1. **ALL PUBLIC MEMBER FUNCTION IMPLEMENTATIONS:**

**\*\* = “**Make the object hungrier. Check whether it’s sleeping or out of energy.”

**ACTOR BASE CLASS PUBLIC MEMBER FUNCTIONS:**

int setHitPts(int set)

Sets the energy/health points of a given actor. ALL actors (except the pebble) use this in some sort of way, whether it’s to store energy or health points.

void setDead()

Changes the state of a given actor to “dead”. ALL actors (except the pebble) can “die” in some sort of way, whether it’s an insect running out of health points or a food item running out of available “energy” that can be given to other objects.

void setNewMove()

Every new turn, this sets every actor to move this turn.

void setMoved()

Similarly, sets that the actor has moved this turn.

bool movedThisTurn()

Returns whether or not an actor has moved this turn.

StudentWorld\* callWorld() const;

Returns a pointer to StudentWorld, which is a private member variable of every actor. This is used so that actors can access information about its world and its relation to other objects.

virtual bool blocksInsect() const;

Returns whether the actor blocks insects (which only pebbles do). This function is virtual because different types of actors may or may not block insects.

std::string typeOfActor() const;

Returns a string representing the type of actor. (YES I know I could have used IMAGE\_ID, but somewhere in the source code said I shouldn’t access that, and I also ran into problems at the time I tried to implement that, so I ended up deciding to create my own system of keeping track of actor types).

*The following functions are implemented in the* ***Actor base class*** *even though at first it may seem like it would make more sense to implement them in the* ***Insect base class****. The reason for doing so is because*

1. *these functions are accessed by Actor iterators from other functions, and thus these*

*functions only work properly when implemented in the Actor class.*

1. *It does allow the program to be more flexible – what if I wanted to implement a non-insect animal actor class that also implemented retaliation, for example?*

int getSleep()

Returns the number of turns left sleeping (or stunned).

void setSleep(int howLong)

Adds or subtracts to the number of turns to sleep.

virtual void retaliate()

Returns. This function is virtual because other actor derived classes may or may not have a need to retaliate.

bool eat(int eatThisMuch)

Check if there’s food on the actors square. IF there is, subtracts eatThisMuch (or however much energy is left from food) from the food’s hit/energy points, and adds that same amount of energy to the object.

bool checkBit();

Returns whether the actor has been bitten.

void setBit(bool bit);

Sets whether the actor has been bitten.

**INSECT (DERIVED ACTOR) PUBLIC MEMBER FUNCTIONS:**

virtual bool checkEnergy()

Checks whether the insect has more than 0 energy points. (YES I KNOW, I REALIZED I COULD JUST CHECK USING THE SETHITPTS() FUNCTION, BUT I WAS ALREADY DEEP INTO THE PROJECT THAT IT WOULDN’T HAVE BEEN WORTH IT TO CHANGE THIS).

virtual void decompose(int foodUnits)

When an insect dies, add foodUnits worth of food onto its square. Only insects decompose.

bool canIMove(int x, int y) const;

Call StudentWorld to check whether the insect can move onto the space specified (e.g. if there are any objects that block insects on that space). Only insects can move.

bool attemptMove()

Select a random direction, check if it can move in that direction, then move if it can (return false without moving if it can’t). Only insects can move.

virtual void checkForHarm()

Check if the insect is on the same position as water or poison. If so then have the water or poison affect the insect as appropriate. This function is virtual because they can have different effects depending on the insect (which is by default, to do nothing). Only insects need to worry about this.

bool checkBlocked()

Returns whether the insect was blocked from moving in a given turn. Only insects can be blocked.

void setBlocked(bool block)

Set whether the insect was blocked from moving in a given turn. Only insects can be blocked.

**ENERGYHOLDER (DERIVED ACTOR) PUBLIC MEMBER FUNCTIONS:**

virtual void doSomething();

Set it to dead if it is not holding any energy. Function virtual because classes derived from Actor may have different implementatons of doSomething().

**HARMER (DERIVED ACTOR) PUBLIC MEMBER FUNCTIONS:**

virtual void doSomething();

Return. Harmers (as well as pebbles) don’t do anything.

