[TYPE THE COMPANY NAME]

IAT 265 Final Report

RoboBug Life! project

Kristofer Ken Castro

8/6/2013

Contents

1. A Brief Introduction to the application

RoboBug's Life is a virtual environment application that attempts to create a virtual species called RoboBug as they live in their environment. They are a simple race who collects resources in order to reproduce and sustain their existence. The environment is user-interactive that allows the user to control their population and how they live. For example, you can give the bugs resources to collect by spawning new ones, you can help them find resources by changing the way they move, you can even destroy them by manipulating the weather to create a storm.

2. Features

(Note: I chose to display it in this easy-to-read format instead of the specified, compact, 2-3 paragraphs to list all features since I don't see it possible to do so with the amount of features I've implemented)

2.1.RoboBug Behavior

Each robobug has a unique behavior. The program has implemented two roles: the worker who collects resources and the soldier who protects the workers.

Worker bugs have search radar feature that detects resources below them. When detected, they head towards the detected resource and collect it.

2.2.RoboBug Mobility

Robobug's has the ability to travel using their wings or hover closely to the ground. Users can change their modes by **left clicking on them.**

You can also pull flying bugs toward a direction by **right clicking at a location above the ground**.

2.3.RoboBug Health

Each bug last for 2 minutes. You can recharge their energy by feeding them 1 resource by **right clicking** on the bug that is dying.

2.4.RoboBug Home Base

Bugs return their resources here. They also come here during the night. You can force bugs out of the house by **left-clicking on their house during night time**. Bugs are immune to lightning inside their base.

2.5.Resources

Resources are objects that grow from the land that the Robobug's collect. They spawn during the day time or forcefully spawned via user **clicking on a ground layer.** Resources disappear after a bug has finished extracting minerals from it.

2.6.Multi-layer Ground

The ground is a more complex 3-layer landscape where resources and bugs can travel to.

2.7.Weather Manipulation

The weather can change from day to night time. The switch occurs approximately every minute. However, users can force time of day change using the **up and down arrow keys**.

Users can also change the weather into a violent stormy weather by clicking on the toggle storm button. **There is a 5% chance a lightning can strike a bug every second**

The user can force a lightning strike, during a storm, on a bug by pressing the **space bar key.**

2.8.RoboBug Creation

The user can create more bugs by **clicking on the RoboBug creation cards** at the bottom left after the Robobug's has collected enough resources.

2.9.Music!

The program contains multiple sound effects from mining resources, finishing extraction, return resource sounds, day/night transitions, stormy weather, and lightning strikes.

3. Statistics

Statistic name	value
Lines of Code (LoC)	2200
Average LoC per method	8.69
Average number of fields per class	4.15
Number of packages	10
Number of classes	40

4. Final UML Diagram

Note: my program is actually pretty big to "clearly" view the entire architecture of the program so I removed some class fields and methods in the UML.

Please refer to this link to view the UML Diagram created using an online UML diagram editor:
https://creately.com/diagram/hjdi1fg51

5. Classes

I have many classes that are complex in itself and its containing algorithms. I'll explain by categorizing by packages

5.1.Collision Package

Every collide-able or interactive object has an array list of collision shapes that are the invisible "hit boxes" that the system checks for when interacted upon. Example, clicking a RoboBug -> the system will check if the mouse position is in any of the collision shapes "body", "head" or "claw".

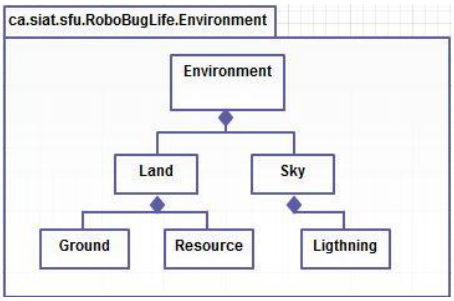
Class Name	Some Methods & Fields	Description
CollisionShape.java	position – origin point of ALL shapes of the image center – center point of the	Parent class

	collision shape	
CollisionRect.java	width and height	Sub-class of collision shape with width and height
CollisionCircle.java	radius	Sub-class of collision shape with radius

5.2.Environment package

The environment is composed of the Land and the Sky and controls them. The Land is composed of ground objects and resources and controls them. The Sky has properties that turns it into day and night and can also be composed of lightning when it's a stormy weather.

Figure 1: Simplified Environment UML



Class Name	Some Methods & Fields	Description
Environment.java	+strikeLightning(PVector position) +spawnResource() : void +updateTimeOfDaySounds() : void	Uses the land and sky it is composed of to control environmental factors. Essentially the controller that controls how the land and sky changes
Ground.java	- image : PImage + drawImage() : void	A block of ground. The land consists of several ground tiles.
Home.java	- image : PImage - collisionShapes : HashMap	The spaceship home base of the bugs. Clickable so we need collision hit box.
Land.java	- resources : ArrayList<Resource> - land : ArrayList<Ground> - collisionRects : HashMap	Land is responsible for functions that grow resources on its land.
Lightning.java	- startPosition : PVector - endPosition : PVector	Simple struct just containing start and end positions of the lightning
Resource.java	- image : PImage -collisionShapes : HashMap -BACK_LAND_Y : float -MIDDLE_LAND_ Y : float -FRONT_LAND_Y : float	Resources that the bugs collect. Note there are constants BACK_LAND_Y, MIDDLE_LAND_Y, and FRONT_LAND_Y. These are y positions of the resources based on their ground layer position
Sky.java	- isStormy : Boolean - isDay : Boolean - isNight : Boolean - numberOfLightning : int - weightOfLightning : float - displacement : float - lightnings : ArrayList - onScreenLightnings : ArrayList	Functions to control the time of day (night and day) as well as creating dangerous weather such as a lightning stormy weather

5.2.1. Complex Algorithms in Environment

5.2.1.1. Creating the lightning effect (Sky.java)

The lightning effect is done through recursive random displacement type of algorithm where I take a line consisting of two points and break it down into several more points in between by recursively creating midpoints and then horizontally displacing them by a random value.

