ITI 1121. Introduction to Computing II
Winter 2018

Assignment 2
(Last modified on February 7, 2018)

Deadline: February 25th, 2018, 11:30 pm

Learning objectives

- Designing an application utilizing event-driven programming.
- The Model-View-Controller design pattern.
- Using a stack.

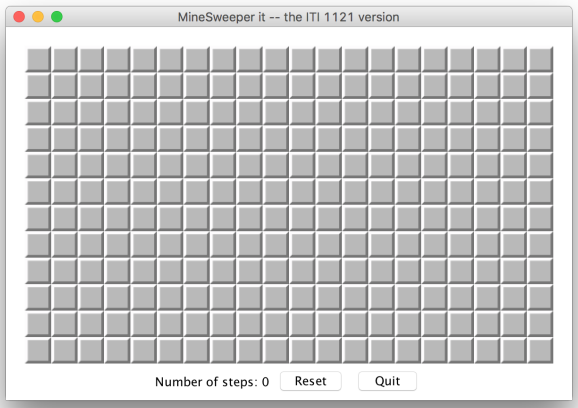

Figure 1: Initial screen of our MineSweeper game.

Background information

We are going to create our own implementation of the game "MineSweeper". Several Web versions are available, for example here.

Here is a description of the game, taken from http://minesweeperonline.com/: in this game, a board is filled with many small squares. Some squares contain mines (bombs), others do not. If you click on a square containing a mine, you lose. If you manage to click all the squares that do not contain a mine, without clicking on any square that contains a mine, you win. Clicking a square which does not have a mine reveals the number of neighbouring squares containing mines. Use this information plus some guess work to avoid the mines. To open a square, point at the square and click on it. A squares "neighbours" are the squares adjacent above, below, left, right, and all 4 diagonals. Squares on the sides of the board or in a corner have fewer neighbours. The board does not wrap around the edges. If you open a square with 0 neighbouring mines, all its neighbours will automatically open. This can cause a large area to automatically open.

In the original game, the first square you open is never a mine, however, to simplify our implementation, our first square could be a mine. The original game has also a feature to mark the square that are thought to be a mine, but we won't be implementing that part here.

Figure 1 shows an example of the initial state of the game. Figure 2 shows one step of the game. The player wins when all the mines are found (Figure 3). The player loses when a square containing a mine is clicked (Figure 4).

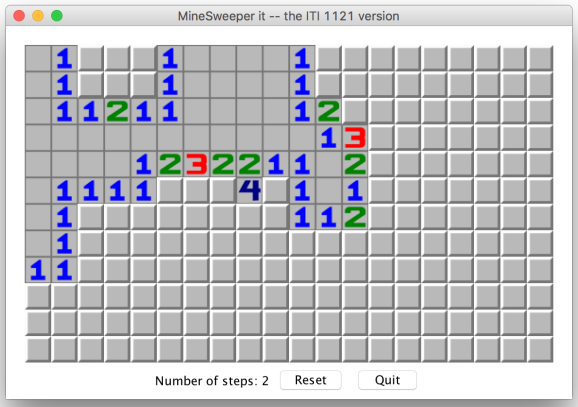

Figure 2: One step of the game.

Figure 3: The player wins when all the squares that are not mines are uncovered. The remaining squares of board are then uncovered and the player is offered a choice of playing again or quitting.

Model-View-Controller

Model-View-Controller (MVC) is a very common design pattern, and you will easily find lots of information about it on-line (e.g. Wikipedia, Apple, Microsoft to name a few). The general idea is to separate the roles of your classes into three categories:

- The **Model**: these are the objects that store the current *state* of your system.
- The **View** (or views): these are the objects that are representing the model to the user (the UI). The representation reflects the current state of the model. You can have several views displayed at the same time; though in our case, we will have just one.
- The **Controller**: these are the objects that provide the logic of the system, how its state evolves over time based on its interaction with the "outside" (typically, interactions with the user).

