Report

Actor:

virtual void doSomething() = 0; pure virtual function because every actor has a different way of doing something in each tick, for example a Pebble nothing, whereas an Ant does a variety of things depending on the program it was given.

bool isDead() const; this function returns true if the actor is dead. This is not a virtual function because all Actors die the same way, when there m\_state is set to true. const because it doesn’t make any changes.

virtual bool blocksMovement() const; this function returns true if the actor blocks the movement of the other actor else false. virtual function because only Pebbles block movement, all other actors do not. Defined here so it can be called from StudentWorld.

virtual void getBitten(int amt); Defined here so it can be called from StudentWorld. virtual function only insects get bitten, all other actors do not. Insects get bitten in different ways. Further description in specific classes.

virtual void getPoisoned(); Defined here so it can be called from StudentWorld. virtual function only insects(ants and baby grasshoppers) get poisoned, all other actors do not. Further description in specific classes.

virtual bool isEdible() const; Defined here so it can be called from StudentWorld. returns true if the actor is edible i.e. food object, else false. This is a virtual function because all actors other than food have same behaviour.

virtual bool isPheromone(int colony) const; Defined here so it can be called from StudentWorld. returns false for all actors other than the pheromone object of that colony. Virtual function because all actors have a same behaviour other than the pheromone.

virtual bool isEnemy(int colony) const; Defined here so it can be called from StudentWorld. returns false for other actors other than the insects. Virtual function because all insects act in a certain way.

virtual bool isDangerous(int colony) const; Defined here so it can be called from StudentWorld. returns false if not dangerous, virtual function because some actors are dangerous where the other other are not. Further description in specific classes.

virtual bool isAntHill(int colony) const; Defined here so it can be called from StudentWorld. returns true if the actor is a anthill else false, this function is virtual function because all actors act the same way but the anthill class doesn’t.

StudentWorld\* getWorld() const; returns the StudentWorld that the current actor is part of, not virtual because all actors use this function the same way.

void setDead(); sets m\_state is true, not virtual because all actors use this function the same way.

virtual bool canBeTriggered() const; Defined here so it can be called from StudentWorld. returns true if the actor can be poisoned or stunned else false, this function is virtual function because all actors act the same way but the ant and baby grasshopper class doesn’t.

bool isStunned() const; returns true if m\_isStunned is true, that is if the actor stunned, not virtual because all actors use this function the same way.

void setStunned(); sets m\_isStunned is true, not virtual because all actors use this function the same way.

void setUnstunned(); sets m\_isStunned is false, not virtual because all actors use this function the same way.

virtual int getEnergy() const; returns the hitpoints of the actor, virtual function because actors only that are energyholders return the true energy, other actors return 0, a default value. Defined here so it can be called easily while using pointers to actors.

virtual void updateEnergy(int amt); changes total hitpoints, virtual function because actors only that are energyholders updates the energy appropriately, other actors do nothing. Defined here so it can be called easily while using pointers to actors.

virtual int getColony() const; Defined here so it can be called from StudentWorld. returns the colony of the anthill, defined in this class so we can access this from studentWorld. Virtual because only anthills return the colony, other actors return -1.

virtual Compiler\* getProgram() const; returns the colony of the anthill, defined in this class so we can access this from studentWorld. Virtual because only anthills return the colony, other actors return nullptr. Defined here so it can be called easily while using pointers to actors.

virtual void increaseSleepTicks(int amt); only virtual so can be called from studentworld. Doesn’t do anything for other actors, but works for insects. Defined here so it can be called easily while using pointers to actors.

Pebble:

virtual void doSomething(); virtual because in actors do something different. In the case, the pebble does not.

virtual bool blocksMovement() const; returns true because pebbles do block movement. Defined here because, Pebbles behave differently from other actors, when it comes to blocking.