Figure 2: Lightning Effect using Complex Shapes and Recursion

```
1   /**
2    * The function class users will use to strike a lightning from one position to another.
3    *
4    * @param startingPoisiton where you are
5    * @param endPosition
6    */
7   public void drawALightning(PVector startingPosition, PVector endPosition){
8
9       p.beginShape();
10      drawALightningRecursion(startingPosition,endPosition,0, 3);
11      p.endShape();
12  }
```

Figure 3: Lightning Effect Recursive Call

```
1   /**
2    * The function helper class for drawing a lightning that does the recursive calls of
3    * creating the lightning effect.
4    *
5    * @param startPoint of the lightning
6    * @param endPoint target of the lightning
7    */
8   private void drawALightningRecursion(PVector startPoint, PVector endPoint, int currentDetail, int maxDetail){
9       p.stroke(p.random(230,255),p.random(240,255),255);
10      p.strokeWeight(weightOfLightning);
11      p.noFill();
12      float randomDisplacement = p.random(-displacement,displacement);
13      PVector midpoint = new PVector((startPoint.x + endPoint.x)/2 + randomDisplacement,(startPoint.y + endPoint.y)/2);
14
15      if ( currentDetail >= maxDetail){
16          p.vertex(startPoint.x, startPoint.y);
17          p.vertex(midpoint.x, midpoint.y);
18          p.vertex(endPoint.x, endPoint.y);
19          return;
20      }
21
22      drawALightningRecursion(startPoint, midpoint,++currentDetail, maxDetail);
23      drawALightningRecursion(midpoint, endPoint,++currentDetail, maxDetail);
24
25  }
```

The core of the algorithm is this recursive call. Notice how we recursively break a line into "maxDetail" amount of mid points and then randomly displace them by some displacement value. (Line 13 determines the midpoint; line 22 and 23 recursively finds the midpoints of the two lines divided by the current midpoint)

One problem now is to display them long enough so that we can see it. This is done by storing lightning objects in an array list then destroying them some x milliseconds in the future. Look at the figures below to understand.

Figure 4: Displaying lightning via creation and deletion

```
1   /**
2    * Create a lightning that is displayed at the front of the screen (not the background)
3    * @param position
4    */
5   public void createLightning(PVector position){
6       final Lightning l = new Lightning(new PVector(position.x, 0), position);
7       onScreenLightnings.add(l);
8
9       removeLightningAfter(300, l);
10  }
11
12  /**
13   * Remove lightning from display after waiting x milliseconds
14   * @param millisecond to wait before deleting
15   * @param l lightning to delete
16   */
17  private void removeLightningAfter(int millisecond, final Lightning l){
18      // This is a lightning strike effect so we delete the lightning at some future point
19      // when we don't want it to display anymore.
20      Timer timer = new Timer();
21      timer.schedule( new TimerTask(){
22
23          @Override
24          public void run() {
25              onScreenLightnings.remove(l);
26          }
27
28      },millisecond);
29  }
```

Here you see on line 7 we are creating a lightning and then 300 milliseconds later we delete it on line 9. We are making use of Java's Timer class and scheduler method to run a function after some time has passed.

5.2.2. Creating Resources (Land.java)

Since the ground is multi-layered, creating a resource is a little more complex. We first have to find which layer the mouse is currently pointed at then grab the Y value of where we are to place the resource that corresponds to its ground layer Y position. Look at the diagram below for details.

Figure 5: Figuring out which ground layer was clicked

```
/**
     * Returns which layer the player clicked on the ground.  Used to spawn resources
     * @param mouse position that was clicked
     * @return layer
     */
    public Layer layerClicked(PVector mouse){
        Layer layer = null;

        for(String key : this.collisionRects.keySet()){
            CollisionRect currentCollisionRect = collisionRects.get(key);
            PVector topLeft = currentCollisionRect.position;
            PVector topRight = new PVector(topLeft.x + currentCollisionRect.width, topLeft.y);
            PVector bottomLeft = new PVector(topLeft.x, topLeft.y + currentCollisionRect.height);

            if ( mouse.x >= topLeft.x && mouse.x <= topRight.x  //check x boundaries
                && mouse.y >= topLeft.y && mouse.y <= bottomLeft.y  ){ // check y boundaries
                layer = new Layer();
                if ( key.equals("back") )
                    layer.setToBack();
                else if ( key.equals("middle") )
                    layer.setToMiddle();
                else
                    layer.setToFront();
            }
        }
        return layer;
    }
```

Figure 6: Creating a resource

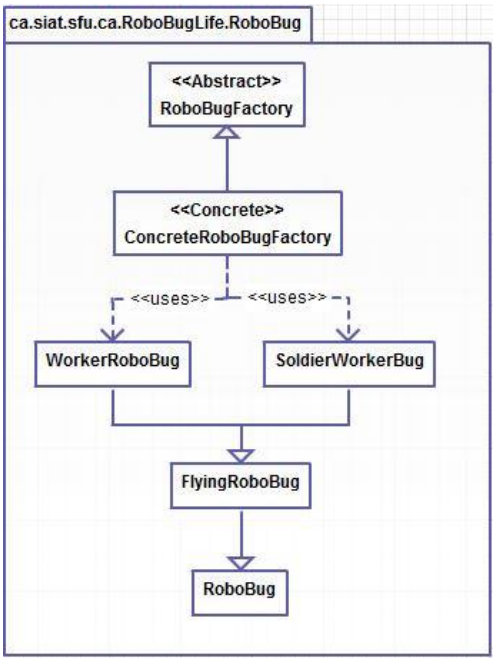
```
    /**
     * Create resource based on mouse position.
     * @param mouse position where to create resource
     */
    private void createResource(PVector mouse){
        Layer layer = layerClicked(mouse);
        PVector groundLocation = new PVector(mouse.x, 0);
        if (layer == null) return;
        if (layer.getLayer().equals("back")){
            groundLocation.y = Resource.BACK_LAND_Y;
        }else if ( layer.getLayer().equals("middle") ){
            groundLocation.y = Resource.MIDDLE_LAND_Y;
        }else if ( layer.getLayer().equals("front") ){
            groundLocation.y = Resource.FRONT_LAND_Y;
        }
        resources.add(new Resource(p, groundLocation, layer));
    }

```

Notice that in order to place the resource at the correct spot, we use the x position of the mouse, but not the y position; we use the constant y position of all resources that spawn in a specific ground layer.

