ITI 1121. Introduction to Computing II
Winter 2016

Assignment 2
(Last modified on February 10, 2016)

Deadline: March 4th, 2016, 11:59 pm

[PDF]

Learning objectives

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

Background information

We are going to create our own implementation of the game "circle the dot". A version for android, on which this description is based, can be found here. A version for iOS can be found here. This game is simple: a blue dot is trying to "escape" the board (that is, exit from the board), while the player attempts the prevent that by selecting gray dots, turning them into orange dots that the blue dot cannot cross. Figure figure1 shows two examples of the initial game configuration screen[1].

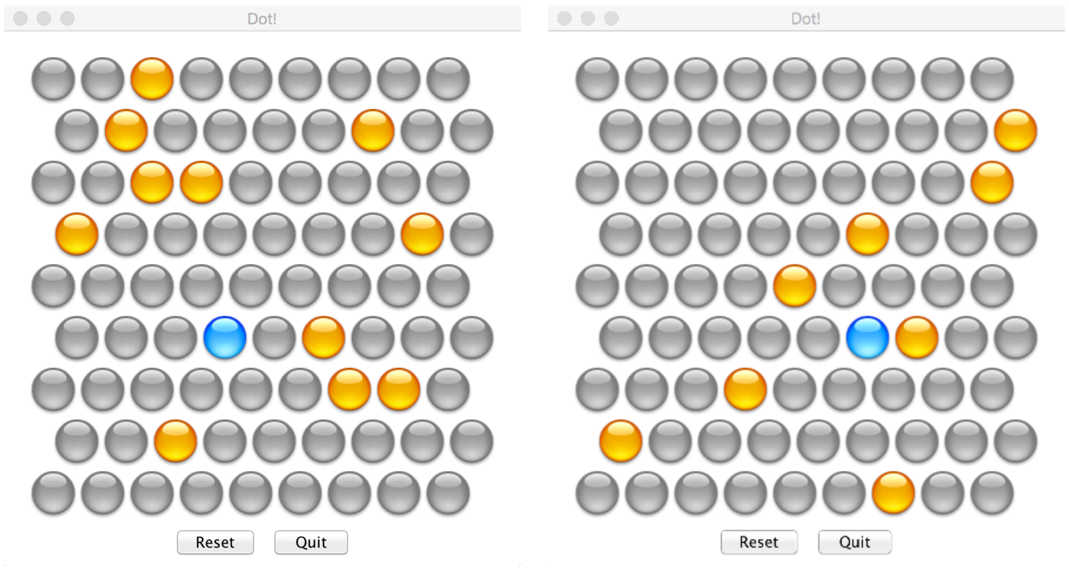


Figure 1: Two examples of the initial configuration of the game.

The player wins when the blue dot is encircled by orange dots (Figure figure2). As can be seen on Figure figure1, initially, some of the dots are already orange (this is a random selection, as we will explain later). The initial location of the blue dot is also randomly selected, but toward the center of the board. If the board has an even number of rows/columns, then the blue dots is randomly located on one of the central four dots, while if the number of rows/columns is odd, it is randomly located on one of the central nine dots (Figure figure3). At each step the blue dot will move to one of the six neighboring dots (Figure figure4).

[1]The UI is based on "Puzzler" by Apple.

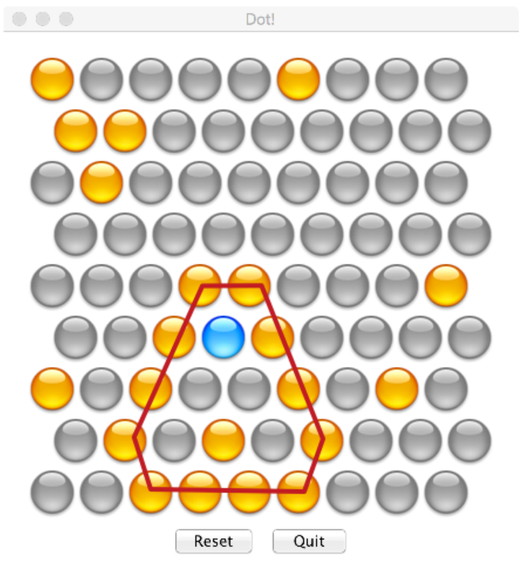


Figure 2: The player won: the blue dot is circled by orange dots.

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 of 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 overtime based on its interaction with the "outside" (typically, interactions with the user).

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.

In addition to the separation, MVC also provides a logical collaboration-schema between the three components (Figure figure5). In our case, it works as follows:

1. When something happens on the view (in our case, when the user selects a gray dot to turn it orange), the controller is informed (message 1 of Figure figure5).
2. The controller processes the information and updates the model accordingly (message 2 of Figure figure5).
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 figure5).
4. Finally, each view re-read the model to reflect the current state accurately (message 4 of Figure figure5).

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 **Point**. Our unique instance of the class GameModel will have to store the current state of the game. This includes

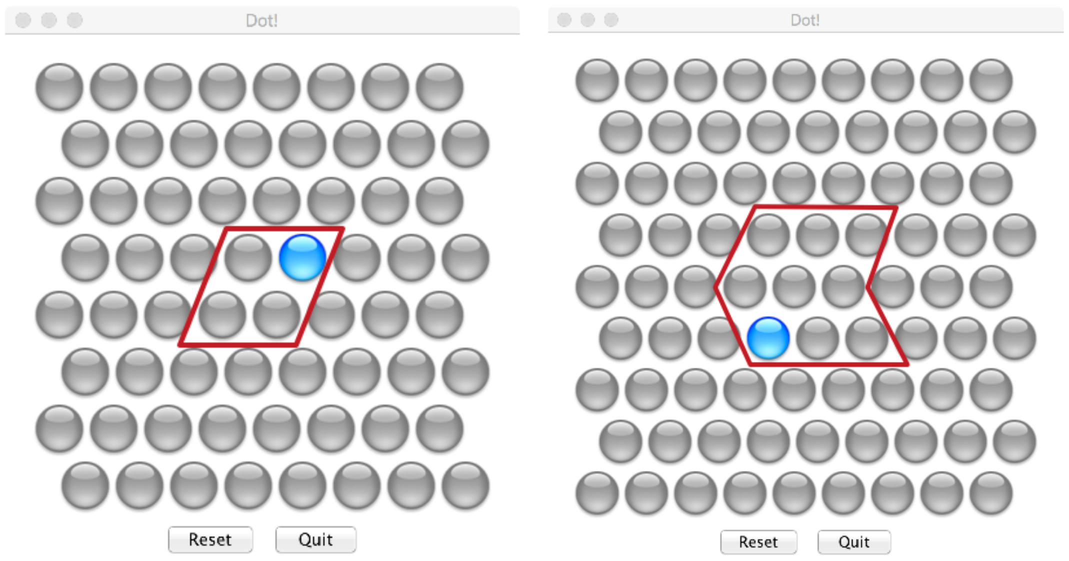


Figure 3: Initial location of the blue dot: on an even board (left), the position is randomly selected among the four central dots. On an odd board (right), it is randomly selected among the nine central locations.

- The definition of three possible status of a dot. Each dot has one of the following status
  - **AVAILABLE**: this dot is not the blue dot, and it hasn't been selected,
  - **SELECTED**: this dot is not the blue dot, but it has been selected,
  - **DOT**: this dot is the blue dot.
- The current location of the blue dot
- The current status of every dot on the board
- The number of steps taken by the player so far
- The size of the board.

It also provides the necessary setters and getters, so the the controller and the view can check the status of any dot, and the controller can change the status of a dot or move the blue dot. Finally, it provides a way to initialize the game, placing the blue dot randomly as explained earlier, and preselecting some of the other dots with a set probability. In our case, we are going to use 10% as the probability that a dot that isn't the blue dot is initially selected.

A detailed description of the classes can be found here:

- GameModel Documentation
- GameModel.java
- Point Documentation
- Point.java

The view (30 marks)

We then need to build the UI of the game. This will be done with two main classes. The first class is **GameView**, which extends JFrame. It is the main window of the application. Mostly, it shows two buttons at the bottom, to reset and to quit the game, and it includes an instance of the second class, **BoardView**.

The class **BoardView**, which extends JPanel, contains the board. The board is made of a series of dots, n lines and n columns of them (where n is the size of the board). To implement the dots, we will use the class **DotButton** which extends the

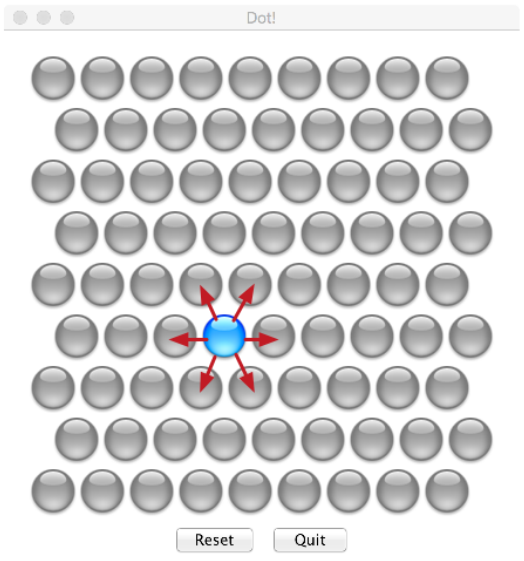


Figure 4: The blue dot moves one dot at a time, in one of the six dots that surrounds it.

