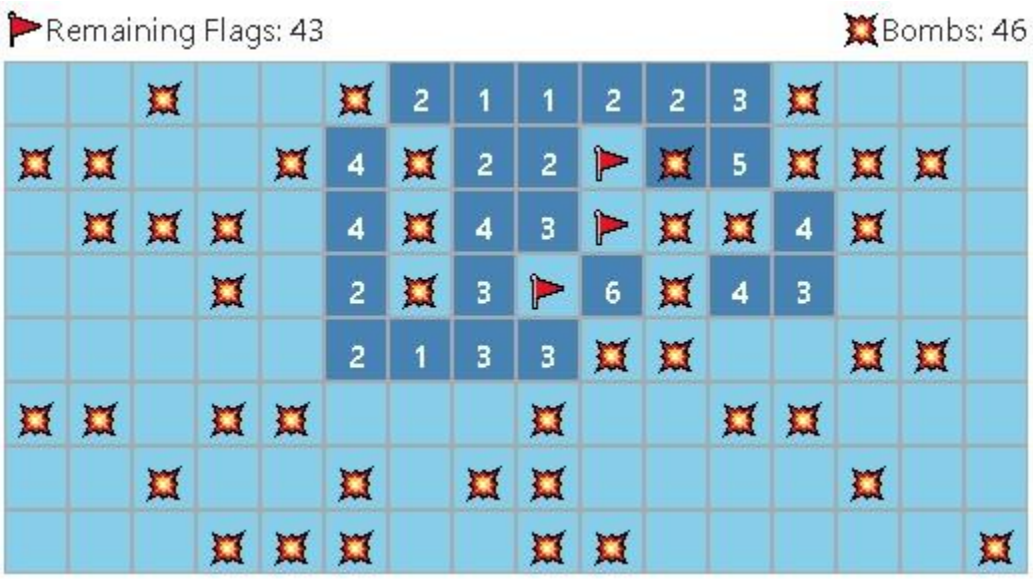
Description

This project builds a complete working game called *Bomb Squad*. This game is similar to the old Microsoft Windows *Minesweeper* game.

A gameboard containing a number of cells has hidden bombs in certain "armed" cells. All cells are initially displayed as blanks except for one hint cell. When a player clicks an unarmed cell, that cell and several surrounding cells will display numbers (hints) indicating how many bombs are adjacent to those cells. A player can right-click a cell to mark it as possibly containing a bomb. Right-clicking will place a flag icon on the cell. If a player left-clicks an armed cell all unmarked armed cells explode and the game is over. The score is the percentage of correctly marked bombs.



Instructions

Since this project uses the model-view-controller pattern in a web application much of the code is already provided for you. Most of the provided code is in the HTML views. The controller class is also provided for you. **You should not make any changes to the Game.cshtml file or the HomeController.cs file.**

1. Download the **BOMBSQUAD.ZIP** file from MyBLC.
2. Extract the ZIP file to a location on your computer where you will be able to find it.
3. Open the **BOMBSQUAD.sln** Visual Studio solution file in Microsoft Visual Studio.

Instructions continue on the next page…

4. Locate the **Models folder** in the Solution Explorer window.

5. Add a class named **BombSquad** to the Models folder.
   Make sure the BombSquad class is in the *BOMBSQUAD.Models* namespace and make sure it references the *System* and *System.Collections.Generic* namespaces as shown in the following code snippet.

```
using System;
using System.Collections.Generic;

namespace BOMBSQUAD.Models
{
    public class BombSquad
    {

    }
}
```

6. Add another class named **Cell** to the Models folder.
   Make sure the Cell class is in the same namespace as BombSquad and that it also references the *System* and *System.Collections.Generic* namespaces.

```
using System;
using System.Collections.Generic;

namespace BOMBSQUAD.Models
{
    public class Cell
    {

    }
}
```

7. Use the class diagram on the following page to add the properties and methods to the **BombSquad** and **Cell** classes.

   Notes:

   - Properties named in all UPPER_CASE are constants, and their constant values are shown on the class diagram. Constants are declared in C# like any other field except they include the **const** modifier.

   - Access modifiers used are: **+** for public, **-** for private, **~** for internal, **~/-** for internal get/private set, **+/-** for public get/private set, and **+/~** for public get/internal set.

   - The HomeController class is shown on the diagram, but you do not need to build that.

The following describes the requirements for each property and method of these classes.

8. **BombSquad constants**: The class diagram shows all the information needed to add the constants to the BombSquad class. Most of the constants are self-explanatory by their names, but possibly not these two:
    a. HINT_RADIUS: When a cell is clicked this is the distance from that cell that hints are shown.
    b. BOMB_DENSITY: The average density of bombs per game board. (2 = 1/2 cells, 3 = 1/3 cells, etc.)