5.3. RoboBug Package

Robobugs are a species that can have different type of roles and behaviors but at the same time sharing common species properties. I incorporate the use of generalization, specialization, and factory pattern to implement this description.

Figure 7: RoboBug Package showing factory architecture



Class Name	Some Methods & Fields	Description
RoboBugFactory		Abstract factory class
ConcreteRoboBugFactory	+ createRoboBug(type: String) : void + createRoboBug(type: String, position: PVector) : void	Concrete robobug factory class
RoboBug	+ BACK_LAND_Y : float + MIDDLE_LAND_Y : float + FRONT_LAND_Y : float - position : PVector - velocity : PVector - collisionShapes : HashMap # currentHealth - DIES_EVERY_X_SECONDS : int -layer : Layer # setRandomGroundPosition() : void # checkBoundaryCollisions() : void # drawHealthBar() : void	The highest parent class of all RoboBug. Common methods and properties contains here
FlyingRoboBug.java	-aboutToFly : Boolean -isFlying : Boolean -radiusNoise : float -radius : float -wingCenter : PVector -wingPosition : PVector -wingLastPosition : PVector -angle : float # attractionPoint : PVector	Bugs that can fly have specialized display and “attract-able” behaviors.

	- isAttracted : boolean - checkBoundaryCollisions() : void - checkHitGround() : void - drawWings() : void + dropStraightDownToCurrentGround() : void + dropToCurrentGround() : void + attract() : void + reachAttractionPoint(): boolean + startFlying(): void + stopFlying (): void	
WorkerRoboBug.java	- image : PImage - extractedResourceImage : PImage - collectinResource : Boolean - isHoldingAResource : Boolean - checkReachDestination() : void + collectResource(resourcePosition : PVector) : void - collectingResource(resourcePosition : PVector) : void - stopCollectingResource() : void - drawImage() : void	Bugs that collects resources
SoldierRoboBug.java	- image : PImage - MIN_ATTACK_DAMGE : int - MAX_ATTACK_DAMAGE : int + draw () : void + drawImage () : void	Bugs that protects the other robobugs from enemies. (Doesn't do much in this version)

5.3.1. Complex Algorithms in RoboBug

5.3.1.1. Collision Handling in General

Collision handling in this program in general is done using multiple collision shapes per images. The figure below demonstrates that the RoboBug has 3 body part collision circles.

Figure 8: Using CollisionShapes for collision handling

```
    // body
    float collisionBodySize = 20*scaleFactor;
    float bodyOffsetX = -2*scaleFactor;
    float bodyOffsetY = 15*scaleFactor;
    CollisionShape bodyCollisionShape = new CollisionCircle(new PVector(position.x, position.y),
        new PVector(bodyOffsetX,bodyOffsetY), collisionBodySize/2);
    .
    .
    .
    collisionShapes.put("Body", bodyCollisionShape);
    collisionShapes.put("Head", headCollisionShape);
    collisionShapes.put("Claw", clawCollisionShape);
} // end of initializeCollisionShapes
```

We then detect collision by checking that the distance between any two object's collision shapes is less than the sum of their radius. The figure below demonstrates how we detect collision using collision shapes.

Figure 9: Collision detection using collision shapes

```
/** checks to see if the bug has been clicked
 *
 * @return if the bug has been clicked or not
 */
public boolean isClicked(){
    for(String shapeKey : collisionShapes.keySet()){
        if (!shapeKey.equals("Search Range")){
            CollisionCircle circle = (CollisionCircle)collisionShapes.get(shapeKey);
            if(p.dist(circle.center.x, circle.center.y, p.mouseX, p.mouseY) <= circle.radius){
                return true;
            }
        }
    }

```

5.3.2. Flying Animation Effect using Perline Noises

The "wings" of flying objects are done using Perline noise generated spirals.

Figure 10: Flying effect using perline noise with spirals

```
/**
 * Draw a perline noise generated wings or flight animation
 */
private void drawWings(){
    radiusNoise = p.random(50);
    radius = 2;
    p.stroke(p.random(0, 50), p.random(0, 50), p.random(0, 50), 80);
    p.strokeWeight(1.2f);
    float startangle = p.random(360);
    float anglestep = 0.9f;// 1 + (int) random(2);

    for (float ang = startangle; ang <= 500 + p.random(100); ang += anglestep) {
        radiusNoise += 0.01;
        radius += 0.009;

        float noiseRadius = radius + (p.noise(radiusNoise) * 200) - 100;
        float rad = p.radians(ang); // converts angle to radians

        p.pushMatrix();
        p.translate(wingCenter.x, wingCenter.y);
        wingPosition.x = noiseRadius * p.cos(rad);
        wingPosition.y = noiseRadius * p.sin(rad);
        if (wingLastPosition.x > -999){
            p.stroke(p.random(0, 50), p.random(0, 50), p.random(0, 50), 50);
            p.strokeWeight(p.random(1.2f,2));
            p.line(wingPosition.x, wingPosition.y, wingLastPosition.x, wingLastPosition.y);
            p.stroke(191,186,54,50);
            p.strokeWeight(p.random(1.5f,3f));
            p.line(wingPosition.x+p.random(-5,5), wingPosition.y+p.random(-5,5), wingLastPosition.x+p.random(-5,5),
                wingLastPosition.y+p.random(-5,5));
        }
        p.popMatrix();
        wingLastPosition.x = wingPosition.x;
        wingLastPosition.y = wingPosition.y;
    }
}
```

5.3.3. Attraction algorithm

Attracting a robobug to a point requires understanding of basic trigonometry. I find the angle between the current position and the attraction point using inverse tan function (since we know the length of the x and y). We then find the velocity using algebra. Once velocity is set, the bug should move towards the attraction point direction over time. Look at the figure below for details.

