all Actors can be Poisoned, so those will not be affected (this is taken care of by the virtual getPoisoned function in Actor class)

xii) void stunActorsAt(int x, int y)

This function simply iterates through all Actors at the given location and causes them to get poisoned. Not all Actors can be stunned (such as an Ant already stunned before on current square), so those will not be affected (this is taken care of by the virtual getStunned function in Actor class)

xiii) void addOrCreateFood(int x, int y, int units)

This function creates a new Food object at location with specified units of food, if there was no non-dead Food object already present. If there was an already-present non-dead Food object, that Food object gains the specified units of food.

This function simply iterates through all Actors at the location. If it finds a non-dead Food object, that object gains the specified units of food.

If no Food object was present, a new one is created and added to the simulation.

**NOTE: As a consequence of this function, no two non-dead Food objects can have the same location.**

xiv) void addOrCreatePheromone(int x, int y, int colonyNum)

This function creates a new Pheromone object of the specified colony at location with 256 units of strength, if there was no non-dead Pheromone object of the colony already present. If there was an already-present non-dead Pheromone object of that colony, that Pheromone object gains 256 units of strength, up to a maximum of 768 units.

This function simply iterates through all Actors at the location. If it finds a non-dead Pheromone object of the specified colony, that object gains the specified units of strength. If no such Pheromone was present, a new one is created and added to the simulation. It is a Pheromone of the specified colony.

xv) void createAdultGrasshopper(int x, int y)

Creates a new AdultGrasshopper and adds it to the simulation at the specified location (by adding a pointer to the new object to the linked list at [x] [y] ).

xvi) void addAnt(Ant\* ant)

Adds an already-created Ant to the simulation at the correct location (by adding the pointer to the linked list holding Actors at that location).

**The following functions are private as they are not called outside of StudentWorld member functions:**

xvii) int initCompilers()

Creates new Compiler objects for each colony, compiles the Bugs! program of each colony, and stores compilers in a vector. If some file failed to compile, this function returns -999. Otherwise it returns 0.

Pseudocode:

Get a vector containing file names of programs

Repeatedly for each file:

Create a new Compiler

Store it in vector

Try to compile the file, returning -999 upon failure

Return 0

xviii) void makeActorsDoSomething()

Causes each Actor in the simulation to do something (calls the doSomething() function) and updates the location of each Actor in case it moved.

Pseudocode:

Repeatedly for each actor:

If the Actor has not done something this tick

Call its doSomething() function

else

consider the next Actor

Set a boolean to indicate Actor has done something this tick

If Actor’s coordinates are different from previous coordinates

Add pointer to Actor to linked list representing new coordinates

Remove pointer to Actor from linked list representing old coordinates

xix) void removeDeadActors()

This function iterates through all Actors in the simulation, removing any that are dead. For the non-dead Actors, this function prepares for the next call to doSomething() by setting a boolean to indicate the Actor has not done anything in this tick.

xx) void deleteCompilers()

This function simply deletes all Compiler objects created for the colonies.

xxi) void updateWinner()

This function checks if any colony has produced at least 6 ants and has produced more ants than any other colony. If this is the case, the winner is updated to be the colony which satisfied these conditions.

xxii) void updateDisplayText()

Creates a string which begins with “Ticks:” followed by a right-aligned 5-character quantity representing the number of ticks. This is followed by “ - ” and then the name of the zeroth ant colony, followed by “: ”, followed by a 2-character wide quantity representing the number of ants created by the zeroth colony. This is followed by two spaces, and then the details of the remaining colonies (in the same format as the zeroth colony), each separated from the details of the previous colony by two space.

This function then calls setGameStat() to update the text on the display screen.

Pseudocode:

Initialize new stringstream

Send “Ticks:” to internal storage

Set the width to 5

Send number of ticks to internal storage

Send “ - ” to internal storage

Make ‘0’ the fill character

Repeatedly for each colony:

if this colony is currently winning

send colony’s name to internal storage

send “\*: ” to internal storage

Set width to 2

Send number of ants created by colony to internal storage, followed by “ “

else

send colony’s name to internal storage

send “: ” to internal storage

Set width to 2

Send number of ants created by colony to internal storage, followed by “ “

Call setGameStatText to update display text

xxiii) virtual int init()

Initializes all data members and loads the field file. It then creates new Actors as specified in the field file and adds them to the simulation.

