Project MineSweeper

Collaboration: Solo Complete this Project by yourself or with help from Section Leaders and Rick

You are asked to write the non-graphical user interface aspects of the popular game Minesweeper. This project has the following goals:

- Understand how to apply a Stack to solve a complex problem
- Gain experience using a non-trivial algorithm
- Learn how to make critical design choices based on an overall goal
- Prove your current programming experience can solve fun, complex problems.

Start with an Eclipse Project with Files and Images 😊

To get a faster start and be able to run the GUI, import an archive file as an Exiting Eclipse Project:

1. Download this file to a place you can easily find in the minute or so
   http://www.cs.arizona.edu/~mercer/Projects/MineSweeperStart.zip
2. From Eclipse, select File > Import > General > Existing Projects into Workspace > Next
3. Click the radio button to the left of Select archive file and click the Browse button to the right
4. Browse to MineSweeperStart.zip file you just downloaded
5. Click the Finish button

Your new project should now have three Java classes and a folder named images that holds all images.

1. MineSweeper.java All methods implemented as stubs, including both required constructors. Additionally, there is a private inner class GameSquare that you will find useful. You must use a 2D array of the given GameSquare objects. If you do not use GameSquare, we will not help you.
2. MineSweeperTest.java A very small beginning of a unit test use to demonstrate the behavior of method getAdjacentMines(int, int). You must add many more @Test methods
3. MineSweeperGUI.java A GUI that should work if you get 100% on WebCat
4. images A directory (folder) that stores all images needed by the GUI in a place they will found
5. Use java.util.Stack with methods: void push(E el), E pop(), E peek(), boolean isEmpty()

MineSweeper is a game that comes packaged with virtually every version of Windows, going back to the pre-95 days. The player assumes the role of a guy with a mine detector tasked to find the land mines in the field. The "field" is a two-dimensional board of squares containing either mines, or the number of mines next to the square. The objective of the game is to "clear" every square that doesn't contain a mine using logical reasoning based on the number squares you have cleared. Initially, none of the board is cleared and the contents of every square are not visible to the player. To clear a square, the player clicks on it. If it's a mine, the player has lost (since touching mines is generally not productive.) If the square has a number of mines adjacent to it greater than 0, the number is revealed. If the square has zero mines next to it, the game clears all the nearby squares, since all squares next to a '0' would be safe. Here's where things get tricky: if one of the squares next to the '0' is also a '0' (has no mines next to it.) then everything around it gets cleared as well, and the same goes for any newly found '0's, and so on until the "blank" area is completely cleared.

Sound easy? There's more to this than you may first think. You need to use a form of 'backtracking' in your search. The algorithm given in the Minesweeper's comments explains how to do this. The highlight of this algorithm is the use of a Stack. This algorithm is also shown below and in the click(int, int) method of class MineSweeper.
Develop class Minesweeper

Begin in class MineSweeper by implementing the following constructor and two methods at the beginning of MineSweeper.java.

1. `public MineSweeper(boolean[][] mines)`
2. `private void setAdjacentMines()`
3. `public int getAdjacentMines(int row, int column)`