**ANT (DERIVED INSECT) PUBLIC MEMBER FUNCTIONS:**

virtual void doSomething()

\*\* Then do as instructed by its Bugs program file (via runCommand).

int getColonyID()

Return colonyID, a member variable that specifies which colony (0 – 3 inclusive) the ant belongs to. Only applies to ants.

bool runCommand(const Compiler& c)

Make the ant do what’s instructed by the Bugs program file. Only applies to ants.

bool checkCondition(int operand)

Helper function for runCommand; check whether certain conditions for a given command is true. Return whether such conditions are met. Only applies to ants.

**GRASSHOPPER (DERIVED INSECT) PUBLIC MEMBER FUNCTIONS:**

virtual void doSomething()

\*\* Bite if it feels like it, OR jump if it feels like it, OR eat if it feels like it, OR try to move if it feels like it. Grasshoppers move a certain direction for a certain number of turns. If it chooses to move and either it’s blocked or it has moved in that direction for that certain number of turns, then choose a new direction. If it moved, then go back to sleep.

double getDistance()

Return how much further the Grasshopper is set to move in a given direction. Only applicable to Grasshoppers.

void setNewDistance()

Set how far Grasshopper wants to move in a new given direction. Only applicable to Grasshoppers.

void decrementDist()

Decrease the distance the Grasshopper will move in a given direction. Only applicable to Grasshoppers.

virtual bool checkEvolve()

Return. However, BabyGrasshopper, a derived class, does make use of this function. This function is implemented in the (adult) Grasshopper class so that BabyGrasshopper can use the same doSomething() function (which just calls these other functions). Only applicable to Grasshoppers.

virtual bool bite()

Bite if it feels like biting.

virtual bool jump()

Jump if it feels like jumping. Only applicable to Grasshoppers.

virtual void retaliate()

If it got bitten, then bite back if it feels like doing so. Only applicable to Grasshoppers.

**BABYGRASSHOPPER (DERIVED GRASSHOPPER) PUBLIC MEMBER FUNCTIONS:**

virtual void checkForHarm()

If it’s on water, make it sleep. If it’s on poison, it loses 150 hit points.

virtual bool checkEvolve()

Check whether it has enough energy (1600) to turn into an adult grasshopper, and do so if it does. Only applicable to Grasshoppers.

virtual bool bite()

Do nothing (this is a derived function from Grasshopper, which DID make use of this function).

virtual bool jump()

Also do nothing (this is a derived function from Grasshopper, which DID make use of this function). Only applicable to Grasshoppers.

**PHEROMONE (DERIVED ENERGYHOLDER) PUBLIC MEMBER FUNCTIONS:**

void EnergyHolder::doSomething()

Set it dead if it has no more energy.

***FOOD DOES NOT HAVE ITS OWN SPECIAL PUBLIC MEMBER FUNCTIONS***

**ANTHILL (DERIVED ENERGYHOLDER) PUBLIC MEMBER FUNCTIONS:**

virtual void doSomething()

Lose one energy point. Die if it has no energy. Eat whatever is on its square. If it has enough energy, give birth to a new any of its colony, at a cost of 1500 energy.

int getNumberAnts();

Return the number of ants that this anthill has produced. Only applicable to anthills.

***WATERPOOL DOES NOT HAVE ITS OWN SPECIAL PUBLIC MEMBER FUNCTIONS***

***POISION DOES NOT HAVE ITS OWN SPECIAL PUBLIC MEMBER FUNCTIONS***

**STUDENTWORLD PUBLIC MEMBER FUNCTIONS**

virtual int init()

Initialize all the data structures (only ONE of my data structures actually holds the field objects; the rest are helper data structures used for my other functions). Set ticks to zero.

virtual int move()

Ask each object to doSomething() if it hasn’t done so already. Update positions of any objects that move. Remove any dead objects. If the game is over, determine the winner.

void updateTickCount()

Increase ticks by one (duh.)

int getCurrentTicks()

Get current number of ticks.

void updatePosition(Actor\* actor, int oldx, int oldy)

Add actor to new position, remove it from old position.

void resetMove()

Set that the actor can move.

bool canMove(int x, int y)

Return whether a given actor can move to the specified condition (i.e. check for any objects that won’t allow an actor to move there).

void updateDisplayText()

Get all the parameters needed and call formatDisplayText(). Then call setGameStatText(). (This function doesn’t really do anything on it’s own; it calls other helper functions).

std::string formatDisplayText(int ticks, int ants0, int ants1, int ants2, int ants3, int winningAntNum);

Take all the parameters, get info about each ant’s colony name, convert all the data into strings, and paste all that together!

bool loadField(int x, int y)