Pseudocode:

Set tick count to 0

Set number of ants created by all colonies to 0

Set integer variable to indicate no current winner

Create compilers for each class and compile all programs (by calling initCompilers() )

Create new Field

Get name of field file and load it

Repeatedly for each character in field file:

Determine which type of object is at the current location

Create the object and add it to linked list representing the location

Return a value indicating the game should be continued

xxiv) virtual int move()

This function causes all Actors to do something, updating their positions if they moved. It removes any Actors that died, and updates the display text. It does this by delegating work to many private functions already explained previously.

Pseudocode:

Increment tick count

Make all Actors do something and update positions if needed (by calling makeActorsDoSomething() )

Remove any dead Actors (by calling removeDeadActors() )

Update display text (by calling updateDisplayText() )

If 2000 ticks have elapsed

Return value indicating no winner if there is no winner

Set the winner to the winning colony (by calling setWinner )

Return value indicating there is a winner

xxv) virtual void cleanUp()

Deletes all Actors in the simulation and removes their pointers from the linked lists. It then deletes the compiler objects (by calling deleteCompilers() ).

2. **All functionality has been implemented. There are no known bugs.**

3. **Design decisions:**

1. When a BabyGrasshopper turns into an AdultGrasshopper, the StudentWorld object will ask the AdultGrasshopper object to do something on the current tick. The spec did not clarify whether the AdultGrasshopper should wait until the next tick or not
2. The last random number of an Ant is initialized to be 0 when the Ant is created. The spec did not specify what the default value should be, and presumably users will actually call the command to generate a random number before checking the value of the last random number
3. When an AdultGrasshopper wants to jump, all open squares within 10 square radius are counted and a random one is chosen. Then the AdultGrasshopper moves to that square. The spec did not clarify whether a random square should be chosen first and then the attempt to move should be made (which will fail if the square has Pebble), or whether all open squares should be enumerated first and then a random one chosen, and then the AdultGrasshopper should move there. In the latter case, as long as there is a single open square within the radius, the AdultGrasshopper will move to it. I have decided upon the latter implementation. Note also that AdultGrasshoppers do not jump to their current square itself.
4. When an AdultGrasshopper or an Ant is asked to randomly bite an enemy, all enemies at the same square are enumerated, a random number is generated between 1 and the enemy count (say it is n ), and then the nth enemy is attacked. Thus, instead of attacking, for instance, the first enemy at any square, a random one is picked after enumerating all enemies present. The spec does not clarify what is meant by “random” in this context, so I decided upon this implementation as it is more random than simply picking the first enemy at the square every time
5. The ability to add any arbitrary Actor to the simulation has not been provided. Only public functions to add existing Ant objects or create new AdultGrasshopper objects have been provided. This is because Actors should be initialized from the field file, which is checked thoroughly for errors. Players could invalidate the simulation if given the ability to add any Actor they wish ( for example, adding multiple non-dead Food objects at the same square )
6. It is possible for two AdultGrasshopper objects to keep biting each other and retaliating over and over until one of them dies. This is possible since each AdultGrasshopper has a 1/3 chance of retaliating when bitten. But the probability is very small, since an AdultGrasshopper starts with 1600 hitpoints. 1600 / 50 = 32, and 1 / (3^32) ~ 5.396595277E-016 which is negligible. So this scenario is extremely, extremely unlikely to happen.
7. As in the simulation given on the class website, my implementation starts counting at 1999 turns and goes down to 0 turns.
8. If two ant colonies produce the same number of Actors in the same tick, then the winner is considered to be the colony who’s Anthill was asked to do something first. This is not totally fair, but this is what the simulation given on the website does. So I decided to do the same.

4. **Testing**

For the purposes of testing, it will be useful to add the following function to the public part of the StudentWorld class:

void tempAddActor(Actor\* act) {

m\_ActorLists[act->getX()][act->getY()].push\_back(act);

}

This function takes an existing Actor and adds it to the simulation at the desired location. This function will allow us to track specially created Actors once they are in the simulation.

**Each of the classes which is tested below using a main routine, should be tested also on lnxsrv07** by using the command:

g32 Actor.cpp StudentWorld.cpp main.cpp -o main

Then running ./main **will catch most memory leaks**.