EnergyHolder:

virtual int getEnergy() const; return the energy of the actor, defined here because only energy holders return their true energy, all other actors return some default value, virtual because easy access while using pointers to Actors and not specific pointers to Energyholders.

virtual void updateEnergy(int amt); updates the amount of energy in the energyholder. Virtual because allows easy access while using pointers to actors.

Write pseudo//void addFood(int amt); Adds amt of food to the location of the actor that calls it. If food already exists in that location then updates the units it holds. If it doesn’t then creates a new Food object using the studentWorld. Defined here because only energyHolders can addFood and not virtual because all Energyholders add food the same way.

int pickupAndEatFood(int amt); makes the actor that calls it eat food from the returns the amount of food an actor pick or 0 is there was no food. amt is the max amount of the actor can pick up. It calls student world to search for a food object in that location and if found then the actor and the food object update energy appropriately. If the food object doesn’t have any hitpoints left, sets it as dead. Not virtual because all energyholders(insects) pick and eat food the same way.

Food:

virtual void doSomething(); virtual because all actors do something different. In this case, the food object does nothing.

virtual bool isEdible() const; returns true because food is edible. Defined here because, Food behaves differently from other actors, when it comes to being edible.

AntHill

virtual void doSomething(); virtual because all actor do something different. decreases the hitpoints of the hill, if it reached to zero then setdead and return otherwise check if food exists on square if it does tries to pick up 10000, else how much ever available on the location and returns, if there was no food, checks energy if it has sufficient then produces a new ant using the help of the student world and reduces it’s hit points if it does and asks student world to keep track of the number of ants.

virtual int getColony() const; returns the colony of the anthill, virtual because easy access from the studentworld while access actor pointers.

virtual Compiler\* getProgram() const; returns the program that runs the ants of this anthill. virtual because easy access from the studentworld while access actor pointers.

Pheromone:

virtual void doSomething(); reduces hitpoints every tick.

virtual bool isPheromone(int colony) const; returns true if the colony is the same(Pheromones can only be detected by it’s ants ). Defined here because, pheromones behaves differently from other actors. virtual because easy access from the studentworld while access actor pointers.

TriggerableActors: (Ants and BabyGrasshoppers)

virtual bool isDangerous(int colony) const; returns false, BabyGrasshoppers are not dangerous but Ants are (defined again in the Ant class). virtual because easy access from the studentworld while access actor pointers.

WaterPool:

virtual void doSomething(); stuns all stunnable actors at that location, uses Studentworld to do so.

Poison

virtual void doSomething(); poison all poisonable actors at that location, uses StudentWorld to do so.

Insect:

virtual void doSomething() = 0; pure virtual function because every insect has a different way of doing something in each tick.

virtual void getBitten(int amt); deducts the bite damage from the energy, if the actor’s energy hits zero then sets dead. Virtual because each insect gets bitten differently. E.g :AdultGrashopper might decide to bite back. virtual because easy access from the studentworld while access actor pointers.

virtual void getPoisoned(); deducts the poison damage from the actor. virtual because easy access from the studentworld while access actor pointers.

virtual bool isEnemy(int colony); return true; this is just a general response, all insects are enemies, only ants have a specific function for this. virtual because easy access from the studentworld while access actor pointers.

void getXYInFrontOfMe(int& x, int& y); Set x,y to the coordinates of the spot one step in front of this insect. Not virtual because all insects use this function the same way.

bool moveForwardIfPossible(); Move this insect one step forward if possible( Uses studentworld to check it the new position is blocked), and return true; otherwise, return false without moving. Not virtual because all insects use this function the same way.

int getSleepingTicks() const; return the number of ticks the insect needs to be sleeping, Not virtual because all insects use this function the same way.

void decreaseSleepingTicks(); decreases the number of ticks the insect need to be sleeping. Not virtual because all insects use this function the same way.

void increaseSleepTicks(int amt); increase the number of ticks the insect needs to be sleeping by the amt. Not virtual because all insects use this function the same way.