9. **BombSquad.ArmedCells**: The **ArmedCells** property is a simple integer value. No special code is needed for the getter or setter. This property will contain the number of armed cells on the gameboard. Its value is set in the RefreshCells method.

10. **BombSquad.GameOver**: The **GameOver** property is a simple Boolean value. No special code is needed for the getter or setter. This property will be set to **`true`** when the game is over. Its value is set in the Reset and FlagCell methods.

11. **BombSquad.Cells**: The **Cells** property is a **Dictionary** containing all the cells on the gameboard. Each element's key is a string representing the Cell's Address, and each element's value is the Cell object itself. The Cells property is populated in the RefreshCells method.

12. **BombSquad.ClickedCell**: The **ClickedCell** property is a Cell object. No special code is needed for the getter or setter. When a player clicks a cell on the screen, this property will contain a reference to that cell. Its value is set in the ClickCell method.

13. **BombSquad.UnflaggedBombCount**: The **UnflaggedBombCount** property is an integer value. There is no setter for this property. The getter must return the number of unflagged bombs on the gameboard. To retrieve this value, you can read through each Cell in the Cells Dictionary (#11 above) to get the total number of cells which are Armed but not Flagged. (The Cell class has Armed and Flagged properties for this purpose.)

14. **BombSquad.FlagCount**: The **FlagCount** property is an integer value. There is no setter for this property. The getter must return the number of flags that have been placed on the gameboard. You can retrieve this value the same way you retrieved the UnflaggedBombCount value above, except this time we only want to check whether a call has been Flagged.

15. **BombSquad.Score**: The Score property's data type is **`double`**. It does not have a setter. Its getter must return the number of correctly placed flags as a **percentage** of ArmedCells.

16. **BombSquad. RefreshCells():** The **RefreshCells()** method does the following:
    a. Sets ArmedCells value to zero.
    b. Initializes the Cells Dictionary:
    ```
    Cells = new Dictionary<string, Cell> { };
    ```
    c. Uses the VERTICAL_CELLS and HORIZONTAL_CELLS constants in a nested for-loop to instantiate a Cell object for each position on the gameboard and add that object to the Cells Dictionary. The gameboard has VERTICAL_CELLS rows and HORIZONTAL_CELLS columns.
    d. As each cell is added to the Cells Dictionary within the for-loop, check its Armed property. If a cell is Armed then increment the ArmedCells property value by 1.

17. **BombSquad.SetInitialHint()**: The **SetInitialHint()** method randomly selects one of the cells from the Cells Dictionary. If that cell is armed, then it must continue looking for an unarmed cell until it finds one. Once it finds an unarmed cell, set that cell's **`Clicked`** property to **`true`**.
    a. Remember you can use .NET's **Random** class and its **Next()** method to generate random numbers: `Random rand = new Random();`

18. **BombSquad.ClickCell()**: The **ClickCell(string address)** method takes a single string input parameter. If the Cells Dictionary contains a key matching this parameter's value then set `ClickedCell = Cells[address]` and then call the `Click()` method of that cell.

19. **BombSquad.FlagCell()**: The **FlagCell(string address)** method works much like the ClickCell() method (#18 above) except that it calls the Cell's `ToggleFlag()` method instead of its Click() method.

    In addition, the FlagCell method must also check whether the UnflaggedBombCount value is zero, and if it is it set the value of GameOver to `true`. (UnflaggedBombCount will equal 0 when all bombs have been flagged. The game is over at that point, and the player wins!)

Now for the Cell class's properties and methods…

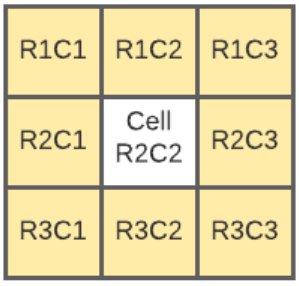
20. These properties can be added to the Cell class by looking at the class diagram on page 3. There is no special code necessary for these properties' getters and setters except for the access modifiers specified on the class diagram.
    a. **game**
    b. **Armed**
    c. **XCoordinate**
    d. **YCoordinate**
    e. **Exploded**
    f. **Flagged**
    g. **Clicked**
    (Note that Clicked means that a cell has had its Click() method called. Once a cell's Clicked property is `true` it will remain `true` until the game is over. The game.ClickedCell property contains a reference to the cell that a player just clicked.)