For the purposes of testing, all functions in all classes should be made public.

Class: **BabyGrasshopper**

This class can be tested by tracking a single BabyGrasshopper in a program with a main routine, and testing all of its functions.

Create a StudentWorld object in the main routine, and then create a new BabyGrasshopper, passing it a pointer to the StudentWorld object.

Using the functions we can test the BabyGrasshopper’s image ID, initial random direction, initial move distance. The initial energy should be 500 and can be tested with getEnergy() function. We can test all identifier functions immediately, such as canIBeEaten() and amIWalkable() . We can check the location with getX() and getY() . Using isDead() check it is not dead.

The FaceRandDir() and SetRandDist() functions can be called and the new direction can be checked with getDirection() function. The new chosen random distance can be seen by setting a breakpoint and checking the value of m\_desiredMoveDist, which is a data member of the Grasshopper class.

Now we can call the moveInDir function and see that it returns true, then use getX() and getY() to ensure the new location is correct. Do this for each and every direction (right, left, up, down). Now use the tempAddActor function to insert a Pebble in every tile adjacent to the BabyGrasshopper, and call moveInDir for each direction, checking it returned false and getX() and getY() give the correct location of the BabyGrasshopper.

Call getBitten with 50 as the argument, and check using getEnergy that the hitpoints are now 450, and use setDead() to check BabyGrasshopper is not dead. Use tempAddActor to insert a Food object with 500 units on the BabyGrasshopper’s current location. Call Eat(200) and check using getEnergy() that the hitpoints increased by 200. Using a breakpoint, check that the food object in m\_ActorLists data member of the StudentWorld lost 200 units of food.

Now call getPoisoned() on the BabyGrasshopper and check using getEnergy() that it lost 150 hitpoints. Then call getBitten with 20 as the argument, an use getEnergy() to check the hitpoints decreased by 20. Use isDead() to check the BabyGrasshopper is not dead. Now call getStunned() on the BabyGrasshopper and check using a breakpoint that the m\_prevStunned data member of Insect class is true, and that m\_SleepOrStunTicks has increased by 2.

Create a new BabyGrasshopper in a faraway location (passing it a pointer to the same StudentWorld object). Now call its doSomething() function. Use getEnergy() to check the hitpoints decreased by 1 and use isSleepingOrStunned() to check the BabyGrasshopper is sleeping. Use getX() and getY() to check that the BabyGrasshopper moved in the direction it is facing. Call doSomething() two more times, checking that hitpoints decreased by 1 each time and the BabyGrasshopper did not move (since it is sleeping). Now use tempAddActor to add a food object with 1000 units at the current location, and call doSomething() . Check using getEnergy() that the BabyGrasshopper ate 200 units of food.

Now use gainEnergy() to give the BabyGrasshopper 1600 hitpoints. Call doSomething() and check using a breakpoint that m\_ActorLists contains an AdultGrasshopper at the same location and the food object there gained 100 units of food. Use setDead to check the BabyGrasshopper is dead.

Class: **AdultGrasshopper**

This class can be tested by tracking a single AdultGrasshopper in a program with a main routine, and testing all of its functions. We can also run the simulation with specially created fields to test specific functions.

Create a StudentWorld object in the main routine, and then create a new AdultGrasshopper, passing it a pointer to the StudentWorld object.

Using the functions we can test the AdultGrasshopper’s image ID, initial random direction, initial move distance. The initial energy should be 1600 and can be tested with getEnergy() function. We can test all identifier functions immediately, such as canIBeEaten() and amIWalkable() . We can check the location with getX() and getY() . Using isDead() check it is not dead.

The FaceRandDir() and SetRandDist() functions can be called and the new direction can be checked with getDirection() function. The new chosen random distance can be seen by setting a breakpoint and checking the value of m\_desiredMoveDist, which is a data member of the Grasshopper class.

Now we can call the moveInDir function and see that it returns true, then use getX() and getY() to ensure the new location is correct. Do this for each and every direction (right, left, up, down). Now use the tempAddActor function to insert a Pebble in every tile adjacent to the AdultGrasshopper, and call moveInDir for each direction, checking it returned false and getX() and getY() give the correct location of theAdultGrasshopper.