Figure 4: The player loses when a mine is clicked. The remaining squares of board is then uncovered, with the clicked mine displayed on a red background. The player is offered a choice of playing again or quitting.

One of the great advantages of MVC is the clear separation it provides between different concerns: the model only focuses on capturing the current state, and doesn't worry about how this is displayed nor how it evolves. The view's only job is to provide an accurate representation of the current state of the model, and to provide the means to handle user inputs, and pass these inputs on to the controller if needed. The controller is the "brain" of the application, and doesn't need to worry about state representation or user interface. Thus each part can be designed, programmed, and tested independently.

In addition to the separation, MVC also provides a logical collaboration-schema between the three components (Figure 5). In our case, it works as follows:

1. When something happens on the view (in our case, mostly when the user clicks the square which is covered), the controller is informed (message 1 of Figure 5).
2. The controller processes the information and updates the model accordingly (message 2 of Figure 5).
3. Once the information is processed and the model is updated, the controller informs the view (or views) that it should refresh itself (message 3 of Figure 5).
4. Finally, each view re-reads the model to reflect the current state accurately (message 4 of Figure 5).

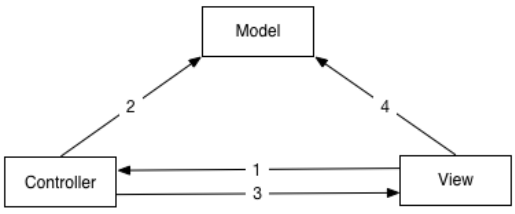

Figure 5: The collaboration between the Model, the View and the Controller.

The model (20 marks)

The first step is to build the model of the game. This will be done via the class **GameModel** and the helper class **DotInfo**. Our unique instance of the class GameModel will have to store the current state of the game. This includes

- The icon (number of neighbouring mines, or blank if none) and status (covered or not) of each dot on the board. The class **DotInfo** helps with this.

- The size of the board.
- The number of steps taken by the player so far
- The number of mines around each dots
- The total number of mines in the Model
- The number of uncovered dots so far

It also provides the necessary setters and getters, so the the controller and the view can check the status of any square, and the controller can change the status of a square. Finally, it provides a way to (re)initialize the game, reallocating the initial position of the mine randomly.

A detailed description of the classes can be found in the documentation. The starting files for the source code are at https://www.eecs.uottawa.ca/ gvj/Courses/ITI1121/assignments/02/A2src.zip

The Controller (40 marks)

Note: as a development strategy, we suggest that you first create a very simple, temporary text-based view that prints the state of the model (via the model's **toString()** method) and asks the user to input the next square to select (by asking for a column and a line). Having this basic view will give you an opportunity to test your model and your controller independently from building the GUI.

The controller is implemented in the class **GameController**. The constructor of the class receives the width and height of the board and the number of mines to hide on the board as parameters. The instance of the model and the view are created in this constructor. The instance of the class GameController is the one in charge of the computing the consequences of the user clicking a square. The controller's **play(width, height)** method is used to compute the consequence of clicking the square at location width X height. In that method, the controller computes the new state of the game, including the possibility of a loss, in which case the user is being informed (see Figure 4). That method also checks if all the mine-free squares have been clicked, and ends the game if that is the case (see Figure 3). At the end of the game, the player can choose to play again or exit the game.

When the game finishes, regardless if won or lost, the entire board should be uncovered. In case of a loss, the mine that has been clicked should be displayed on a red background (see Figure 4).

"Minesweeper" game

When the player clicks on a square that was not clicked before, that square must be uncovered. If it contained a mine, then the player just lost (Figure 4). Otherwise, the number of neighbouring mines should be displayed; a blank icon is used when that number is zero (instead of an icon showing "0"), otherwise an icon with the actual number of neighbouring mines, from 1 to 8, is used.

