Edward Chu

504772148

**Report for Project 3: Bugs!**

A. Description of all public member functions:

StudentWorld:

Int init(){

Initialize all member variables;

Get pointers to compilers of all entrants, store pointers in m\_compilerForEntrant[4];

Load field data, push data into STL structure

}

Int move(){

Update tick count;

(Moving phase)

For each actor:

Call each actor’s doSomething()

Update STL structure if actor has moved

(Damaging Object Phase)

For each damaging object (Water pools and Poison)

Activate the damaging object

(Delete Dead Actors Phase)

For each actor:

If dead:

Delete the Actor

Remove from STL structure

UpdateWinner();

SetDisplayText();

If m\_ticksElapsed > 2000

If there’s a winner

Return GWSTATUS\_PLAYER\_WON

Else return GWSTATUS\_NO\_WINNER

Return GWSTATUS\_CONTINUE\_GAME (Continue to next tick)

}

void cleanup(){

For each actor:

Delete the actor

Remove from STL Structure

For each compiler:

Delete the compiler

}

Public methods called by actors:

1. bool ableToMoveHere(int x, int y) const;

Checks if any of the actors at (x,y) blocks insects, by calling each actor’s blockInsect(). Returns false if blocked, otherwise return true.

1. int eatFoodHere(int x, int y, int amount);

Checks if there’s food on the square, if yes, call the food’s eat() function. Returns the actual amount of food eaten (e.g. 0 if no food is here).

1. Int addFoodHere(int x, int y, int amount);

Either add the amount to an already existing food on the square, or create a new food on the square if there’s none. Return the total amount of food on the square.

1. Bool hasFoodHere(int x, int y);

Returns true if there’s food on this square, otherwise return false.

1. Bool spawnAdultGrasshopperHere(int x, int y);

Spawns an adult grasshopper at (x,y), returns true if (x,y) is a valid location (Not outside boundaries/on a pebble), otherwise return false.

1. int getNumAnts(int colonyNum) const;

Return the total number of ants this colony has spawned.

1. bool spawnAnt(Anthill \*self);

Spawn an ant of the colony of the Anthill on top of the anthill. Returns true if the anthill has over 2000 hp, otherwise return false.

1. int damageInsects(Actor\* attacker, int x, int y, int damageType, int damage, bool damageAll);

Called by an actor when they want to deal damage to other actors. Returns the number of insects damaged.

DamageType refers to the type of damage dealt (0 = Bite Damage, 1 = Stun Damage, 2 = Poison Damage). This is slightly clumsy as I was unable to put in my own constants in GameConstants.h.

DamageAll is true when the damage is dealt to all of the insects on the square (i.e. water pools and poison), and false when it only damages one insect (i.e. bite damages).

Example of how to call this function in AdultGrasshopper, which does 50 points of Bite damage to a random opponent insect:

getStudentWorld()->damageInsects(this, getX(), getY(), 0, 50, false);

Note: This function will not allow ants of the same colony to damage each other.

1. bool isStandingWithEnemy(Ant \*self) const;

Returns true if the ant is standing with a grasshopper or an ant from another colony, otherwise return false.

Note: this function simply calls bool hasEnemyHere(Ant \*self, int x, int y) const with the ant’s position (i.e. x = self->getX() and y = self->getY())

1. bool hasEnemyHere(Ant \*self, int x, int y, bool checkPoison) const;

Returns true if the ant has an enemy (i.e. a grasshopper or an ant from another colony) (or poison, if checkPoison is true) at the square, otherwise return false.

1. bool isOnOwnAnthill(Ant \*self) const;

Returns true if the ant passed in is standing on its own anthill, otherwise returns false.

1. bool hasPheromoneHere(Ant \*self, int x, int y) const;

Returns true if the ant has a pheromone of the same colony at the square, otherwise return false.

1. int emitPheromone(Ant \*self);

Either add 256 strength (hitpoints) to an existing pheromone of the same colony at the ant’s position, or create a new pheromone of the same colony if there isn’t one. Returns the pheromone’s strength (hitpoints).

Actor:

Class diagram:

Class Actor:

Constructor:

Actor(int imageID, int startX, int startY, StudentWorld \*sw, Direction dir = right, int depth = 0);

A pointer to the StudentWorld is passed in in order for the actor to get information about the world (e.g. for Insects to check if there’s a pebble in front of them).

Destructor:

Nothing here!

Public methods:

1. virtual void doSomething(){m\_actionsTaken++};

Add 1 to the m\_actionsTaken. All children should call Actor::doSomething() at the start of their doSomething, then compare getActionsTaken() with getStudentWorld->getTicksElapsed. If the former is equal (or greater) to the latter, return without doing anything.

This is because due to the implementation of the STL Structure, some actors may move twice during the same tick. Thus, m\_actionsTaken is compared to the studentWorld’s m\_ticksElapsed to prevent this.

1. virtual void getHurt(int damageType, int damage) = 0;

Allows the actor (usually insect) to getHurt based on the damageType or damage. This function should never be called by the actor itself or other actors directly, but should instead be called indirectly through studentWorld’s damageInsects().

Identifiers:

virtual bool blocksInsect() const {return false;};

virtual bool isInsect() const {return false;};

virtual bool isDamagingObject() const {return false;};

virtual bool isAnt() const {return false;};

virtual bool isAnthill() const {return false;};

virtual bool isGrasshopper() const {return false;};

virtual bool isPheromone() const {return false;};

These functions are all assumed to be false unless specified in the children (for example, pebble’s blocksInsect() returns true instead).

This is due to the large number of identifiers, which makes it unwieldy and clumsy if all of them are declared as pure virtual functions, as all of the children will have to make implementations for them, even though most of them would just return false.

Others:

bool isAlive() const {return m\_isAlive;};

Returns if the actor is alive or not.

StudentWorld\* getStudentWorld() const {return m\_sw;};

Returns a pointer to the studentWorld of the actor.

Class EnergyHolder : public Actor

Constructor:

EnergyHolder(int imageID, int startX, int startY, int initialHP, StudentWorld \*sw, Direction dir = right, int depth = 0)

Allows the user to set an initial amount of hitpoints for the EnergyHolder.

Public methods:

1. int getHitpoints() const {return m\_hitpoints;};

Returns the amount of hitpoints the EnergyHolder has.

Class Insect : public EnergyHolder

Constructor:

Insect(int imageID, int startX, int startY, int initialHP, StudentWorld \*sw, Direction dir = right, int depth = 0)

Public methods:

1. virtual void getHurt(int damageType, int damage);

Implementation of actor’s getHurt.

For bite damage and poison damage, the Insect’s hitpoints is reduced according to the damage. For Stun damage, the Insect is stunned for the amount of turns specified in damage.

Note: DamageType: 0 = Bite damage, 1 = stun damage, 2 = Poison damage.

Class Grasshopper : public Insect

Constructor:

Grasshopper(int imageID, int startX, int startY, int initialHP, StudentWorld \*sw)

A randomDirection is assigned for the initial direction of the grasshopper.

Public Methods:

1. virtual bool isGrasshopper() const {return true;};

This is a grasshopper!

Note: This class is declared mostly due to similarities in AdultGrasshopper’s and BabyGrasshopper’s doSomething, which is hidden away in protected methods in this class.

Class AdultGrasshopper : public Grasshopper

Constructor:

AdultGrasshopper(int startX, int startY, StudentWorld \*sw)

Public Methods:

1. virtual void doSomething(); //Implementation of actor’s doSomething

Pseudocode:

{

Calling grasshopper’s protected method, check if already moved/stunned/dead, return immediately if true.

1/3 Chance to bite another insect on the same square,

if successful:

Sleep for 2 turns

Return

Jump to an open square without a boulder in circular radius of 10, 1/10 chance

Calling grasshopper’s protected method, eat, move or reset desired distance accordingly.

}

1. virtual void getHurt(int damageType, int damage);

Overrides insect’s getHurt, as adult grasshoppers don’t get poisoned/stunned.

If the adult grasshopper is bitten, there’s 50% chance it will retaliate and bite another insect on the same square, which is implemented here.

Class BabyGrasshopper : public Grasshopper

Constructor:

BabyGrasshopper(int startX, int startY, StudentWorld \*sw)

Public methods:

1. virtual void doSomething(); //Implementation of actor’s doSomething()

Pseudocode:

{

Calling grasshopper’s protected method, check if already moved/stunned/dead, return immediately if true.

If hitpoints >= 1500

Set this BabyGrasshopper’s status to dead

Tell studentWorld to create a new Adult Grasshopper here

Return

Calling grasshopper’s protected method, eat, move or reset desired distance accordingly.

}

Class Ant : public Insect

Constructor:

Ant(int startX, int startY, int colonyNum, Compiler \*cp, StudentWorld \*sw)

Public Methods:

1. virtual void doSomething();

Pseudocode:

{

Call insect’s protected method, check if already moved/stunned/dead, return immediately if true.