You are given an @Test method for these methods in MineSweeperTest.java the will get you started on the right track: make sure all GameSquare objects know how many adjacent mines they have (0..8).

```java
/**
 * This class represents the model for a game of MineSweeper. It has a constructor
 * that takes a preset boolean 2D array where true means there is a mine. This
 * first constructor (you'll need 2) is for testing the methods of this class.
 * The second constructor that takes the number of rows, the number of columns, and the
 * number of mines to be set randomly in that sized mine field. Do this last.
 * @author YOUR NAME
 */
public class MineSweeper implements MineSweeperModel {

    /**
     * This private inner class stores the data needed for every GameSquare.
     * This is similar to class Node in that
     */
    private class GameSquare {
        private boolean isMine;
        private int row;
        private int col;
        private boolean isVisible;
        private boolean isFlagged;
        private int mineNeighbors;

        // Construct a GameSquare object with all values initialized except
        // mineNeighbors, which is an instance variables that can only be set after
        // all GameSquare objects have been constructed in the 2D array.
        public GameSquare(boolean isMine, int row, int col) {
            this.isMine = isMine;
            this.row = row;
            this.col = col;
            isVisible = false; // Default until someone starts clicking
            isFlagged = false; // Default until someone starts clicking
            // call setAdjacentMines() from both constructors
            // to set this for each new GameSquare.
            mineNeighbors = 0;
        }
    }

    // The instance variable represents all GameSquare objects where each knows its row,
    // column, number of mines around it, if it is a mine, flagged, or visible
    private GameSquare[][] board;
```
/**
 * Construct a MineSweeper object using a given mine field represented by an
 * array of boolean values: true means there is mine, false means there is not
 * a mine at that location.
 * @param mines
 * A 2D array to represent a mine field so all methods can be tested
 * with no random placements.
 */
public MineSweeper(boolean[][] mines) {
    // TODO: Complete this constructor first so you can test preset mine fields
    // (later on you will need to write another constructor for random boards).
    // new GameSquare objects store all info about one square on the board such
    // as its row, column, if it's flagged, visible, or is a mine.
    board = new GameSquare[mines.length][mines[0].length];

    // Example construction of one GameSquare stored in row 2, column 4:
    // /// board[2][4] = new GameSquare(mines[2][4], 2, 4);
    // Use a nested for loop to change all board array elements
    // from null to a new GameSquare

    // You will need to call private void setAdjacentMines() to set
    // mineNeighbors for all GameSquare objects because each GameSquare object
    // must first know if it is a mine or not. Set mineNeighbors for each.
    setAdjacentMines();
}

/**
 * Use the almost initialized 2D array of GameSquare objects to set the
 * instance variable mineNeighbors for every 2D array element (even if that
 * one GameSquare has a mine). This is similar to GameOfLife neighborCount.
 */
private void setAdjacentMines() {
    // Example to set the instance variable mineNeighbors of the one GameSquare
    // object stored in row 2, column 4 to 8:
    // /// board[2][4].mineNeighbors = 8;
    // Use a nested for loop to set mineNeighbors for ALL GameSquare objects
}

/**
 * This method returns the number of mines surrounding the requested
 * GameSquare (the mineNeighbors value of the square). A square with a mine
 * may return the number of surrounding mines, even though it will never
 * display that information.
 * @param row
 * - An int value representing the row in board.
 * @param column
 * - An int value representing the column in board.
 * @return The number of mines surrounding to this GameSquare (mineNeighbors)
 * @return Must run O(1)
 */
public int getAdjacentMines(int row, int column) {
    // TODO: Implement this method
    return -1;
}

// See MineSweeper.java for other methods and the 2nd constructor

// Several documented methods have been removed to save space

// End class MineSweeper
Unit test MineSweeperTest

Begin with the following unit test that just checks to see if every GameBoard object has the number of adjacent mines set. This start of a unit test is also included in the MineSweeperStart project.

```java
/**
 * The beginning of a unit test for MineSweeper. You will need to add many more!
 */
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class MineSweeperTest {
    @Test
    public void testGetAdjacentMinesWithAGivenTwodArrayOfBooleans() {
        boolean[][] b1 =
            { { false, false, false, false, false },
                { false, false, true , true, false },
                { false, false, true , false, false } }; // Use the non-random constructor when testing to avoid random mine placement.
        MineSweeper ms = new MineSweeper(b1);

        // Check adjacent mines around every possible GameSquare
        // First row
        assertEquals(0, ms.getAdjacentMines(0, 0));
        assertEquals(1, ms.getAdjacentMines(0, 1));
        assertEquals(2, ms.getAdjacentMines(0, 2));
        assertEquals(2, ms.getAdjacentMines(0, 3));
        assertEquals(1, ms.getAdjacentMines(0, 4));
        // Second row
        assertEquals(0, ms.getAdjacentMines(1, 0));
        assertEquals(1, ms.getAdjacentMines(1, 1));
        // Should work even if GameSquare has a mine like the next 2
        assertEquals(2, ms.getAdjacentMines(1, 2));
        assertEquals(2, ms.getAdjacentMines(1, 3));
        assertEquals(2, ms.getAdjacentMines(1, 4));
        // Third row
        assertEquals(0, ms.getAdjacentMines(2, 0));
        assertEquals(1, ms.getAdjacentMines(2, 1));
        assertEquals(3, ms.getAdjacentMines(2, 2));
        assertEquals(2, ms.getAdjacentMines(2, 3));
        assertEquals(2, ms.getAdjacentMines(2, 4));
    }
}
```