Ant:

virtual void doSomething();

{

Reduce hits points by one;

Check to see if energy has reached 0

{ if yes, then set dead and food to location; and return }

Check to see if Ant is stunned

{

If yes, decrease the ticks it needs to remain sleeping and return.

}

For 10 commands

{

get the command from element m\_counter of the vector

If couldn’t load the command,

Then return;

Else check to see what command and do the command;

Update the command counter, whether the ant was bitten, or blocked ect.

}

} virtual function because every actor has a different way of doing something in each tick

virtual bool isEnemy(int colony) const; if it is not the same colony as the ant return true, else false. virtual because easy access from the studentworld while access actor pointers.

virtual void getBitten(int amt); deducts the bite damage from the energy, if the actor’s energy hits zero then sets dead. Virtual because each insect gets bitten differently. In this case, ants set m\_bitten to true.

virtual int getColony() const; returns the colony of the ant, virtual because easy access from the studentworld while access actor pointers.

virtual bool isDangerous(int colony) const; returns true if not the same colony else false. virtual because easy access from the studentworld while access actor pointers.

virtual bool canBeTriggered() const; returns true; it can be poisoned and stunned. virtual because easy access from the studentworld while access actor pointers.

void setBitten(); sets m\_bitten to true, not virtual because this function is unique to ants.

Grasshopper:

virtual void doSomething() const; virtual because the baby and adult grasshoppers do different things

int getDistance() const; returns the distance left to travel in a certain direction. Not virtual because both the grasshoppers use this function the same way.

void resetDistance(); resets the max distance the grasshopper needs to travel in one direction. Not virtual because both the grasshoppers use this function the same way.

void decreaseDistance(); decreases the distance left to travel in a certain direction. Not virtual because both the grasshoppers use this function the same way.

int getTicks() const; returns the number of resting ticks. Not virtual because both the grasshoppers use this function the same way.

void decreaseTicks(); decreases the number of resting ticks. Not virtual because both the grasshoppers use this function the same way.

void resetTicks(); resets the number of resting ticks. Not virtual because both the grasshoppers use this function the same way.

void setDistanceZero(); resets the distance left to travel in a certain direction to 0. Not virtual because both the grasshoppers use this function the same way.

BabyGrasshopper:

void doSomething()

{

Decrease the hitpoints

check to see if its hit points have reached zero

{

If so, it must:

Add 100 units of food to the simulation world in its current x,y location

Setdead and return

}

if BabyGrasshopper is sleeping

{

decreaseTicks();

return;

}

If it is stunned

{

decreaseTicks();

return;

}

setUnstunned();

resetTicks(); // start ticks over

The baby Grasshopper checks its hit points, if greater than or equal to 1600

{

Create and add a new adult Grasshopper object to the simulation in the same square as the baby.

Set the baby’s status status to dead, resulting in a pile of 100 units of food being return;

}

if BabyGrasshopper picked up any food

{

50% chance of sleeping after eating return;

}

if max distance travelled in certain direction

change the direction and pick another random distance

If couldn’t move forward in current direction

{

Set the distance that needs to be travelled in that direction to 0

return;

}

else

{

Decrease the distance travelled in present direction

}

}

bool canBeTriggered() const; returns true; it can be poisoned and stunned. virtual because easy access from the studentworld while access actor pointers.

virtual bool isDangerous(int colony) const; returns true all grasshopper are dangerous. virtual because easy access from the studentworld while access actor pointers.

virtual void doSomething()

{

Decrease the hitpoints

check to see if its hit points have reached zero

{

If so, it must:

Add 100 units of food to the simulation world in its current x,y location

Setdead and return

}

if AdultGrasshopper is sleeping

{

decreaseTicks();

return;

}

resetTicks(); // start ticks over

The Adult Grasshopper checks it wants to bite something( 1 in 3 chance)

{

Bite an enemy in the same square as the adult , return

}

If the adult grasshopper wants to jump( 1 in 10 chance )

{

Set x and y to new ones with radius 10;

If you can move, move

Else

Keep setting new coordinates till you find ones you can move to

}

if Adult Grasshopper picked up any food

{

50% chance of sleeping after eating return;

}

if max distance travelled in certain direction

change the direction and pick another random distance

If couldn’t move forward in current direction

{

Set the distance that needs to be travelled in that direction to 0

return;

}

else

{

Decrease the distance travelled in present direction

}

}

virtual void getBitten(int amt); deducts the bite damage from the energy, if the adult’s energy hits zero then sets dead. If bitten 50% chance it will bite back in the same tick. virtual because easy access from the studentworld while access actor pointers.

