Programming Project: Game of Life

Collaboration Solo: All work must be your own with optional help from UofA section leaders

The Game of Life was invented by John Conway to simulate the birth and death of cells in a society. The following rules govern the birth and/or death of cells between two consecutive time periods. At time T

- A cell is born if there was none at time T-1 and exactly three of its neighbors were alive.
- An existing cell remains alive if at time T-1 there were either two or three neighbors.
- A cell dies from isolation if at time T-1 there were fewer than two neighbors.
- A cell dies from overcrowding if at time T-1 there were more than three neighbors.

A neighborhood consists of the eight elements around any element (N represents 1 neighbor):

```
N N N
N  N
N N N
```

The neighborhood can extend to the other side of the society. For example, a location in the first row has a neighborhood that includes three locations in the last row. The following patterns would occur when T ranges from 1 to 5, with the initial society shown at T=1. 0 represents a live cell; a blank indicates that no cell exists at the particular location in the society.

```
T=0     T=1     T=2     T=3     T=4  Society dies off at T=4

........  ........  ........  ........  ........  ........  
..0..0..  ..0..0..  ..0..0..  ..0..0..  ........  ........  
..000..  ..0..0..  ..000..  ..0..0..  ........  ........  
........  ........  ........  ........  ........  ........  
```

Other societies may stabilize like this:

```
T=0     T=1     T=2     T=3     T=4  This pattern repeats

........  ........  ........  ........  ........  ........  
..0...0..  ..0...0..  ..0...0..  ..0...0..  ........  ........  
..000..  ..0...0..  ..000..  ..0...0..  ........  ........  
........  ........  ........  ........  ........  ........  
```

Implement class GameOfLife along with unit test class GameOfLifeTest. The design is given because Rick thought this was a good way to develop a final solution. Also, by following the design, you can use the provided graphical user interface to provide another view of how societies grow. Finally, these methods must exist with the same names, parameters, and return types so we can run our reference tests against your code. This avoids compiletime errors on WebCat that results in 0.

Do not just test with the GUI. Write careful assertions first, especially for wraparound and corners. For each of the eight required methods (except toString). Please follow this recommended approach for developing well-tested software:

1. Add a test method first with calls to non-existent methods
2. Make the test compile (add the method heading and return some bogus return value if necessary)
3. See the test fail (get the red bar)
4. Write the code necessary to make your tests pass
5. Refactor your code: make small changes to elucidate code without changing its meaning

Here is the beginning of a unit test intended to help explain expected behavior from some methods such as growCellAt, cellAt, and a simple neighborCount (no wraparound).
import static org.junit.Assert.*;
import org.junit.Test;

public class GameOfLifeTest {

    @Test
    public void testConstructorAndGetters() {
        GameOfLife society = new GameOfLife(5, 8);
        assertEquals(5, society.numberOfRows());
        assertEquals(8, society.numberOfColumns());
        for (int r = 0; r < society.numberOfRows(); r++)
            for (int c = 0; c < society.numberOfColumns(); c++)
                assertFalse(society.cellAt(r, c));
    }

    @Test
    public void testGrowCellAtAndCellAt() {
        GameOfLife society = new GameOfLife(4, 4);
        society.growCellAt(1, 1);
        society.growCellAt(2, 2);
        society.growCellAt(3, 3);

        assertTrue(society.cellAt(1, 1));
        assertTrue(society.cellAt(2, 2));
        assertTrue(society.cellAt(3, 3));
    }

    @Test
    public void testNeighborsWrapping() {
        GameOfLife society = new GameOfLife(10, 16);
        society.growCellAt(3, 3);
        society.growCellAt(3, 4);
        society.growCellAt(3, 5);

        assertEquals(0, society.neighborCount(2, 1));
        assertEquals(1, society.neighborCount(2, 2));
        assertEquals(2, society.neighborCount(2, 3));
        assertEquals(3, society.neighborCount(2, 4));
        // ... many more assertions expected
    }
    // ... Add many more @Test methods here
}

Getting Started:
Start a new Eclipse project with http://www.cs.arizona.edu/~mercer/Projects/GameOfLife.java that has the method stubs shown below (it should compile). Then write tests for one method at a time, preferably in the order shown. You should have a large number of @Test methods (more than 10) for neighborCount and update.

/**
 * A model for John Conway's Game of Life.
 * The design (methods, parameters, return types) by Rick Mercer
 */
public class GameOfLife {

    /**
     * Write the constructor so it takes two integer arguments to represent
     * the number of rows and columns in the game of life. The constructor
     * creates a society with no cells but space to store rows*cols cells.
     * @param rows The height of the grid that shows the cells.
     * @param cols The width of the grid that shows the cells.
     */
    public GameOfLife(int rows, int cols) {
    }
/**
 * Return the number of rows, which can be indexed from 0..numberOfRows()-1.
 * @return The height of the society.
 */
public int numberOfRows() {
    return 0;
}

/**
 * The number of columns, which can be indexed from 0..numberOfColumns()-1.
 * @return The height of the society.
 */
public int numberOfColumns() {
    return 0;
}

/**
 * Place a new cell in the society.
 * Precondition: row and col are in range.
 * @param row The row to grow the cell.
 * @param col The column to grow the cell.
 */
public void growCellAt(int row, int col) {
}

/**
 * Return true if there is a cell at the given row and column.
 * Return false if there is none at the specified location.
 * @param row The row to check.
 * @param col The column to check.
 * @return True if there is a cell at the given row or false if there is not
 */
public boolean cellAt(int row, int col) {
    return false;
}

/**
 * Return one big string of cells to represent the current state of the
 * society of cells (see output below where '.' represents an empty space
 * and 'O' is a live cell. There is no need to test toString. Simply use it
 * to visually inspect if needed. Here is one sample output from toString:
 * GameOfLife society = new GameOfLife(4, 14); society.growCellAt(1, 2);
 * society.growCellAt(2, 3); society.growCellAt(3, 4);
 * System.out.println(society.toString());
 */
// Output
// ............
// ..O.......... 
// ...O.........
// ....O........
/**
 * @return A textual representation of this society of cells.
 */
@override
public String toString() {
    return "Under construction";
}
/**
* Count the neighbors around the given location. Use wraparound. A cell in
* row 0 has neighbors in the last row if a cell is in the same column, or
* the column to the left or right. In this example, cell 0,5 has two
* neighbors in the last row, cell 2,8 has four neighbors, cell 2,0 has four
* neighbors, cell 1,0 has three neighbors. The cell at 3,8 has 3 neighbors.
* The potential location for a cell at 4,8 would have three neighbors.
*/
// .....O..O
// O........
// O.......O
// O.......O
// ....O.O..
/**
* The return values should always be in the range of 0 through 8.
* @return The number of neighbors around any cell using wrap around.
*/
public int neighborCount(int row, int col) {
    return 0;
}

Optional GUI
When you have 100% on WebCat, download and run this GUI. Do not turn in your project with this file

http://www.cs.arizona.edu/~mercer/Projects/GameOfLifeGUI.java

Grading Criteria (Subject to change)

+100 Web-Cat Code Coverage and Problem Coverage. Your code must compile using the specified names, Rick's tests pass, and you have tested all of your methods with assertions in a test method. These 100 points are derived from Web-Cat. You may submit an unlimited number of times. The final submission will be the one that is graded. Notice that a multiplication feature is employed that means 90% and 90% results in 0.81 * 100 or 81/100 points. This is only 81% rather than 90%.