Call getBitten with 50 as the argument, and check using getEnergy that the hitpoints are now 1550, and use setDead() to check AdultyGrasshopper is not dead. Use tempAddActor to insert a Food object with 500 units on the AdultGrasshopper’s current location. Call Eat(200) and check using getEnergy() that the hitpoints increased by 200. Using a breakpoint, check that the food object in m\_ActorLists data member of the StudentWorld lost 200 units of food.

Now call getPoisoned() on the AdultGrasshopper and check using getEnergy() that there was no change. Then call getBitten with 20 as the argument, an use getEnergy() to check the hitpoints decreased by 20. Use isDead() to check the AdultGrasshopper is not dead. Now call getStunned() and using a breakpoint, check that m\_SleepOrStunTicks is still 0.

Create a new AdultGrasshopper in a faraway location. Now check the jumpToRandSquare(). Use tempAddActor to insert several Pebbles within 10 squares of the AdultGrasshopper and call the jumpToRandSquare() function many times. Check using getX() and getY() that the AdultGrasshopper jumped to a location within 10 square tile radius, and that there was no Pebble there.

Another way to test this function is to simply run a simulation with a map containing one BabyGrasshopper surrounded by a lot of Food and some Pebble objects. (temporarily disable creation of Anthills in StudentWorld’s init function). Once the BabyGrasshopper has become an Adult, it should be seen to be jumping to random open squares, one average once every ten ticks (press f key once, then press any key except r to advance simulation by 1 tick).

Now modify the field so that there is a horizontal wall of Pebble objects more than 10 rows deep. This divides the field into two regions. Keep a BabyGrasshopper in one region, with a lot of nearby Food. When it turns into an Adult, it should not be able to jump to the other side of the field.

Now use tempAddActor to add a BabyGrasshopper, another AdultGrasshopper, and four Ant objects (one from each colony) at the same location as the AdultGrasshopper. Store pointers to all these enemy Actors. Call biteRandomEnemy() multiple times until each of the enemies has been bitten, and check after each call that exactly one of the enemies has been bitten and the others have not (using getEnergy() ). The enemy that was bitten should lose 50 hitpoints, unless it was an AdultGrasshopper (in which case it may bite back, possibly causing the AdultGrasshopper we are considering to bite back again).

Move AdultGrasshopper to an empty square, and call getBitten with 4000 as the argument. Check that the AdultGrasshopper is dead (using isDead() ) and use a breakpoint to check new Food object with 100 units has been created at the AdultGrasshopper’s square.

Now create a fresh StudentWorld object and a new AdultGrasshopper, passing to its constructor a pointer to the new StudentWorld. Call doSomething() and print out the facing direction (using getDirection() ) and new location ( getX() and getY() ) . Use getEnergy() to check the hitpoints are now 1599. Now call doSomething() two times to ensure the AdultGrasshopper did not move (as it is sleeping). Using tempAddActor, place a Food object at the AdultGrasshopper’s position and call doSomething(). Using getEnergy, check that the AdultGrasshopper ate 200 units of food. Repeatedly call doSomething() and print out the new location of the AdultGrasshopper to ensure it avoids Pebbles and occasionally jumps to an open square.

Class: **Ant**

This class can be tested by tracking a single Ant in a program with a main routine, and testing all of its functions. We can also run the simulation with specially created fields and Bugs! programs to test specific functions.

Create a StudentWorld object and a Compiler object in the main routine, and then create a new Ant of colony 0, passing it pointers to the StudentWorld object and Compiler object.

Using the functions we can test the Ant’s image ID, initial random direction, and colony number. The initial energy should be 1500 and can be tested with getEnergy() function. We can test all identifier functions immediately, such as canIBeEaten() and amIWalkable() . We can check the location with getX() and getY() . Using isDead() check it is not dead. Check also that wasPreviouslyBlocked() and wasPreviouslyBitten() return false. Print out the direction using getDirection(), then call rotateClockwise() and rotateCounterClockwise() repeatedly, printing out the direction after each call. Check that the Ant rotates correctly.