StudentWorld:

virtual int init();

{

initialise the date structure

Load a field, if it wasn’t possible return error

resetTicks();

otherwise the load was successful and you can access the contents of the field

get the names of all of the ant program source files

If any don’t conform to the requirements, return false.

access each coordinate in the field and if the coord holds a particular Actor then create one and push it onto the data structure

return GWSTATUS\_CONTINUE\_GAME;

}

virtual int move()

{

update the current tick # in the simulation

Give each actor a chance to do something

Remove newly-dead actors after each tick

Update the simulation Status Line

check if something is winning, use this to update the int m\_winner to make

updateDisplayText better at handling the asterisk '\*'

if the sim is over and you have a winning ant

{

Return that the winning ant won

}

else

return GWSTATUS\_NO\_WINNER;

}

return GWSTATUS\_CONTINUE\_GAME;

}

virtual void cleanUp(); the cleanUp() function traverses through the list of Actors using an iterator and deletes each one, freeing up any allocated memory.

cleanUp() is used in the destructor for StudentWorld.

bool canMoveTo(int x, int y) const; function traverses through the list of Actors using an iterator and checks if a pebble exists at that particular spot, returns true if a pebbles doesn’t exist else false.

Actor\* getEdibleAt(int x, int y) const; If an item that can be picked up to be eaten is at x,y, return a pointer to it; otherwise, return a null pointer. (Edible items are only ever going to be food.) function traverses through the list of Actors using an iterator and checks if a food item exists at that particular spot.

Actor\* getPheromoneAt(int x, int y, int colony) const; If a pheromone of the indicated colony is at x,y, return a pointer to it; otherwise, return a null pointer. function traverses through the list of Actors using an iterator and checks if a pheromone of that colony exists at that particular spot.

bool isEnemyAt(int x, int y, int colony) const; If an enemy of an ant of the indicated colony at x,y then return true else false. function traverses through the list of Actors using an iterator and checks if an enemy of that colony exists at that particular spot.

bool isDangerAt(int x, int y, int colony) const; if something dangerous to an ant of the indicated colony at x,y return true else false, function traverses through the list of Actors using an iterator and checks if an enemy or danger to that colony exists at that particular spot.

bool biteEnemyAt(Actor\* me, int colony, int biteDamage); Bite an enemy of an ant of the indicated colony at me's location (other than me; insects don't bite themselves). Return true if an enemy was bitten. If more than one enemy exist, pick a random one to bite. function traverses through the list of Actors using an iterator and checks if an enemy exists at that particular spot, if it doesnt then pushed it on to a vector, at the end picks a random enemy from that vector to bite.

bool poisonAllPoisonableAt(int x, int y); Poison all poisonable actors at x,y. function traverses through the list of Actors using an iterator and checks if an actor that can be poisoned exists at that particular spot.

bool stunAllStunnableAt(int x, int y); Stun all stunnable actors at x,y. function traverses through the list of Actors using an iterator and checks if an actor that can be stunned and is not already stunned exist at that particular spot.

void increaseScore(int colony); Record another ant birth for the indicated colony, increase the score of the anthill.

void removeDeadSimulationObjects(); This function deletes dead simulation objects from the data structure. function traverses through the list of Actors using an iterator and deletes one when it is dead, freeing up any allocated memory.

void updateDisplayText(); This function updates the simulation Status Line using the someFunctionToFormatThingsNicely.