When the square has zero neighbouring mines, the it means that all covered squares around that square can safely be clicked on without risking to click on a mine. The game does this automatically for the player. So each time a square without neighbouring mines is clicked, all (up to 8) neighbouring squares must in turn be uncovered. If some of these neighbouring squares are themselves without neighbouring mines, then their own neighbour should be cleared, etc. We call this effect **zone clearing**.

Note: as an implementation strategy, we suggest that you do not implement the zone clearing mechanism at first. Without zone clearing, the game is perfectly playable, but the "clearing" must be done manually. Once you have a game that works well without zone clearing, you can complete it by adding this feature.

In this assignment, we are going to rely on a stack to implement zone clearing. A stack interface is specified, but you need to provide a proper implementation for it. For the time being, we will use an array-based implementation.

Here is a sketch of an algorithm that you will use to implement the stack-based zone clearing. The parameter **initialDot** is the newly clicked square. Remember that zone clearing is only initiated when the square initially selected by the player has zero neighbouring mines (in other words, **initialDot** has zero neighbouring mines, otherwise the method is not called).

```
Stack-Based-clearZone(initialDot).
   create an empty stack.
   push the initial dot to the stack
   While the stack is not empty Do
```

```
        remove a dot d from stack
        For all dot n neighboring d
           If n is not covered then
              uncover it
              if  n has no neighbouring mines
                push n onto the Stack
              End if
           End If
        End For
    End While
```

The instance of the class GameController is created by the class **Minesweeper**, which contains the **main**. By default, a width of 20 and a height of 12 is used for the board, and 36 mines are used. Three runtimes parameters can be passed on to the main to change these defaults parameters (exactly three parameters, or none at all). Some verifications are still done when parameters are used: the width should be greater than 10, otherwise the default is used. The height should be greater than 12, otherwise the default is used, There should be at least 1 mine, otherwise the default is used. Finally, if the number of mines specified equals or exceeds the number of squares on the board, then the value used instead is the number of squares minus 1. In this assignment, you can assume that the parameters, if provided, can be parsed as integers.

A detailed description of the classes can be found in the documentation. The starting files for the source code are at https://www.eecs.uottawa.ca/ gvj/Courses/ITI1121/assignments/02/A2src.zip

The view (40 marks)

We finally need to build the UI of the game. This will be also be done based on the class **GameView**, which extends JFrame. It is a window which shows at the bottom the number of steps played so far, and two buttons, one to reset and one to quit the game. Above these the board is displayed. The board is made of a series of squares, h lines and w columns of them (where h is the height of the board, w is the width of the board). Figure 6 shows the full GUI.