21. **Cell.Address**: The **Address** property is a string value that represents the Cell's "R1C1" address. There is no setter for this property. Its getter must return the "R1C1" address, which is easy to construct. This is simply the character "R" followed by the YCoordinate value followed by the character "C" followed by the XCoordinate value. For example, "R2C13" is the address of the cell with YCoordinate 2 and XCoordinate 13.

22. **Cell.Content**: The **Content** property is a string value that represents the Cell's HTML content. There is no setter for this property. Use the following instructions for this property's getter.
    a. If Exploded is `true`, then return the string "💥" including the semicolon.
    b. Else if Flagged is `true`, then return the string "🚩" including the semicolon.
    c. Else if Armed is true, then return "" (An empty string.)
    d. Else if Clicked is true, then return the value of the AdjacentBombCount property.
    e. If none of the above is true, then return "" (An empty string.)

23. **Cell.AdjacentCells**: The **AdjacentCells** property is a List of Cell objects directly touching the current Cell. For most cells, the ones not along an edge, there are eight adjacent cells. There is no setter for this property. This property's getter must return a **`List<Cell>`** (list of Cell objects) which contains the cell objects adjacent to the current cell.

    As an example, the yellow highlighted cells in this image are the AdjacentCells of the Cell at Address R2C2.



24. The **AdjacentBombCount** property is an integer value indicating the number of AdjacentCells which have their Armed property equal to **`true`**. This property's getter should simply iterate through the cells in AdjacentCells (#23 above), sum the number of them whose Armed value is true, and return that number.

25. **Cell()**: The **Cell(int row, int col, BombSquad bombsquad)** constructor takes three parameters:
    a. an integer representing the cell's row
    b. an integer representing the cell's column
    c. a BombSquad object which is the BombSquad object this cell belongs to

    Inside the Cell constructor you should set the values of the Cell's readonly properties using these input parameters. (See the class diagram for the list of readonly properties.)

    Inside the constructor you must also use the .NET Random class to randomly determine if this cell should have its Armed property set to **`true`**. It is important that the chance of the cell being armed is based on the value of the BombSquad.BOMB_DENSITY constant. Here is a hint on how to do that: *Generate a random integer between 0 and the number of elements in* **game.Cells**. *Then divide that integer by BombSquad.BOMB_DENSITY. If the remainder is 0 then set Armed to true*.

26. **Cell.ToggleFlag()**: The **ToggleFlag()** method sets Flagged to **`true`** when…
    a. Exploded is false **and**
    b. Flagged is false **and**
    c. Clicked is false **and**
    d. the value of game.FlagCount is less than the value of game. ArmedCells

    Otherwise this method sets Flagged to **`false`**.

27. **Cell.Explode()**: The **Explode()** method does the following:
    a. Sets game.GameOver to **`true`**.
    b. Sets the Exploded property to **`true`** if the Flagged property is not **`true`**.
    (This makes sure that if a player has flagged an armed cell it remains flagged and will be calculated into their score.)
    c. Iterates through all the cells in game.Cells and calls their Exploded() methods.
    Important: Make sure you only call their Explode method if they are Armed, not already Exploded, and not Flagged. Otherwise you may end up in a continuous recursive function that results in a stack overflow exception.

28. **Cell.Click()**: The **Click()** method does several things, but only does anything when the **Flagged** property is **`false`** or the current Cell is equal to game.ClickedCell. So enclose all of the code inside the Click() method in an if-block that checks for one of those two conditions. Remember to use the **`this`** keyword to refer to the current Cell object.

    Now, inside the if-block that you built from the above paragraph…
    a. Set the Clicked property's value to **`true`**.
    b. Set the Flagged property's value to **`false`**.
    c. If the Armed property is **`true`** and the Exploded property is **`false`**, then call the Explode() method.
    d. Iterate through every cell in AdjacentCells and…
        i. if that cell is not Armed **and**
        ii. that cell is not Clicked **and**
        iii. that cell's XCoordinate and YCoordinate values are within BombSquad.HINT_RADIUS from game.ClickedCell's X and Y coordinates

        then call that cell's Click() method.
        (This will cause those cells to display the number of adjacent bombs.)

29. That finishes the project. Run the game and see if it plays as it should. If not, see if you can find where the error is.

    You have the entire class periods of November 30, December 2, and December 4 to work on this project. We will not have lecture those days, but everyone should log onto Zoom for any announcements or to ask questions. We'll only meet for 5 or 10 minutes and then you will be able to work on the project.

    Also, you do not have to wait for the Zoom session to ask questions. If you have tried to figure something out and have been unsuccessful, feel free to email me between classes.

30. When you are completely finished, upload your **BombSquad.cs** and **Cell.cs** files to MyBLC.