Up to a maximum of 10 times:

Get command from compiler, then switch and execute the command

}

1. virtual bool isAnt(){returns true;};

This is an ant.

1. Int getColonyNum()

Returns the colony number of this ant (0-3).

Class Food : public EnergyHolder

Constructor:

Food(int startX, int startY, int initialAmount, StudentWorld \*sw)

Public methods:

1. virtual void getHurt(int damageType, int damage){};

Food cannot be hurt.

1. int eat(int amount);

Eats the amount of food. Returns the amount of food actually eaten (e.g. when the amount of food available is less than the amount wanted).

1. int add(int amount);

Adds the amount of food. Return the total amount of food.

Class Pheromone : public EnergyHolder

Constructor:

Pheromone(int startX, int startY, int colonyNum, StudentWorld \*sw)

ColonyNum should be 0 to MAX\_ANT\_COLONIES-1.

Public methods:

1. virtual void doSomething();

Reduces the number of hitpoints by 1. SetDead() if hitpoints == 0.

1. virtual void getHurt(int damageType, int damage){};

Pheromones can’t get hurt!

1. int addStrength(int amount);

Returns the actual amount of strength added.

1. int getColonyNum() const {return m\_colonyNum;};

Returns the colony this pheromone belongs to.

1. virtual bool isPheromone() const {return true;};

Identifier. This is a pheromone!

Class Anthill : public EnergyHolder

Constructor:

Anthill(int startX, int startY, int colonyNum, Compiler \*cp, StudentWorld \*sw)

The compiler for this colony is needed to create an anthill.

Public methods:

1. virtual void doSomething();

Pseudocode:

{

Reduces the amount of hitpoints by 1.

If amount of hitpoints <= 0

SetDead

Return

Eat up to 10000 food on the anthill

If food is eaten, return

Spawn ant of the same colony if hitpoints > 2000

If ant is spawned, hitpoints is reduced by 1500

}

1. virtual void getHurt(int damageType, int damage){};

Anthills can’t get hurt lol.

1. Compiler\* getCompiler() const {return m\_cp;};

Returns the compiler for this colony.

1. virtual bool isAnthill() const {return true;};

This is an anthill!

1. int getColonyNum() const {return m\_colonyNum;};

Returns the colony num of that this anthill belongs to.

Class DamagingObject : public Actor

Constructor:

DamagingObject(int imageID ,int startX, int startY, StudentWorld \*sw)

Public methods:

1. virtual void getHurt(int damageType, int damage){};

It’s an object, it can’t get hurt!

1. virtual void damageInsects() = 0;

This is the function that’s called in StudentWorld::move()’s damaging object phase. Every damaging object should have this function.

1. virtual bool isDamagingObject() const {return true;};

Self-explanatory.

Class Poison : Public DamagingObject

Constructor:

Poison(int startX, int startY, StudentWorld \*sw)

Public methods:

1. virtual void damageInsects();

Do 150 points of poison damage to every insect on this spot. Called in StudentWorld::move()’s damaging object phase.

Class WaterPool : Public DamagingObject

Constructor:

WaterPool(int startX, int startY, StudentWorld \*sw)

Public methods:

1. virtual void damageInsects();

Do 2 points of stun damage to every insect on this spot. Called in StudentWorld::move()’s damaging object phase.

Class Pebble : Public Actor

Constructor:

Pebble(int startX, int startY, StudentWorld \*sw)

Public methods:

1. virtual void getHurt(int damageType, int damage){};

You would probably getHurt more if you tried to bite a pebble.

1. virtual bool blocksInsect() const {return true;};

That’s the whole point of a pebble!

B. A List of Unimplemented Functionalities and Well-Know Bugs

1. The stunning and poisoning effect of water pools and poison objects, although working properly, is implemented in a separate function called damageInsects() instead of in their doSomething().

C. List of Design Decisions that I’ve Made and Assumptions

1. Poison objects will only damage each insect once, unless they step off the object and step back onto it (similar to water pools).
2. Actors are not allowed to call public methods of other actors or refer to other actors directly, but instead must always do so through StudentWorld indirectly (e.g. through public methods in StudentWorld like spawnAnt() or damageInsects()).
3. Actor’s variables (e.g. m\_hitpoints) must not be exposed in public, nor should they be modified directly with a public method (e.g. Methods like setHPto100() should not be public). Instead, public methods should be related as closely as possible to their object (e.g. Food has an eat() function, pheromone has an addStrength() function, and insect has a getHurt() function, even though they all modify m\_hitpoints.)