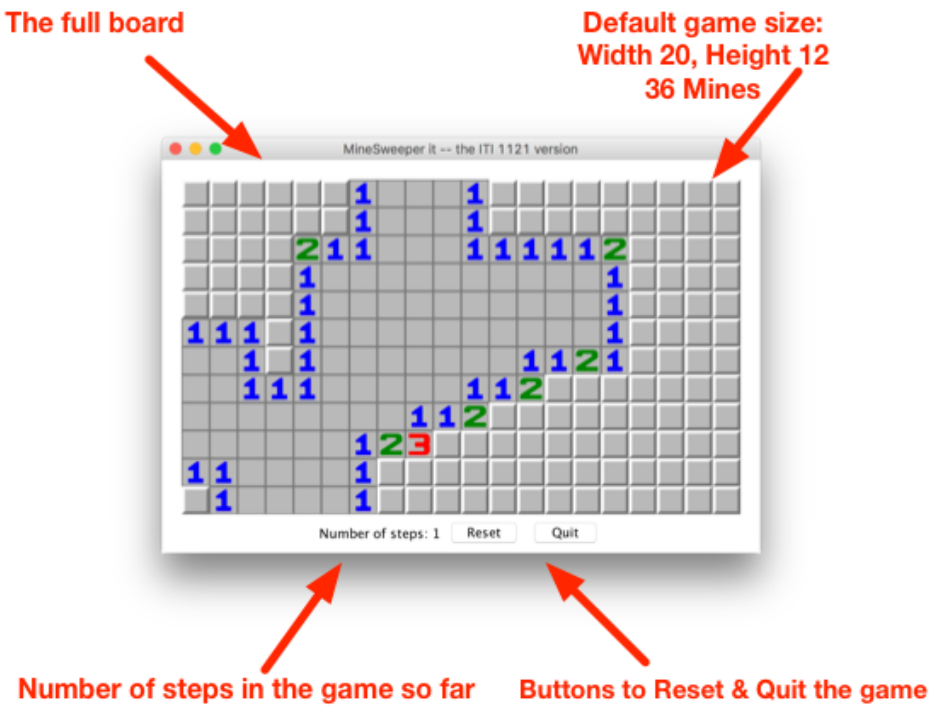

Figure 6: The GUI with a game of width 20 height 12 mine 36.

To implement the square, we will use the class **DotButton** which extends the class JButton. DotButton is based on Puzzler by Apple.You can review the code of the program "Puzzler" seen in lab 5 to help you with the class **DotButton**. The code

for DotButton includes a method **getImageIcon()** which uses the current value of the instance variable **icon** to find the correct **ImageIcon** reference. In order to use **getImageIcon()**, the series of icons must be put in a subdirectory called "icons". The value for the variable "icon" must match one of the predefined constant integers of the class DotButton.java. Note that conveniently, the constants corresponding to the number of neighbours matches the actual number of neighbours (0 for zero neighbour, 1 for one neighbour etc.)

For the dialog window that is displayed at the end of the game, have a look at the class JOptionPane.

Note that although instances of the **GameView** classe have buttons, they are not the listeners for the events generated by these buttons. This is handled by the controller.

A detailed description of the classes can be found in the documentation. The starting files for the source code are at https://www.eecs.uottawa.ca/ gvj/Courses/ITI1121/assignments/02/A2src.zip

Bonus (10 marks)

The actual game has an additional feature which allows to "flag" squares that are believed to hide a mine. Flagging these square helps the player keeping track of the assumed state of the board. Additionally, a flagged square cannot be clicked, so this prevents mistakenly setting of a mine. As a bonus question, you could implement that missing feature. If you do, then you need to find a way for the player to distinguish between "flagging" a square and "clicking" a square. You need to indicate the squares that are flagged (an icon has been included for that purpose). The player should be able to "unflag" a flagged square, to make it clickable as well. Additionally, it would be interesting to show on the UI the number of mines that have yet to be found (that is, to show the difference between the number of mines and the number of flagged squares).

Rules and regulation

Follow all the directives available on the assignment directives web page, and submit your assignment through the on-line submission system BrightSpace.

You must preferably do the assignment in teams of two, but you can also do the assignment individually. Pay attention to the directives and answer all the following questions.

You must use the provided template classes. You **cannot** change the signature of the provided methods.

Files

You must hand in a zip file, and only a zip file, containing the following files, and only the following files:

- A text file README.txt which contains the names of the two partners for the assignments, their student ids, section, and a short description of the assignment (one or two lines).
- The source code of **all** your classes
- a subdirectory "icons" with the icon images in it
- The corresponding JavaDoc doc directory.
- StudentInfo.java, properly completed and properly called from your main.

WARNINGS

- Failing to strictly follow the submission instructions will cause automated test tools to fail on your submission. Consequently, your submission will **not** get marked.
- A tool will be used to detect similarities between submissions. We will run that tool on all submissions, across all the sections of the course (including French sections). Submissions that are flagged by the tool will receive a mark of 0.
- It is your responsibility to ensure that your submission is indeed received by the back-end software, blackboard. If you submission is not there by the deadline, it will obviously **not** get marked.
- Late submission will not be accepted.

Last Modified: February 7, 2018