Now we can call the moveInDir function and see that it returns true, then use getX() and getY() to ensure the new location is correct. Do this for each and every direction (right, left, up, down). Now use the tempAddActor function to insert a Pebble in every tile adjacent to the Ant, and call moveInDir for each direction, checking it returned false and getX() and getY() give the correct location of the Ant. Check also that wasPreviouslyBlocked() returns true.

Call getBitten with 50 as the argument, and check using getEnergy that the hitpoints are now 1450, and use setDead() to check Ant is not dead. Check that wasPreviouslyBitten() returns true. Now move the Ant to a new square and check wasPreviouslyBitten() returns false.

Use tempAddActor to insert a Food object with 6000 units on the Ant’s current location. Call Eat(100) and check using getEnergy() that the hitpoints did not change. Now call pickUpFood() and check using a breakpoint that the Ant is carrying 400 units of Food and that the Food object at that location lost 400 units of Food. Now call dropFood() function and check using a breakpoint that the Ant is carrying no food and the Food object at that location gained 400 units of Food. Now again call pickUpFood() 10 times repeatedly, and check using a breakpoint that the Ant holds 1800 units of Food. Now call Eat(100) and check the Ant gained 100 hitpoints of energy.

Now call getPoisoned() on the Ant and check using getEnergy() that it lost 150 hitpoints. Then call getBitten with 20 as the argument, an use getEnergy() to check the hitpoints decreased by 20. Use isDead() to check the Ant is not dead. Now call getStunned() on the Ant and check using a breakpoint that the m\_prevStunned data member of Insect class is true, and that m\_SleepOrStunTicks has increased by 2.

Now create a fresh StudentWorld object, and a fresh Ant from colony 0. Call initialInsectActions() and check that the Ant lost one hitpoint of energy, is not dead (using isDead() ). Now call getStunned() and then call initialInsectActions() two times, checking using a breakpoint that m\_SleepOrStunTicks decreased by 1 each time.

Finally we need to test the interpreter and commandTriggered functions. To test the goto\_command, faceRandomDirection command, and moveForward command, use the following Bugs! program:

colony: TESTAnt

start:

faceRandomDirection

moveForward

goto start

Run the simulation with only one Ant colony, and no poison/food/pool of water objects. Then we should see each Ant facing a random direction and moving one step forward, then repeating the process until it dies. This verifies that these three commands are working correctly.

The rotateCounterClockwise and rotateClockwise commands can also be easily tested using this Bugs! program:

colony: TESTAnt

start:

moveForward

moveForward

moveForward

goto rotate

rotate:

rotateClockwise

goto rotate

Executing the simulation in a field without poison/food/pool of water objects and only one Ant colony, we should see each Ant move three steps forward and then continuously rotate clockwise. Replacing “rotateClockwise” with “rotateCounterClockwise” we should see the Ant objects rotating counter clockwise.

The emitPheromone command can also be easily checked using this Bugs! program:

colony: TESTAnt

start:

faceRandomDirection

moveForward

moveForward

moveForward

emitPheromone

goto start

Executing the simulation in a field without poison/food/pool of water objects and only one Ant colony, we should see each Ant face a random direction, move three steps forward and then emit a Pheromone object. It should then repeat the procedure. The Pheromone objects will be clearly seen in the simulation.

Now to test pickupFood and eatFood commands, we use the following Bugs! program:

colony: TESTAnt

start:

faceRandomDirection

moveForward

pickupFood

eatFood

goto start

Test this in a field where there are no poison/pool of water objects and grasshoppers are separated from Ants by a vertical wall of Pebbles (dividing field into two sides). Keep a lot of Food near the Anthill and no food on the Grasshopper side. Use only one Ant colony.

Then we should see that the Ant objects do not die during the entire simulation. This could only happen if they picked up food and ate it. So this verifies the above commands are working.

The dropFood command can be tested by the following Bugs! program:

colony: TESTAnt

start:

faceRandomDirection

moveForward

if i\_am\_standing\_on\_food then goto on\_food

if i\_am\_standing\_on\_my\_anthill then goto on\_hill

goto start

on\_food:

pickUpFood

goto start

on\_hill:

dropFood

goto start

Test this in a field without poison/pool of water objects, only one Anthill, surrounded by a lot of Food objects. We expect the Anthill to produce many more Ants and add them to the simulation (many more ants than the initial 5 it produces). This could only happen if the Ants were dropping food on the Anthill’s square, so this verifies the dropFood command works.