Figure 11: Attraction algorithm

```
/**
 * start attracting the bug from its current position to the stored attraction point.
 * Keep attracting, once it reaches destination, stop attracting.
 */
public void attract(){
    float marginOfError = 2;
    if(p.dist(position.x, position.y, attractionPoint.x, attractionPoint.y) >= marginOfError){
        setAttracted(true);
        angle = p.atan2(attractionPoint.y - position.y, attractionPoint.x - position.x);
        velocity.x = this.maxSpeed*p.cos(angle);
        velocity.y = this.maxSpeed*p.sin(angle);
    }else{
        this.stopAttracting();
    }
}
```

Note there is a marginOfError that we check against instead; we can't precisely get the position of the bug to equal the attraction point due to the value of velocity as it changes the position over time. (In other words, can't get the distance between them to equal exactly zero)

5.3.4. Start Flying Algorithm

The algorithm works as expected: if the bug is on the ground, launch it to the air then continue flying in the air. The algorithm does exactly that, it launches the bug into the air at a max velocity (line 9 and 10), then transitions to a random velocity (line 18) for the bugs "cruise" mode flying. Look at the figure below for details. Notice the use of Java's scheduler to transition from launch flight to normal flight.

Figure 12: Start flying algorithm

```
1  /**
2   * launching to fly (initially move at a higher speed to elevate then slow down)
3   */
4  public void startFlying(){
5
6      if (onGround){
7          aboutToFly = true;
8          onGround = false;
9          float speedX = maxSpeed;
10         if ( velocity.x < 0) speedX *= -1;
11
12         velocity = new PVector(speedX,-3);
13         Timer timer = new Timer();
14         timer.schedule(new TimerTask(){
15
16             @Override
17             public void run() {
18                 velocity = new PVector(p.random(minSpeed, maxSpeed),p.random(-1,1));
19                 aboutToFly = false;
20             }
21
22         },1000);
23     }
24 }
```

5.3.5. Collecting Resource Animation Algorithm

Collecting resource is an algorithm where once you are given the resource position, start collecting the resource, after some seconds later when you are done extracting, stop collecting resource and return home. Then once returned, resume back to searching for resources.

Figure 13: Collecting resource algorithm

```
1  /**
2   * Collect resource animation
3   * @param resourcePosition the location of the resource
4   */
5  public void collectResource(PVector resourcePosition){
6      if ( returnHome == false){
7          stopMoving();
8          collectingResource = true;
9          .
10         . // play some sound effect
11         .
12         collectingResource(resourcePosition);
13
14         // return home
15         Timer timer = new Timer();
16         timer.schedule(new TimerTask(){
17
18             @Override
19             public void run() {
20                 .
21                 . // play some sound effect
22                 .
23                 returnHome = true;
24                 startFlying();
25                 stopAttracting();
26                 setAttractionPoint(new PVector(150, p.height/2-250));
27             }
28
29         },2000);
30     }
31 }
```

Line 6 makes sure that we stop running this function when we already extracted the resource so stop trying to collect the resource. Line 7 and 8 makes sure we stop moving while collecting the resource and change our state. Line 12 draws the collecting resource animation. Line 15-29 is the block of code that runs after 2 seconds; that is, start going home to return the resource and set the attraction point to where the home base is.

5.4. Main Package

The main package contains the Main.java and IntroScreen.java files. These are the files that are essential for the program to start and run.

Class Name	Some Methods & Fields	Description
IntroScreen.java	- introImage : PImage - startButton : Button - isVisible : Boolean - startButtonClickHandler() : void + isVisible() : boolean	The intro screen containing the "how-to" interact picture and button to transition to the main program
Main.java	- bugs : ArrayList - bugFactory : RoboBugFactory - controlHUD : ControlHUD - introScreen : IntroScreen - setupLightningStrikeInterval(int) : void - setupSpawnMineralsInterval(int) : void	The central point of the program where it controls all other classes to create an interactive environment

- startDecreaseHealthOfBugs() : void
- checkMoveBugsHome() : void
- removeResources() : void
- setupRoboBugCreationListeners() : void
- setupWeatherManipulationListeners() : void
- setupBugResourceCollection () : void
- checkBugClicked() : void
- moveFlyingBugsToLocation () : void
- isAnyBugsClicked() : boolean
- checkHomeBaseClicked() : void
- checkRechargeClickedBugs() : void
- checkForDeadBugs() : void
- srikeRandomBug() : void
- resetBugScared() : void
- decreaseHealthOfBugs() : void

5.4.1. Complex Algorithm in Main

This is a pretty big program so there are several "complex" algorithms but the most significant and complex one that most applies to a feature is the one collecting resource.

5.4.1.1. Collecting Resource

Collecting a resource can be simplified in the following pseudo-algorithm:

- Iterate over all bugs
- Isolate only worker bugs since only workers can collect resources
- For each worker bug's collision shapes check for collision with the resources collision shape
- When we find a collision we look check for two things
    - If we collided the resource using the bug's "seach range" collision circle we pull ourselves towards the resource
    - If we collided the resource using the bug's "claw" and our ground level is the same as the resource's ground level then we start extracting the resource.

The details and implementation are shown in the figure below.

Figure 14: Collecting Resource

```
/**
     * checks if any of the bugs has collided with a resource, if so start collecting
     */
    private void bugResourceCollision(){
        ArrayList<Resource> resources = environment.getResources();
        for(int bugI = 0; bugI < bugs.size(); bugI++){
            for( int resI = 0; resI < resources.size(); resI++){
                RoboBug currentBug = bugs.get(bugI);

                // only worker bugs can collect resource so check if we're dealing with workers
                if(currentBug instanceof WorkerRoboBug){

                    // store values for easier readability and caching performance
                    WorkerRoboBug currentWorkerBug = (WorkerRoboBug) currentBug;
                    Resource currentResource = resources.get(resI);

                    // go through each collision shape of the bug  to check for collision
                    for(String bugShapeKey: currentBug.getCollisionShapes().keySet()){

                            CollisionCircle bugCircle = (CollisionCircle) currentBug.getCollisionShapes().get(bugShapeKey);

                            // go through each collision shape of the resource to check for collision with the bug
                            for(CollisionShape resourceShape : currentResource.getCollisionShapes().values()){
                                CollisionCircle resourceCircle = (CollisionCircle) resourceShape;

                        // if we have a collision
                        if (bugCircle.center.dist(resourceShape.center) <= bugCircle.radius + resourceCircle.radius){

                            // if the collision between the bug's search range and resource is vertically aligned
                            if(bugShapeKey.equals("Search Range") && Math.abs(currentBug.position.x - resourceCircle.center.x) <= 10){

                                // if the bug is not on the ground, bring it to the ground layer, that is where the resource is!
                                if( !currentWorkerBug.onGround ){
                                    currentWorkerBug.changeLayer(currentResource.layer);
                                    currentWorkerBug.dropStraightDownToCurrentGround();
                                }
                            }else{ // if we are colliding, not the search range, but a real bug body part

                                // if the body part is the claw (which is what bugs use to extract resource) and on the ground level of the resource
                                if(currentBug.layer.equals(currentResource.layer)
                                        && bugShapeKey.equals("Claw")){

                                    // start collecting resource animation
                                    currentWorkerBug.collectResource(bugCircle.center);

                                    // if the bug doesn't have a resource holding yet, add this one
                                    if ( !currentWorkerBug.hasResource()){
                                        currentWorkerBug.setResource(currentResource);
                                    }
                                }
```