Load the field file, get the contents at the specified location, then dynamically allocate and push into data structure any corresponding objects as needed.

void decomposeInsect(int x, int y, int foodUnits)

Dynamically allocate foodUnits worth of food at the same location that an insect has died.

std::list<Actor\*> findActor(int x, int y)

Return the list of actors at the specified location.

void removeTheDead(int x, int y)

Dynamically deallocate any actors that are dead at the specified location.

void setAnts(int numberAnts, int colonyID)

Set the number of ants that a given anthill has produced.

void setTickAnt(int tick, int colonyID)

Set the tick (‘the time’) that an anthill has produced its most recent ant.

int winningAntNum()

Determine which ant is winning – the ant that has produced the most ants, and in the case of a tie, the one which as produced that ant first. If there is still a tie, then return -1, indicating that there is no winner.

*Because certain functions in Actor and its derived classes need to access information about the field, I have created the following helper functions in StudentWorld for those functions:*

int insectEat(int x, int y, int eatThisMuch)

See if there’s food at the location. Subtract eatThisMuch (or however much energy the food object has left, whichever is lower) from the food object. Return the magnitude of that number.

bool antCheckPheromone(int x, int y, std::string type)

bool antCheckInsect(int x, int y, std::string type);

bool insectCheckObject(int x, int y, std::string type)

These three functions have fundamentally different implementations and thus I have decided to implement them as separate functions. These functions basically just check whether a given type of actor is at a given location (guess which kind of object each function checks).

void newFood(int x, int y, int amt)

void newPheromone(int x, int y, std::string image)

void newAnt(int x, int y, Actor\* a)

void newGrasshopper(int x, int y)

These four functions have fundamentally different implementations and thus I have decided to implement them as separate functions. These functions basically just dynamically allocate a new object (you can obviously figure out which kind of object each of these functions allocate).

bool antgrasshopperbite(Actor\* itself, int x, int y)

Bite a random insect on the same square, if it can. Let that insect retaliate if it feels like it.

virtual void cleanUp()

Dynamically deallocate everything because it’s game over.

1. **ALL KNOWN BUGS AND STUFF I FAILED TO IMPLEMENT:**
2. I’ve implemented everything.
3. I’ve noticed that the USCAnts get somewhat higher scores (up to 20) in my program. I believe that these higher scores are not random, that they are significant enough of a difference to indicate that either my ant’s interpreter or my anthills may not be implemented correctly.
4. Also, sometimes, my display text may occasionally not put an asterisk (\*) at the right ant; this seems to happen when the ants switch positions (i.e. switching from first to second place) a lot.
5. **ASSUMPTIONS I MADE:**
6. Can grasshoppers that are a victim of retaliation also retaliate themselves? And can those retaliated by that grasshopper also retaliate? I assumed so.
7. Does food immediately become dynamically allocated as soon as an insect dies? I assumed that, while the insect is NOT dynamically deallocated until the end of the turn, it’s corresponding “food remains” IS dynamically allocated immediately.
8. **HOW I TESTED EACH OF MY (USEFUL) CLASSES:**

* Ant: I wrote different ant class files. Each class file served to test each command. I checked whether each ant properly interacted with the world (i.e. it does what it’s supposed to do when it encounters certain objects). I also used cerr to check whether the ant was handling energy correctly and setting itself to dead when needed.
* Grasshopper: I commented out the allocation of ants and anthills to simplify testing. I then isolated each function (except doSomething) to see whether each function worked properly. I checked simple features like whether distance was properly decremented as well as more higher level ones such as whether jumping worked as expected (i.e. the grasshopper doesn’t try to jump out of bounds).
* BabyGrasshopper: Basically the same way I tested out the grasshopper. In particular, I checked whether the baby grasshopper properly allocated a grasshopper and deallocated itself whenever it had 1600 energy points. I also checked whether it interacts properly with harmful objects (i.e. poison and water).
* Pheromone/Food/WaterPool/Poison/Pebble: These all don’t really do very much. Anything that they are supposed to do are handled by the other classes. The only thing I really tested for these functions was whether they EXISTED in the game and whether other objects interacted with them.
* AntHill:I used cerr to check whether allocated ants only when it’s supposed to (whenever it has more than 2000 energy points), and checked whether it “ate” properly. I also checked whether it allocated the correct kind of ant (and not an ant of a different colony). Of course I checked whether it corresponded to the correct ant colony (no USC ants allowed in UCLA territory!).