bool theSimulationIsOver() const; Returns true if the sim is over else false

void updateTickCount(); Changes the ticks everytime move() is called

bool weHaveAWinningAnt(); if an ant has produced more than the original 5 ants returns true else false,

string getWinningAntsName() const; Returns the name of the winning ant

Actor::Direction generateRandDir() const; gets a random direction, used when insects are created facing a rand direction.

void addAnt(Actor\* anthill); adds an Ant at the location of the anthill depending on the colony the anthill, then adds it to the data structure.

void addPheromone(Actor\* ant); adds an pheromone at the location of the ant depending on and colony the ant, then adds it to the data structure.

void addAdultGrasshopper(Actor\* BabyGrasshopper); adds an AdultGrasshopper at the location of the baby, then adds it to the data structure.

void addFoodSW(Actor\* actor, int amt); adds an food object with initial energy amt at the location of the actor, then adds it to the data structure.

string someFunctionToFormatThingsNicely(int ticks, int antsAnt0, int antsAnt1, int antsAnt2, int antsAnt3, int winningAntNumber); functions formats things for updateDisplayText using string streams.

void resetTicks(); resets ticks to 2000, used in init().

3. One of the assumptions I made was that even if an anthill dies, but it still had more ants than another anthill, it can still be a winner.

4.

Actor Class:

The was I tested this was to make sure that I overrode the correct functions in the correct places. For example, I wrongly overrode the isDead function because I set the isDead function to virtual. This was wrong, so I changed the isDead function to non-virtual.

Pebble Class:

To test this class, I made only one pebble to make it easier to test. Then I made sure that every Insect, or movable object, was blocked by it.

Anthill Class:

I tested to make sure that the Anthill made enough Ants at the start of the game by putting cerr statements to print out its energy every time i wanted an ant to be created. I tested to see if the energy in the anthill was increasing as well to make sure that ants were bringing food back to the anthill so that the anthill could produce more ants.

Obviously I was able to test the Anthill class by looking to see if the number of ants increased on the display screen, which showed if the anthill was able to gain food and create a new ant.

Pheromone Class:

I made sure that after each tick, the pheromones lost energy using cerr statements. I also tested ants to make sure that they were connected to the corresponding Pheromone class according to their colony number

TriggerableActor Class:

I tested to ensure that only the water and the poison were set to be dangerous

Ant Class:

I put cerr statements after each case in the switch statement and if condition. This helped me figure out if there were any conditions that were flawed or not being reached at all. I also only added one ant to begin with, which made it easier to keep track of and test.

Grasshopper: there was not much to test here beside making sure that the adult and baby grasshopper constructors were working properly and passing in the right variables from grasshopper, because grasshopper is the base class for baby and adult grasshoppers.

babyGrasshopper:

I set breakpoints throughout the babyGrasshopper’s doSomething function to make sure that every part of the code is reached. I also set cerr statements throughout to make sure that the private member variables like the number of ticks and energy were incrementing and decrementing properly.

adultGrasshopper:

Similar to how I tested the babyGrasshopper, I added breakpoints throughout the code to make sure everything was being reached. Also, just like how i did in babyGrasshopper, when I ran the program, I used the f key so that I could look at the movement of each adultGrasshopper and babyGrasshopper for every tick. I also printed the new coordinates every time a grasshopper decided to jump to make sure of the radius.

StudentWorld class:

One of the most useful and efficient ways that I used to test the StudentWorld class was to manually put in coordinates when I wanted to test functions like isEnemyAt or isDangerAt. Because I knew what object was at these specific points, I already knew the outcome, so I could use that information to test the function and see if its outcome matched my predicted outcome. For the move function, I placed cerr lines throughout to make sure that every part of the code was being hit. For obvious functions, like updateDisplayText, I ran the program and looked at the output on the console. I also used several static variables to keep track of my Actors, and the various categories of Actors and I put cerr statements while adding or deleting an actor, and while calling the cleanUp and ~Studentworld().