5.5.HUD Package

The HUD contains information such as the resources collected, the run time, and buttons for users to click. Any external display not part of the virtual environment is contained here.

Class Name	Some Methods & Fields	Description
ControlHUD.ja	-resourceLogo : PImage -resourceCountFont : PFont -workerCard : PImage -soldierCard : PImage -lightningCard : PImage -buyWorker : Button -buySoldier : Button -toggleStorm : Button -createWorkerListeners : ArrayList -createSoldierListeners : ArrayList -toggleStormListeners : ArrayList	Handles all display external from the virtual environment. i.e. buttons, statitsics, timers.

	+ addCreateWorkerChangeListener (listener : ChangeListener) + notifyCreateWorkerChangeListeners() + addCreateSoldierChangeListener (listener : ChangeListener) + notifyCreateSoldierChangeListener (listener : ChangeListener) + addToggleStormChangeListener (listener : ChangeListener) + notifyToggleStormChangeListeners() + drawCards() + drawTimer() + drawResourceText()

5.6. User Package

This is package contains the User.java singleton class that maintains all user statistical data.

Class Name	Some Methods & Fields	Description
User.java	- static manager : User + resources : int - static getInstance() : User	Maintains all user statistical data. Currently only keeps track of number of resources

5.7. Utility Package

This package has helper classes used in the program.

Class Name	Some Methods & Fields	Description
RGB.java	+ red : int + green : int + blue : int + alpha : int	Struct to hold RGBA values
Layer	+ middle : boolean + back : boolean + front : boolean + getLayer() : String	Simple class that bugs and resoruces uses to describe which layer they are in the ground

6. Main Application

6.1.How are your objects created, stored, and destroyed?

Objects are stored in two data structures and using their add/put method: ArrayList and HashMap. HashMap is only used to store collision shapes. HashMap is used since you need to provide a key and a value. The key is used to provide descriptive value to objects being stored. For example, if we are dealing with “head” collision shape then we get it from the HashMap of collision shapes via collisionShapes.get (“head”) -- very descriptive. They are destroyed using the data structures built-in remove functions.

There are two types of object creations in this program: one via a user-class using a factory method to create a specific object and one via the user-class simply using the “new” keyword.

Figure 15: Creating objects using Factory

```
1  public class Main extends PApplet{
2
3      ArrayList<RoboBug> bugs;
4      RoboBugFactory bugFactory;
5      .
6      .
7      .
8      public void setup(){
9          bugFactory = new ConcreteRoboBugFactory(this);
10         .
11         .
12         .
13         bugs = new ArrayList<RoboBug>();
14         bugs.add(bugFactory.createRoboBug("Worker"));
15         bugs.add(bugFactory.createRoboBug("Soldier"));
16     }
```

Lines 14-15 shows one way of creating objects, through a factory method namely createRoboBug(String type)

Figure 16: Create resource object using new keyword

```
1  /**
2   * Create resource at a random ground layer (back, middle, or front)
3   */
4  void createResource(){
5      Layer layer = new Layer();
6
7      PVector resourceGroundPosition = new PVector();
8      int chance = (int) (Math.random()*3);
9      if( chance == 0){
10         layer.setToBack();
11         resourceGroundPosition = new PVector(p.random(100, p.width),Resource.BACK_LAND_Y);
12     }else if (chance == 1){
13         layer.setToMiddle();
14         resourceGroundPosition = new PVector(p.random(100, p.width), Resource.MIDDLE_LAND_Y);
15     }else{
16         layer.setToFront();
17         resourceGroundPosition = new PVector(p.random(100, p.width), Resource.FRONT_LAND_Y);
18
19     }
20     resources.add(new Resource(p, resourceGroundPosition, layer));
21 }
```

Line 20 shows the more traditional way for a class user to create objects, using the “new” keyword and following the method signature of the object.

Figure 17: Removing objects using data structures remove methods

```
/**
 * remove resources that has already been used up
 */
private void removeResources() {
    for(int i = 0 ; i < bugs.size() ; i++){
        RoboBug bug = bugs.get(i);
        if (bug instanceof WorkerRoboBug){
            WorkerRoboBug currentWorkerBug = (WorkerRoboBug) bug;
            if(currentWorkerBug.isReturningHome() && currentWorkerBug.hasResource()){
                ArrayList<Resource> resources = environment.getResources();
                if (resources.contains(currentWorkerBug.getResource())){
                    resources.remove(currentWorkerBug.getResource());
                }
            }
        }
    }

}
```

6.2.How many states are there in your program?

There are several states per object in my program.

Environment has 2 states: day time and night time. Night time has another state of being a stormy weather or not.

The land has two states depending on the time of day. During day time, resources can grow increasing the number of resources they hold. During night time, resources don't grow. The state of the land can also change by bugs eating up their resource which decrease the lands resources.

Robobugs have plenty of states. A bug can be flying or on the ground. A worker bug can be collecting resources or searching for resources. A bug can be attracted to a point or free to move. A bug can be scared of the night or not.