This program also verifies that the if\_command works correctly. More tests for the if\_command are provided below.

To finally test the bite command, we may use the following Bugs! program:

colony: TESTAnt

start:

faceRandomDirection

moveForward

if i\_am\_standing\_with\_an\_enemy then goto enemy

goto start

enemy:

bite

goto start

Test this in a field with a lot of BabyGrasshopper objects and only one Anthill. Set a breakpoint in the implementation of the interpreter function, at the line corresponding to the case statement for the bite command. Then run the simulation.

Whenever the Ants try to bite an enemy, the breakpoint should be reached. Once at the breakpoint, check the StudentWorld object’s m\_ActorLists to ensure there is actually an enemy at the same square as the Ant.

This verifies that the bite command correctly results in biteRandomEnemy() being called.

To test the generateRandomNumber command we can use the following Bugs! program:

colony: TESTAnt

start:

faceRandomDirection

moveForward

generateRandomNumber 10

if last\_random\_number\_was\_zero then goto zero

goto start

zero:

emitPheromone

goto start

Testing this in a field without poison/pool of water/food objects, we should see the Ants emitting pheromones only occasionally. Replacing 10 with 50, we should see the Ants emitting pheromones even less frequently. On the other hand, using 3 instead of 10 should cause Ants to emit pheromones very frequently. Using the number 0 would cause the Ants to emit Pheromones in every new tile they move to.

Now we need to test the if\_command and the commandTriggered function.

If we wanted to test the i\_smell\_danger\_in\_front\_of\_me command, we could do it using the following Bugs! program:

colony: TESTAnt

start:

faceRandomDirection

moveForward

if i\_smell\_danger\_in\_front\_of\_me then goto rotate

goto start

rotate:

rotateClockwise

goto rotate

Testing in a field with Grasshopper objects, as soon as the Ant checks for danger in front and there is an enemy at that square, the Ant will continuously rotate. This will verify the working of both the if\_command and i\_smell\_danger\_in\_front\_of\_me condition.

We can test i\_was\_bit , i\_smell\_pheromone\_in\_front\_of\_me, i\_am\_hungry, and i\_was\_blocked\_from\_moving conditions in similar ways, by simply making the Ants rotate continuously if any of these conditions are satisfied.

Note that we have already tested i\_am\_standing\_on\_my\_anthill, i\_am\_standing\_on\_food, i\_am\_standing\_with\_an\_enemy, and last\_random\_number\_was\_zero previously.

**NOTE**: In testing the 3 classes above, we have also tested the functions implemented in their abstract base classes. If the base class functions were correctly implemented, then the above test cases should work exactly as specified.

Class: **Anthill**

For this class, we can simply use a function with a main routine, containing a single Anthill object.

Create new Compiler and StudentWorld, and create an Anthill, passing it pointers to the Compiler and the StudentWorld. Check immediately that getX() and getY() return correct coordinates, and getDirection() returns GraphObject::right . Check also the image ID. Check using getEnergy() that the hitpoints are 8999. Use isDead() to check that Anthill is not dead. Check the various identifier functions.

Now call doSomething(). Check that 1 hitpoint of energy is lost, and use a breakpoint to check that the Anthill creates a new Ant of its colony on the same square. Call doSomething() ten more times, checking that each call reduces hitpoints by 1. After all the calls are over, use a breakpoint to check that there are 5 ants of the same colony on the same square as the Anthill.

Now use tempAddActors to place a food object of 11000 units on the same square as Anthill. Now call doSomething() and check that the Anthill eats 10000 units of food, using getEnergy(). This verifies the working of Anthill.

One more optional check is to create an Anthill on a blank space, and keep calling doSomething() until the Anthill dies. Finally, use a breakpoint to ensure that the Anthill does not turn into a Food object after it dies.

Class: **Poison**

For this class, we can simply use a function with a main routine, containing a single Poison object.

Create a new Poison (passing it a pointer to a StudentWorld object) and use tempAddActors to put an AdultGrasshopper, a BabyGrasshopper, and an Ant on the same square as the Poison object. Now call doSomething() on the Poison object and check all insects except the AdultGrasshopper lost 150 hitpoints.

