

CS 251L Homework 3

Modularity: Wedge Dude

Assigned: September 26, 2011

Due: October 10, 2011 @ 12:00pm (submitted before class starts)

Submission Method: Email solution to wokoun@unm.edu by due date.

Please identify your submission in the email subject header as being homework 3. You only need to attach the source files for your **GhostBrain** implementations (not the compiled classes) to your email. You may attach multiple source files separately or archive them using a mainstream archiver (.zip, .sit, .tar.gz, etc.). If you are sending loose files, please create a proper email attachment rather than copy/pasting one after the other. If you send an unreadable file or archive, the instructor will generally notify you that you need to resubmit it, but the homework is not considered turned in until the instructor gets a readable file.

To get started, download **wedge2.jar** from the class website. This Java ARchive contains most of the software for the application. Create four modular plug-ins to direct four ghosts to chase Wedge Dude. The ghost modules will each implement the interface **GhostBrain** (contained in the JAR), which defines these methods:

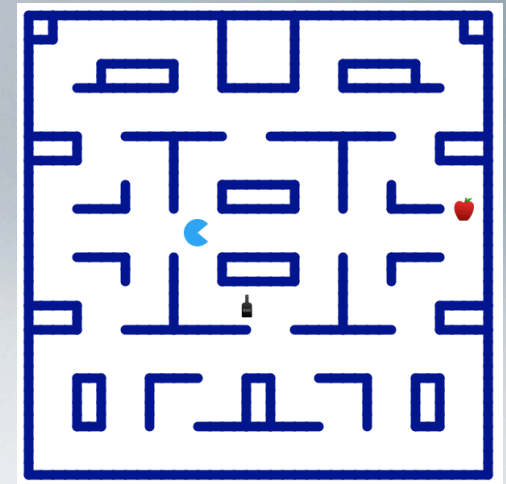
```
public Color getColor();
```

```
public Move nextMove(int currentX, int currentY, Direction direction, Space[][] world, Move[][] toWedge, int wedgeX, int wedgeY);
```

The first method should return a value of your choice from the following set:

GhostBrain.Color.BLACK, **GhostBrain.Color.RED**, **GhostBrain.Color.BLUE**,
GhostBrain.Color.GREEN, **GhostBrain.Color.ORANGE**,
GhostBrain.Color.PURPLE, and **GhostBrain.Color.GOLD**.

The second method will calculate the direction each ghost should move in order to reach Wedge Dude. Arguments passed to the function will give the ghost's current X, Y position, current facing direction, the world map, a "move map" to Wedge Dude (explained on the next page), and Wedge Dude's current X, Y position. Unlike Wedge Dude, the ghosts have a restriction that they may not "reverse course." They may only turn left or right or travel straight ahead.



CS 251L Homework 3

To make things run faster, the application will perform the recursive calculations to find the path to Wedge Dude for the ghosts. The ghost's **nextMove()** function will receive a 2-D array containing the best moves from each grid square in the maze. You can access the appropriate move as **toWedge[currentY][currentX]**.

The “move map” is generated using an alternate algorithm from the one employed for homework 2. Instead of working from the current position to the target, recursively branching out along each possible path, it starts from the target, spreading out through the entire maze (marking all squares with distance 1 from the target, then spreading out from those squares and marking neighboring squares with distance 2, etc.). This eliminates having to evaluate each grid square more than once, but requires performing breadth-first recursion instead of the depth-first recursion used earlier. Doing a **breadth-first** evaluation requires the use of data structures which have not yet been considered in the course.

bad	bad	S1	W2	W3
E2	E1	0	bad	N4
N3	bad	bad	bad	N5
N4	bad	S8	bad	N6
N5	W6	W7	W8	N7

All four ghosts will start from the same location, and only differences in the implementation of the **nextMove()** implementation will cause the ghosts to split up and attack Wedge Dude from multiple sides (if the implementations were identical, all four ghosts will make the same moves and stay piled up in the same location).

Since **nextMove()** will be receiving the “ideal” moves from the application, you will need to add some additional logic to give each ghost its “personality.” Some things you can do include:

- Adding a random element to each ghost’s motion
- Assigning a maze quadrant to each ghost, and having the ghost prefer to stay there
- Having the ghost follow the “ideal” path to the Wedge only when it is closer than a threshold (“homing mode”)
- Alternative heuristics of your choice

You can get random numbers (in the range from 0.0 to 1.0) from the application by calling **WedgeFrame.random()**

CS 251L Homework 3

To add randomness to the ghost's motion, you would:

- Examine the distances of the ideal **Moves** from the four grid squares adjacent to the current location
- Add a random penalty distance to each **Move**
- Compare the four **Moves** and take the adjusted-best one, using the same process as in homework 2.

The larger the penalty distance, the greater the random element in the ghost's motion. Since the **random()** function gives you values between **0.0** and **1.0**, you should "scale" them first by multiplying them by a factor of your choice.

To make a ghost gravitate to a quadrant, you would:

- Examine the distances of the ideal **Moves** from the four grid squares adjacent to the current location
- Add a penalty distance to the **Moves** that take the ghost away from its preferred territory. If the ghost wants to stay in the northeast, penalize the moves to the south and west. The penalty can be a fixed number, or can increase the further the ghost gets from its preferred zone.
- Compare the four **Moves** and take the adjusted-best one, using the same process as in homework 2.

The larger the penalty distance, the greater the ghost's tendency to stay in that corner of the maze. You can add logic to apply this penalty only when the ghost is outside its quadrant. Try combinations of approaches by applying both penalties.

To avoid having a ghost break off pursuit of Wedge Dude when it's close, you may want to turn off penalty calculations when the ghost reaches a certain distance from Wedge Dude. You can get the current distance by looking up the **Move** corresponding to the ghost's current location. Remember that a ghost cannot turn around, so you may not be able to follow the ideal **Move** to get to Wedge Dude.

CS 251L Homework 3

Compile and Run

When you have written your implementation classes, compile them with the following command (make sure `wedge2.jar` is in the same directory with your code):

```
javac -cp wedge2.jar *.java
```

This will allow the compiler to see and link with the other classes that are part of the application. Then, to run the program:

```
java -cp wedge2.jar;. WedgeFrame Pip Marta Blomo Tark
```

The `-cp` option is an abbreviation for `-classpath`. If you are running a Unix-based operating system, replace the semicolon with a colon. The last four terms are the names of your ghost classes (substitute any names you like). The program will load them in on demand when it starts. You can change the names to anything you want.