In a more "big-picture" perspective, the virtual environment has the initial state of starting with two bugs and can progress to a state of varying population depending on user interaction or ultimately end with all bugs dying when left without aid.

6.3.How are objects iterated?

Objects are generally iterated using a for-loop or Java's for each implementation. Examples are shown in the figures below.

Figure 18: Iterate using for-each and for-loop

```
1 public void draw(){
2     for(Ground ground : land){
3         ground.draw();
4     }
5
6     for ( int i = 0 ; i < resources.size() ; i++){
7         resources.get(i).draw();
8     }
9 }
```

Lines 2-4 show the for-each way of iterating. Lines 6-8 show the for-loop way of iterating.

6.4.How interactions between objects are handled

Interactions between objects are handled by checking collision between their collision shapes. For example, collision between a worker bug and resource. Look at the figure below for details using an example and section 4.3.1.1 for collision handling in general. Notice in lines 18, we are iterating over the bugs collision shape and iterating over the resources collision shapes in line 23 to check if there is a collision in any combination of them in line 27. If collision happens we act accordingly (not shown here) such as check if the collision was the search range which we then proceed to pull the bug towards the resource, if it was the claw of the bug that collided with the resource then we start collecting resource.

Figure 19: object interaction using colliding collision shapes

```
1  /**
2   * checks if any of the bugs has collided with a resource, if so start collecting
3   */
4  private void bugResourceCollision(){
5      ArrayList<Resource> resources = environment.getResources();
6      for(int bugI = 0; bugI < bugs.size(); bugI++){
7          for( int resI = 0; resI < resources.size(); resI++){
8              RoboBug currentBug = bugs.get(bugI);
9
10             // only worker bugs can collect resource so check if we're dealing with workers
11             if(currentBug instanceof WorkerRoboBug){
12
13                 // store values for easier readability and caching performance
14                 WorkerRoboBug currentWorkerBug = (WorkerRoboBug) currentBug;
15                 Resource currentResource = resources.get(resI);
16
17                 // go through each collision shape of the bug  to check for collision
18                 for(String bugShapeKey: currentBug.getCollisionShapes().keySet()){
19
20                         CollisionCircle bugCircle = (CollisionCircle) currentBug.getCollisionShapes().get(bugShapeKey);
21
22                         // go through each collision shape of the resource to check for collision with the bug
23                         for(CollisionShape resourceShape : currentResource.getCollisionShapes().values()){
24                             CollisionCircle resourceCircle = (CollisionCircle) resourceShape;
25
26                             // if we have a collision
27                             if (bugCircle.center.dist(resourceShape.center) <= bugCircle.radius + resourceCircle.radius){
28                                 .
29                                 . // do stuff after detected collision
30                                 .
31                             }
32                         }
33                 }
34             }
35         }
36     }
37 }
```

7. Describe How You Met the Following Requirements

Patterns used: Observer Pattern, Abstract Factory Pattern, Singleton Pattern

7.1. Inheritance by superclass, abstract class, and interface

Here is one example per requirement (there are many more examples in the program):

WorkerRoboBug -> FlyingRoboBug (non-abstract super class)
ConcreteRoboBugFactory -> RoboBugFactory (abstract superclass)
Resource -> IDrawableWithImage (interface)

7.2.Polymorphism via overriding

Draw method of WorkerRoboBug and SoldierRoboBug is overridden from the draw method of FlyingRoboBug but each calls the super.draw() since they share wings animation but have different image files.

7.3.Method OverLoading

You can create resources and bugs in two ways: specified position or random. The system randomly positions automatically generated resources but uses specified position creation when the user forcefully wants to create a resource by clicking the ground.

Figure 20: Create resource randomly

```
/**
 * Create resource at a random ground layer (back, middle, or front)
 */
void createResource(){
    Layer layer = new Layer();

    PVector resourceGroundPosition = new PVector();
    int chance = (int) (Math.random()*3);
    if( chance == 0){
        layer.setToBack();
        resourceGroundPosition = new PVector(p.random(100, p.width),Resource.BACK_LAND_Y);
    }else if (chance == 1){
        layer.setToMiddle();
        resourceGroundPosition = new PVector(p.random(100, p.width), Resource.MIDDLE_LAND_Y);
    }else{
        layer.setToFront();
        resourceGroundPosition = new PVector(p.random(100, p.width), Resource.FRONT_LAND_Y);

    }
    resources.add(new Resource(p, resourceGroundPosition, layer));
}
```

Figure 21: Overloaded create resource via mouse position

```
/**
 * Create resource based on mouse position.
 * @param mouse position where to create resource
 */
private void createResource(PVector mouse){
    Layer layer = layerClicked(mouse);
    PVector groundLocation = new PVector(mouse.x, 0);
    if (layer == null) return;
    if (layer.getLayer().equals("back")){
        groundLocation.y = Resource.BACK_LAND_Y;
    }else if ( layer.getLayer().equals("middle") ){
        groundLocation.y = Resource.MIDDLE_LAND_Y;
    }else if ( layer.getLayer().equals("front") ){
        groundLocation.y = Resource.FRONT_LAND_Y;
    }
    resources.add(new Resource(p, groundLocation, layer));
}
```

7.4.Design Patterns

7.4.1. Association

Since aggregation is just a stronger form of association I'll provide that.The main class made up of WorkerRoboBugs and SoldierRoboBugs. Another example is, we have RoboBugs and Resources that contains a property called layer.

7.4.2. Composition

Environment is composed of land, sky, and home.

Figure 22: composition

```
public class Environment {
    PApplet p;
    Land land;
    Sky sky;
    Home home;
```

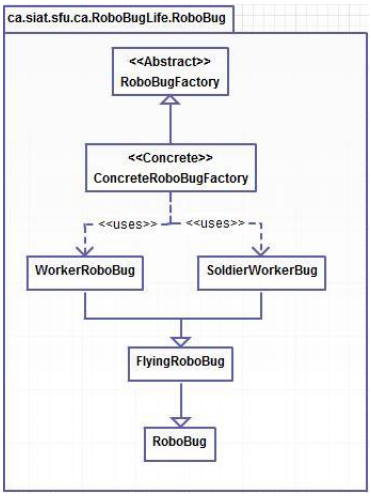
7.4.3. Generalization

Generalization is a way to describe inheritance that common features are pushed up or generalized into methods in a super class. You can see examples of this in WorkerRoboBug and SoldierRoboBug being a sub-class of the super-class FlyingRoboBug; they share common behaviors such as having wings so we generalize the method that draws the wings in the FlyingRoboBug superclass.

7.4.4. Abstract Factory Pattern

RoboBugs are created via using the RoboBugFactory. Look at a simplified uml diagram below, or check the full UML for more details.

Figure 23: Simplified UML RoboBug factory



7.4.5. Singleton (extra)

The ca.siat.sfu.RoboBugLife.User class

Figure 24: Singleton User

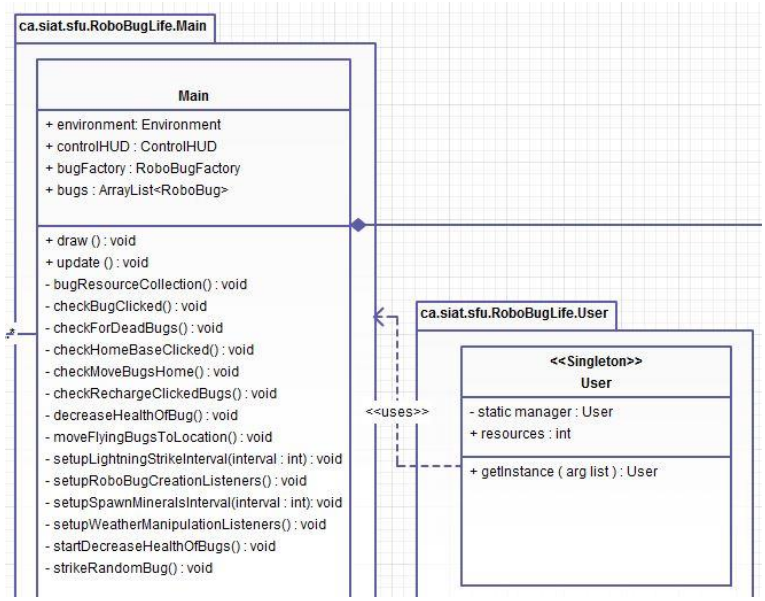


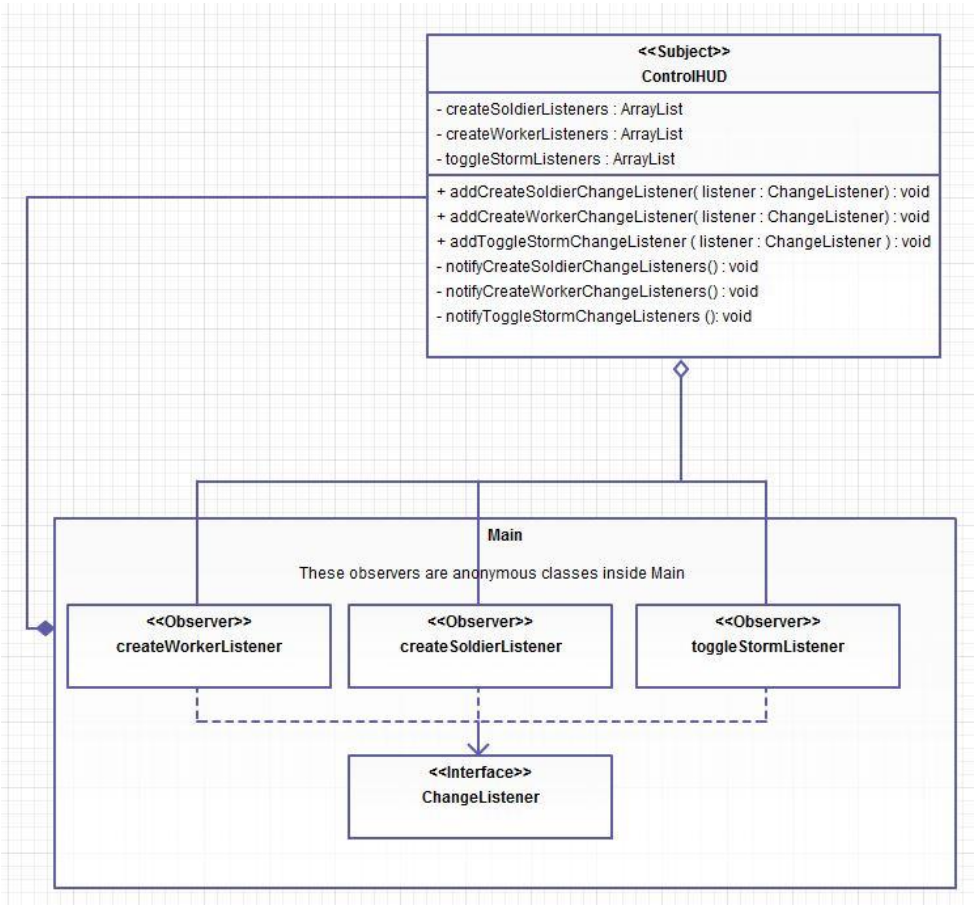
Figure 25: Singleton class

```
1  package ca.siat.sfu.RoboBugLife.User;
2
3  /**
4   * Singleton class that keep tracks of user statistics such as resources.
5   * @author Kristofer Ken Castro
6   * @date 8/5/2013
7   */
8  public class User {
9
10     private static User manager = new User();
11
12     public int resources;
13
14     private User(){
15         resources = 0;
16     }
17
18     public static User getInstance(){
19         return manager;
20     }
21
22 }
```

7.4.6. Observer (extra)

The ca.siat.sfu.RoboBugLife.HUD.ControlHUD class

Figure 26: Observer Pattern via anonymous classes observers



Notice that this isn't the traditional observer pattern. Here we have the observers declared as anonymous classes inside Main and then added as an observer in the control HUD. This is shown in the figure below.

Figure 27: Anonymous Classes as observers

```
/**
 * Listener functions that subscribes to the ControlHUD that contains
 * the buttons for creating RoboBugs.  We handle creation here and listen
 * for click events in the HUD.
 */
private void setupRoboBugCreationListeners(){
    ChangeListener createWorkerListener = new ChangeListener(){

        @Override
        public void stateChanged(ChangeEvent e) {
            if ( User.getInstance().resources >= 2){
                System.out.println(environment.getHome());
                bugs.add(bugFactory.createRoboBug("Worker",environment.getHome().getCenterPosition()));
                User.getInstance().resources -= 2;
                controlHUD.playPurchaseSound();
            }
        }

    };
    controlHUD.addCreateWorkerChangeListener(createWorkerListener);

    ChangeListener createSoldierListener = new ChangeListener(){

        @Override
        public void stateChanged(ChangeEvent e) {
            if ( User.getInstance().resources >= 3){
                bugs.add(bugFactory.createRoboBug("Soldier",environment.getHome().getCenterPosition()));
                User.getInstance().resources -= 3;
                controlHUD.playPurchaseSound();
            }
        }

    };
    controlHUD.addCreateSoldierChangeListener(createSoldierListener);
}
```

7.5.Refactoring

How do you show refactoring was used when the end deliverable code is in one state. Must show refactored vs. refactored versions. I can only talk about some of them.

I've moved properties from FlyingRoboBug that was generalized up to its RoboBug superclass such as health and tint (for fading images).

I've performed "add parameter" refactor technique since a method needs more information from its caller; I achieved this in Resource and Bug creation where I need an extra position parameter that tells the system where to create the objects. I added a position parameter but in an overloaded function. Look at RoboBugFactory.java:

```
public abstract RoboBug createRoboBug(String type);
public abstract RoboBug createRoboBug(String type, PVector position);
```

I performed "pass whole object as parameter" technique for bugs and resources since they needed a notion of knowing if their ground layer is back, middle, or front. I originally had back, middle, front

boolean variables passed along with the creation method signature, then I refactored it into passing only a Layer object that encapsulates the middle, back, and front boolean variables.

```
public Resource(PApplet p, PVector position, Layer layer)
```

I performed "introduce explaining variables" technique for all my constants. For example the representation of the Y position of where resources or bugs are supposed to be placed based on their ground layer. Instead of just providing the value directly, I created constants called BACK_LAND_Y, MIDDLE_LAND_Y, FRONT_LAND_Y.

```
public static final float BACK_LAND_Y = Land.groundPosition-15;
public static final float MIDDLE_LAND_Y = Land.groundPosition+5;
public static final float FRONT_LAND_Y = Land.groundPosition+20;
```

7.6.Exception Handling

Wherever images or sound is loaded NullPointerException is caught. Whenever we're dealing with arrays we can try to catch an index out of bound error.

Advantage of exception handling is that your program can continue to run even if there is a runtime error. More importantly, you can provide a more specific error message.

Figure 28 Catching NullPointerException for sound

```
private void initializeSound(){
    minim = new Minim(p);
    try{
        morningSoundPlayer = minim.loadFile(MORNING_SOUND_PATH,2048);
        morningSoundPlayer.play();
        nightSoundPlayer = minim.loadFile(NIGHT_SOUND_PATH,2048);
        nightSoundLoopPlayer = minim.loadFile(NIGHT_SOUND_LOOP_PATH,2048);
        nightSoundLoopPlayer.setGain(-20.0f);
        stormySoundPlayer = minim.loadFile(STORMY_SOUND_LOOP_PATH,2048);
        stormySoundPlayer.setGain(stormySoundPlayer.getGain()+20.0f);
        lightningSoundPlayer = minim.loadFile(LIGHTNING_SOUND_PATH,2048);
        lightningSoundMissPlayer = minim.loadFile(LIGHTNING_SOUND_MISS_PATH);
    }catch(Exception e){
        System.out.println(e.getStackTrace());
        System.out.println("something is wrong with the sounds from environment.java, initializeSound() method.");
    }
}
```

Figure 29: Capturing ArrayIndexOutOfBoundException for array iterations

```
/**
 * check if the bug is clicked on
 * @param bug
 * @return if a bug has been clicked
 */
private boolean isAnyBugsClicked(){
    try{
        for(int i = 0 ; i < bugs.size(); i++){
            RoboBug bug = bugs.get(i);
            if (bug.isClicked()) return true;
        }
    }catch(ArrayIndexOutOfBoundsException e){
        System.out.println("tried to call isAnyBugsclicked but index > bugs.size()");
    }
    return false;
}
```

8. Summarize the application

8.1.What worked?

Every implemented feature listed in section 2 works. Refer to feature list for list of what worked.

8.2.What didn't?

Some of the proposed features were changed into something better or different. For example, I originally propose there will be clouds and clicking on it can turn it into a stormy cloud. This was changed into no clouds (didn't like the visual object of the cloud in my current virtual environment) and activating it is done through clicking a ControlP5 button.

What kind of doesn't work exactly that was originally proposed was the feature: "only one bug can extract resource at a time." Right now as long as the resource is there any bug can extract from it, but once a bug is done the resource disappears. However, this doesn't stop bug A from extracting and in between the extracting, another bug extracting the same resource. So there is a chance that multiple bugs can still extract from a resource only when another bug is already extracting from it. Not a big bug.

Another feature that was "implied" but not actually proposed since it was only seen in the story board was some type of text story caption that narrates what is going on in the environment. This feature was never proposed but a feature that can potentially be included in the future. It is kind of complex with so many things that can go on in the environment (scalability issues).

8.3.What needs to be done?

As far as this version is concern it is complete. Aside from maybe adding more static background pictures to make it look nice. Look at section 7.5 for next step however.

8.4.Are you happy with the results?

I am happy with the results overall. I gain a lot of experience doing advance collision detection with multiple collision shapes, a lot of interactive types using mouse and keyboard, experience computer-generated special effects using perline noise, improve my photoshop skills with character design, and took my programming to the next level by incorporating many design patterns in my program

8.5.What would be your next step (for further tweaking and improvement)

My next step would be the following:

- Make graphics a little better by introducing more background images to create a more nature-like virtual environment.
- Complete the soldier bug and introduce enemy bugs that they interact with
- Add more violent weathers, perhaps tornado, earthquakes
- Add more animation