- Write and test these simple getter methods, some of which simply return that status of a GameSquare object's instance variable in the 2D array of GameSquare objects named board.
  ```java
  public int getTotalMineCount();
  public boolean isFlagged(int row, int column);
  public void toggleFlagged(int row, int column);
  public boolean isMine(int row, int column);
  public boolean isVisible(int row, int column);
  ```

- Tackle the click algorithm, which is the most challenging.

- When everything is working, implement the 2nd constructors that randomly sets mines with a construction like this: MineSweeper game = new MineSweeper(4, 6, 12)
Algorithm for `click(int, int)`

MineSweeper should automatically make visible all possible `GameSquare` objects adjacent to a blank `GameSquare` object in the board. This includes all `GameSquare` objects with no adjacent mines up to any `GameSquare` with a number from 1..8.

User clicks (4, 2) to get a 2. Just mark that as visible.

Then user selects (6, 3) with no adjacent mines which makes 36 `GameSquare` objects visible.

Required Algorithm for `public void click(int row, int col)`

Here is the algorithm for `public void click(int row, int column)`. The `click` method is called each time a player clicks any `GameSquare`. The GUI send a click to the MineSweeper object for the clicked row and column. There are five possibilities for updating the board during the click messages to your MineSweeper. The `GameSquare` object stored at the just clicked row and column

1. is a mine (player will be notified on loss when the GUI refreshes)
2. is visible already (do nothing)
3. is flagged (do nothing)
4. has mineNeighbors > -1 (simply mark that visible)
5. is not adjacent to any mines with mineNeighbors == 0 (clear all possible `GameSquare`s)

Because Minesweeper automatically clears all squares adjacent to any blank square connected to the square clicked, a special algorithm is needed to set the proper part of the board visible. This pseudocode shows the suggested algorithm.

```java
public void click(int row, int col)  // aka the clicked GameSquare
   // Check special cases first, there may be little or nothing to do
if the clicked GameSquare is a mine
   change the state of this object to lost (when this click method finishes, the GUI will refresh with 😞 )
else if the GameSquare is flagged
   return // nothing to do
else if the GameSquare is already visible
   return // nothing to do
else if the clicked GameSquare has 1 or more neighboring mines
   set the square to be visible  // Clear only the clicked GameSquare when it is numbered 1..8
   return
else { // see next page for the stack and loop part
```
// Clear all possible GameSquare objects up to the borders or until the GameSquare has 1 or more mines
mark square as visible
push the square onto the stack See stack below for the current state: Gamesquare.column == 5
while the stack is not empty:
    pop the stack and mark square as the current square
    if the current square has no neighboring mines (not 1..8)
        for each adjacent square
            if it's not visible and not flagged
                push adjacent square on stack and set its isVisible to true
} // end else

Click (0, 5)

This Stack stores GameSquare objects with these instance variables
(plus 2 others not shown: isFlagged and isMine)

Grading Criteria (100 points, subject to change)

Problem Coverage (No code coverage this time!) 100pts
+100 Web-Cat correctness. To get 100% for these 100 points, you will need 100% problem coverage only,
which means Rick's tests pass your code. There is no code coverage this time because Webcat can not click on
the game board, set flags,… However, you can get a score of 0 even though all of your tests passed in your
workspace because

- WebCat reports a compile time error (look for Unknown symbol).
- One of Rick's test cases placed your loop into an infinite loop (Timeout error)
- One of your assertions failed on WebCat (even if you think it passed for you locally)
- Please note other ways to lose points when graded by your Section Leader: