Darwin 2.0, Part I

Due 11:59 pm Mon, April 21, 2008

1 Assignment

Download the SDK and programmer's guide from http://cs.williams.edu/~morgan/cs136/darwin2.0

This lab is divided into two parts. This week you will implement a subclass of Creature within the Darwin simulator API that supports the helper methods described in the following section. Next week you will use these helper methods to implement your own strategy for winning a significant majority of Natural Selection competitions against Pirate, Rover, SuperRover, and Flytrap, and then (optionally) enter your Creature into a contest.

For full credit, your solution this week must have the following properties:

1. A single Java source file[1] containing a public Creature subclass named "SmartCreature"
   a. With all of the helper methods described in the following section
   b. Override getAuthorName and getDescription so that we know who you are.
2. Exercise good programming discipline with regard to efficiency, readability and documentation

You may want to begin crafting your creature for the competition this week. You can see movies of some recent battles competitions between creatures written by professional programmers and students at other schools on the Darwin 2.0 page. You can test your creature on any of the ns_*.txt maps. You will probably want to make your own simple maps for early testing, as well as more devious maps when you're trying to beat other creatures. Note that you're playing Natural Selection mode this week, not Maze running. See the Darwin documentation for details. You can reserve names for your creatures on the wall of the lab.

Submit your solution using the commands:

turnin –c 136 SmartCreature.java

1.1 Helper Methods

You must implement all of the following helper methods. You may need or desire additional helper methods. Each should be fully documented, implemented efficiently, and carefully guarded against thread synchronization problems. In the comments for each method, explain how your design is threadsafe. **You can't just write synchronized everywhere**—that will be slow and may lead to deadlocks! Some of these methods are relatively easy and some are much more challenging. The recall method and its helpers are probably more challenging than all of the others combined, so plan accordingly.

Be aware that to complete Part II next week, you will need all of these methods, so you're going to have to implement them even if you do not submit a complete solution this week.

void turn180()
Turns to face in the opposite of the current direction

void turn(Direction d)
Turns to face in direction *d*

void turn90Random()
Turns left or right with equal probability

boolean moveForward(int n)

[1] Even if you create multiple classes. Note that in Java, multiple classes may be placed in the same file as long as only one of them is public. You can also use inner classes.

Moves forward *n* spaces. If obstructed, immediately stops trying to move and returns false. Returns true if successfully moved this distance

Direction directionTo(Point p)
Returns the direction to face to move towards *p* along the axis with the largest difference

void turn(Point p)
Turns to face the direction needed to walk towards *p*

int getPopulation()
Returns the number of creatures of the same type as this in the maze. You'll need to be very careful about synchronization with this method and the other helper methods that you'll need in order to implement it. Also, wrap your entire run method with a try { } finally { } block so that you can detect when a creature ceases to operate even if it stops because an exception is thrown.

void update()
Override moveForward, moveBackward, turnLeft, and turnRight to call this update method (they should still perform their original function as well!) This gives you a place to add code that executes every time your creature's state changes. Also call update as soon as your run method begins.

Observation recall(Point p)
Return the most recent observation of point *p* by any instance of your creature class. If this point has never been observed, return null. To implement this you will have to do something like the following:

- Introduce a shared map data structure (below)
- Extend your toString method to display the shared understanding of the current map
- Override your look method to update the shared map; note that when you look and see a point *x*, you have implicitly observed that everything between yourself and *x* is empty!
- Extend your update method to record each creature's own position in the shared map (since if you're on a square, you know that nothing else is!), and that when you leave a square to mark it as empty (not null!)
- Extend your run method so that when your creature is converted (i.e., stops executing) it marks that square as now belonging to an enemy of unknown class.
- Be *really* careful about thread synchronization here

Helper classes: (you must figure out the synchronization scheme):

```
/** 2D array useful for making maps of the game board */
protected static class Array2D<T> {
    Array2D(Dimension d);
    public T get(Point p);
    public T get(int x, int y);
    public void set(int x, int y, T v);
    public void set(Point p, T v);
    public int getWidth();
    public int getHeight();
    public boolean inBounds(Point p);
    public boolean inBounds(int x, int y);

    /** Generates a picture of the contents, using toString(T) to generate the value for an
      individual square */
    public String toString() ;

    /** Override this to change how the map is rendered. */
    protected String toString(T t);
}
```

```
/** Array of what has been seen */
protected static class ObservationArray2D extends Array2D<Observation> {
    private String myName;

    ObservationArray2D(Dimension d, String myClassName);

    /** Record the observation at the location where it applies only.*/
    void set(Observation obs);

    /** Uses the map symbols to display the observation. E.g., EMPTY = " ", WALL = "X", etc.
      shows my class as "m" and other creatures as "c" unless they are Apples or Flytraps.*/
    @Override
    protected String toString(Observation obs);
}
```

2 Tournament Notes

Full details about the tournament will be in next week's lab handout. Generally speaking, you'll be allowed to submit up to two creatures (or to opt out of the contest). Those creatures will independently compete in multiple one-on-one trials against other creatures in an initial round. The creatures with the best win record will progress to the second round (although only one creature per programmer will be allowed to advance). In the second round, four creatures will compete against each other simultaneously for a few trials. The winner will be the creature with the best win record in the second round.

The wall positions (but not the creature start positions) for maps for the tournament will be revealed before the submission deadline. Those maps may contain any Darwin features except for treasures.

3 Evaluation

Correctness (excluding threads)	**25**
Thread Safety	**25**
Readability	**25**
Efficiency (including synchronization)	**25**
Total	**100**
New custom 3D icon	+2 extra credit

4 Advice

My implementation is about 320 lines of code. Yours should likely be between 250 and 500 as well.

Implement a few of the easy helper methods as a warmup, and then jump to recall and look. Spend some time there designing your approach before trying to implement it. You'll need the toString method to debug these.

Thread synchronization is not something that you can implement by just trying changes to your code. You really have to think through and make a careful argument for why there are no race conditions or deadlocks in your code. That's because improperly threaded code might run fine 99.9% of the time, yet still be wrong. Testing is still important, but be aware that your testing may never actually hit the broken cases. Use abstraction to reduce the problem by synchronizing helper methods or classes, and then let the methods that use them depend on that synchronization to keep state consistent.

The following is my code for iterating over all squares between the creature and a point it observes (I removed the synchronization code, however!) This is not the only way of implementing the iteration. For example, you could take advantage of Direction.forward to make a slightly less efficient but much more compact implementation.

```
 static private int sgn(int x) {
       if (x > 0) {
          return 1;
       } else if (x == 0) {
          return 0;
       } else {
          return -1;
       }
   }


….
{
       Point me = getPosition();
        Point it = obs.position;

        int dist = distance(it) - 2;

        // Only one of these is non-zero
        int dx = sgn(it.x - me.x);
        int dy = sgn(it.y - me.y);

       int x = me.x + dx;
        int y = me.y + dy;

        int t = getTime();

        // Lots of empty squares
        for (int i = 0; i < dist; ++i) {
             observationMap.set(new Observation(x, y, t));
             x += dx; y += dy;
       }

        // And one non-empty
        observationMap.set(obs);
        …
   }
```