Check as usual getX(), getY(), getDirection (should return GraphObject::right), the imageID, and isDead() (should return false). Check the various identifier functions.

Class: **PoolOfWater**

For this class, we can simply use a function with a main routine, containing a single PoolOfWater object.

Create a new PoolOfWater (passing it a pointer to a StudentWorld object) and use tempAddActors to put an AdultGrasshopper, a BabyGrasshopper, and an Ant on the same square as the PoolOfWater object. Now call doSomething() on the PoolOfWater object and check using a breakpoint that all insects except the AdultGrasshopper are stunned for 2 ticks. Now call doSomething() again and check that all insects are still only stunned for 2 ticks.

Check as usual getX(), getY(), getDirection (should return GraphObject::right), the imageID, and isDead() (should return false). Check the various identifier functions.

Class: **Food**

By this point, all the testing for previous classes is enough to verify the Food objects are working correctly. But we can still explicitly test this class in a main routine with a single Food object.

Create a new Food object (passing it a pointer to a StudentWorld object) of 200 units. Use getEnergy() to check that it actually has 200 units. Now call gainEnergy(100) and use getEnergy() to check it has 300 units. Now call loseEnergy(100) and check the Food object loses 100 units. Now call loseEnergy(500) and check the Food object is dead with isDead() .

Check as usual getX(), getY(), getDirection (should return GraphObject::right), the imageID, and isDead() (should return false). Check the various identifier functions.

Class: **Pebble**

In a file with a main routine, create a new Pebble object and add it to the simulation using tempAddActors. Check as usual getX(), getY(), getDirection (should return GraphObject::right), the imageID, and isDead() (should return false). Check the various identifier functions.

Finally, run a simulation with various Ants and Grasshoppers and many Pebbles. Check that none of them ever walk into a square with a Pebble.

**This concludes testing for all Actors. Now we test StudentWorld class.**

Class: **StudentWorld**

The **init()** function can be tested by simply running the simulation with different Field files and checking that each Actor is initially in the correct place. Use a field containing every type of Actor that can be present in the field file. Test this with just one Ant colony and change the Bugs! program it is using in very obvious ways: for example, make the ants move 3 steps forward and continuously rotate clockwise. Then change the clockwise to counterclockwise, and build the simulation again, and the changed behavior of the Ants should be obvious.

This verifies that the init function correctly loads the Actors from the field and correctly creates and initializes Compilers for each colony. This also verifies the initCompilers() function called by init() .

Note that it has already been recommended that the files be tested on lnxsrv07 using the g32 command given earlier. In all of the previous test cases which were using main routines and added Actors to the StudentWorld object using tempAddActor, the **cleanup()** function would be called by the StudentWorld destructor. As a result, if all of the previous test cases (which used a main routine) worked correctly on lnxsrv07 and did not cause memory leaks, then the cleanup() function is working correctly. This also verifies the deleteCompilers() function called by cleanup() .

The **move()** function is verified by running the simulation. Create a field with no food objects and only a few grasshoppers (disable the creation of ants and anthills temporarily). Then run the simulation. The BabyGrasshoppers should sleep 2 out of 3 ticks and move on the 3rd tick. If this happens, then move function is correctly causing each Actor to do something, and updating its location as needed.

After 500 turns, the BabyGrasshoppers should immediately die and be replaced by food objects. If this happened, it implies that dead actors were correctly removed from the simulation.

The display text should continuously count turns down from 1999 to 0.

Now run the simulation with Ant colonies (re-enable creation of Ants and Anthills), and check that the asterisk is displayed after the correct Ant colony. Do this many times for multiple colony numbers and various fields. This verifies the updateWinner() and updateDisplayText() functions which are called by move function.

**The remaining functions in the StudentWorld class can be verified in a simple program with a main routine:**

**bool isSquareWalkable(int x, int y) const:** Simply create a new StudentWorld object and add Pebbles to it using tempAddActor, storing pointers to these pebbles. Now call isSquareWalkable on squares with Pebbles, checking the function returns false. On an open square, it should return true.

**bool checkEatable(int x, int y) const:** As with previous function, replace insert Food objects with tempAddActor function and check squares with Food objects cause this function to return true. Otherwise this function should return false.