The exception is when objects behave exactly the same (e.g. Ants and BabyGrasshoppers getHurt() exactly in the same way.)

1. All actors must have a pointer to the StudentWorld.
2. Methods should be extendable in order to facilitate development in the future. For example, StudentWorld::damageInsects() allow for the user to vary their damage, damage type and damage target in the same function. This makes it extremely easy to add, for example, a new grasshopper that does 200 damage to all insects in one square in front of it, or a new waterpool that stuns insects for only 1 turn.

D. Testing my classes

1. StudentWorld:
2. int init()

The loading field part was tested by comparing the loaded field directly with the provided in the sample, which is exactly the same. Std::cout statements were also used to ensure that all objects are pushed onto the STL structure.

1. Int move()

This part was mainly tested using std::cout statements that writes the original position of the object, their original HP, the new position of the object and their new HP, or if they’re dead. This is to ensure that the STL structure is updated correctly after each actor has performed an action.

1. Int damageInsects()

This is the function that’s responsible for bite damage, stun damage and poison damage. This function is tested using breakpoints that activate when this function is called, then going line by line through the code and seeing if the damage is applied correctly to the insects.

1. Other public methods, such as SpawnAdultGrasshopper() and hasFoodHere()

The spawning methods were tested using breakpoints, which will stop the simulation whenever a new object is created, which is then verified by looking at the simulation directly. Other methods were usually tested with std::cout statements and fields with only that object (e.g. a field with one food and one babygrasshopper only).

1. Actor:

a. Public methods

Nothing much to test here, as most methods are virtual or pure virtual.

1. EnergyHolder

a. Public methods

All methods here are trivial, such as getHitpoints() (public), setHitpoints() (protected) and addHitpoints (protected).

1. Insect

a. getHurt()

This is tested using std::cout statements that output HP (or stunnedTurns) before getting hurt, HP (or stunnedturns) after getting hurt, and whether if they’re invulnerable to further damage after being hurt (applies to stun and poison damage.)

c. bool takeStartOfTurnActions() (Protected)

This is tested using one insect (a babyGrasshopper) and using breakpoints, going line by line to see if the right action is done.

b. Other methods

The more complicated methods (takeStartOfTurnActions(), moveInDirection()), which are protected methods, are tested using breakpoints and std::cout statements that writes their state before and after the methods are called.

1. Grasshopper

a. bool moveInDirection() (Protected)

This is mostly tested by freezing the simulation and advancing frame by frame to verify that the grasshopper is actually sleeping and moving as expected.

b. bool eatFoodOnCurrentSquare() (Protected)

This is mostly tested using std::cout statements that activate whenever the grasshopper eats food, which shows the amount of food eaten, the original HP and the new HP.

c. void eatAndMove() (Protected)

This is tested using one grasshopper (a babyGrasshopper) and using the debugger, tracing through the function line by line.

d. Other methods

All other methods are trivial here.

1. BabyGrasshopper

a. void doSomething()

This function is tested by using std::cout statements that activate on BabyGrasshopper’s unique action (morph into a AdultGrasshopper), on a field with only 1 BabyGrasshopper.

1. AdultGrasshopper

a. void doSomething()

This function is tested by using std::cout statements that activate on AdultGrasshopper’s unique actions (Biting another insect, and jumping), on a field with only one AdultGrasshopper. It is verified that the AdultGrasshopper will only jump to a square within 10 units away through the std::cout statements.

1. Ant:

a. void doSomething()

This function is tested by writing my own .bug programs, then going line by line to see if the ant behaves exactly like my program.

1. Food:

a. int eat()

This function is tested using a field with only food and one grasshopper (babyGrasshopper), then using std::cout statements to check if the food is eaten properly.

1. Pheromone

a. int addStrength() and void doSomething()

These functions are tested with an ant from my own .bug program, along with std::cout statements that output the status of the pheromone.

1. Anthill

a. void doSomething()

This function is tested using std::cout statements that outputs the status of the anthill (HP, spawning ant, eating, etc.), then advancing frame by frame in the simulation to check if it’s behaving correctly.

1. DamagingObject

a. Public methods

All methods are virtual here, nothing to test.

1. Poison

a. damageInsects

This method is tested using a field with one insect (ant) and poison objects, and std::cout statements that outputs the status (health) of the insect.

1. WaterPool

a. damageInsects

This method is tested using a field with one insect (ant) and WaterPool objects, and advancing frame by frame to see if the ant behaves correctly.

1. Pebble

BlockingInsects: This is tested by advancing the simulation frame by frame and making sure that no insect is able to walk onto the pebble.