**bool PheromoneAheadOfAnt(const Ant\* caller) const:** Create an Ant of colony 0 and make it face any particular direction. Place a Pheromone of colony 0 in square the Ant is facing. Add these to simulation using tempAddActor. Check this function returns true when given the Ant as the argument.

Do this again but with a Pheromone of a different colony, and this function should return false.

**bool DangerAheadOfAnt(const Ant\* caller) const:** Create an Ant of colony 0 and make it face any particular direction. Place a BabyGrasshopper / AdultGrasshopper in the square the Ant is facing. Add the to simulation using tempAddActor. Check this function returns true when given the Ant as the argument.

Do this again but using Ants from different colonies. The function should return true if the colonies are different and false if the colonies are the same.

i**nt countAntEnemies(const Ant\* caller) const:** Place any number of AdultGrasshoppers / BabyGrasshoppers / Ants from various colonies at a particular location (add them to simulation using tempAddActor). Now create an Ant of colony 0 at the same location, and check this function returns correct enemy count. Do the same for an Ant from colony 1, colony 2, and colony 3.

**int countAdultGrasshopperEnemies(const AdultGrasshopper\* caller) const:** Tested in the same way as the previous function, but now using an AdultGrasshopper instead of an Ant. Note that this function (like the previous function) does NOT consider the caller to be an enemy of itself.

**bool biteRandAdultGrasshopperEnemy(AdultGrasshopper\* caller)**:

Place BabyGrasshoppers / AdultGrasshoppers / Ants from various colonies on one particular square, adding them to the simulation using tempAddActor. Now create a new AdultGrasshopper at the same location, store a pointer to it and add it to the simulation using tempAddActor. Call biteRandAdultGrasshopperEnemy with this grasshopper as the argument, and check that exactly one enemy gets bitten and the others do not. This can be done by setting a breakpoint after the function call and checking the m\_ActorLists data member of the StudentWorld object. Call this function many times until each enemy has been bitten.

**bool biteRandAntEnemy(Ant\* caller):**

Tested in the same way as the previous function, but with an Ant of a specific colony instead of an AdultGrasshopper. Test this function with Ants from each colony.

**int eatOrPickUpFood(int x, int y, int amt):**

Store a pointer to a new Food object with 500 units, and location (5, 5). Add it using tempAddActor. Now call eatOrPickUpFood(5, 5, 200) and this function should return 200. Now call this function with arguments (6, 6, 200) and the function should return 0.

**void poisonActorsAt(int x, int y):**

Simply use tempAddActors to add BabyGrasshoppers, AdultGrasshoppers, and Ants to the simulation at one particular square. Now call this function on that square, and check that all Actors except AdultGrasshoppers lost 150 hitpoints of energy. This can be checked by storing pointers to all the inserted Actors and calling getEnergy() on each pointer.

**void stunActorsAt(int x, int y):**

Tested in the same way as the previous function, except now we will check if an Actor is stunned by setting a breakpoint after the function call and checking each Actor’s m\_SleepOrStunTicks data member (which is a data member of Insect class).

**void addOrCreateFood(int x, int y, int units)**:

Call addOrCreateFood(5, 5, 500) assuming (5, 5) does not contain any Food object. Use a breakpoint after this function call to check m\_ActorLists data member in the StudentWorld object to check a new Food object with 500 units was created at (5, 5). Now call addOrCreateFood(5, 5, 500) again, and check the Food object at (5, 5) now has 1000 units, and that there is only one Food object at (5, 5).

**void addOrCreatePheromone(int x, int y, int colonyNum):**

Tested in the same way as the previous function, except now we distinguish between Pheromone objects of different colonies. So call addOrCreatePheromone(10, 10, 1) and addOrCreatePheromone(10, 10, 2). This should lead to two different Pheromone objects being created at (10, 10), each belonging to a different colony. Also, calling this function on a tile with a Pheromone object should not increase the strength of the Pheromone object over 768 units.

**void createAdultGrasshopper(int x, int y):**

Call this function on an empty square and check using a breakpoint that the m\_ActorLists data member contains a new AdultGrasshopper at the correct location.

**void addAnt(Ant\* ant):**

Create a new dynamically-allocated Ant object, and call this function giving it a pointer to that Ant. Use a breakpoint to check that the Ant is now displayed in the m\_ActorLists data member, using a breakpoint.