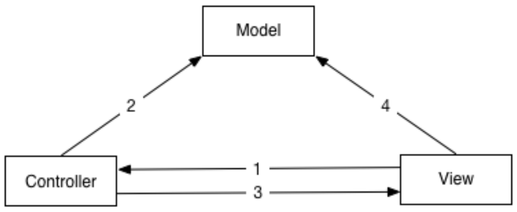


Figure 5: The collaboration between the Model, the View and the Controller.

class JButton. DotButton is based on Puzzler by Apple.You can review the code of the program "Puzzler" seen in lab 4 to help you with the class **DotButton**.

The main difficulty of the UI is the presentation of the board. If the board was a normal square, as shown in Figure figure7, it you be really easy to layout the **DotButton** instances on the Panel. For example, a GridLayout of size (n, n) would have worked. But the actual board of the game is not a perfect square. Instead, each line is shifted left or right when compared to the previous line (see Figure figure7). There is no immediate way of creating such a Layout easily.

We are going to use the following two steps approach: we will put each row of DotButton instances on its own instance of a JPanel class, on which a FlowLayout manager will be used. If you look at the documentation of JPanel, you will see that it is possible to add a border to the panel. Since the icons used by the **DotButton** instances are squares of size 40 pixels, the ideas is to add a border of size 20 at the correct location to obtain the desired result. The set of JPanels can then be added on the BoardView JPanel using for example an appropriate GridLayout manager.

Note that although instances of both **GameView** and **BoardView** classes have buttons, these instances 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 here:

- GameView Documentation
- GameView.java
- BoardView Documentation
- BoardView.java
- DotButton Documentation
- DotButton.java

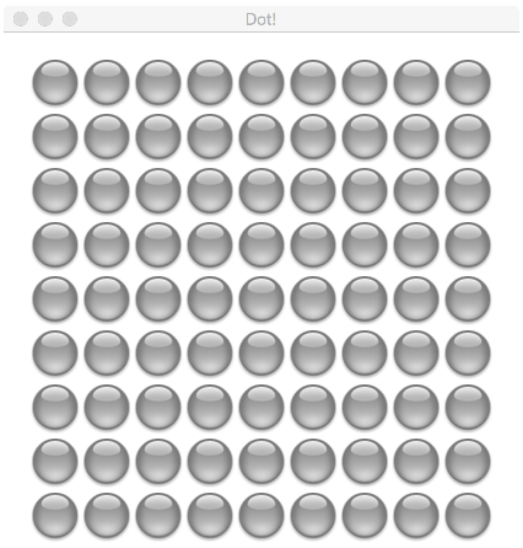


Figure 6: A perfectly square board.

- A zip file containing the icons. Taken from Puzzler by Apple.

The Controller (40 marks)

Finally, we have to implement the controller, via the class **GameController**. The constructor of the class receives as parameter the size of the board. The instance of the model and the view are created in this constructor.

The instance of the class GameController is the one managing the events generated by the interaction of the player with the game. It should thus provide the corresponding methods, and implement the logic of the game. During the game, after the player selects a new dot, the controller must decide what to do next. There are three cases to cover: the player won the game, the player lost the game, or else the blue dot must move to one of its neighboring dots to attempt to avoid encirclement.

In the first two cases, the controller must inform the player of the result, and in case of victory, must also provide the number of steps it took the player to win the game (Figure figure8). In order to achieve this, the controller can use the class method **showOptionDialog** of the class JOptionPane.

If the game is to continue, the controller must implement a strategy for an efficient selection of the next move for the blue dot. We suggest two strategies: one easy one to get the application going, and an efficient one.

- The easy strategy is to simply move the blue dot randomly to an available neighboring dot. This will allow you to get a first version of the game working. This is not the “real” solution, though, so if this is the only strategy you implement, you will not get full marks.
- The better strategy is to find the shortest path possible for the blue dot to exit the board, without going through a blocked dot (an orange dot). This is detailed below.

Breadth-First Search

To find the shortest path, we are going to implement a **breadth-first** search for a path from the current location of the blue dot to the border of the board. As we will soon see in class, such a breadth-first search is guarantied to find the shortest paths first. As we will also see, using a Queue is a one of the easy ways to do a breadth-first search.

To implement our queue, we will use an instance of the class LinkedList. To add an object to our queue, we will use the method **addLast**, and to remove an object from the queue, we will use the method **removeFirst**.

Here is a sketch of a breadth-first search algorithm using a queue. It starts from the position **start** and looks for the shortest path to one of the positions in the list **targets**. It has a list of position called **blocked** which are positions that cannot be used along the path. The list **blocked** contains initially the positions that must be avoided. During the search, we also use “blocked” to avoid building paths that cross each-other. In the pseudo-code below, we say that a position is “neighboring” another one if we can go from the first position to the second in a single move.

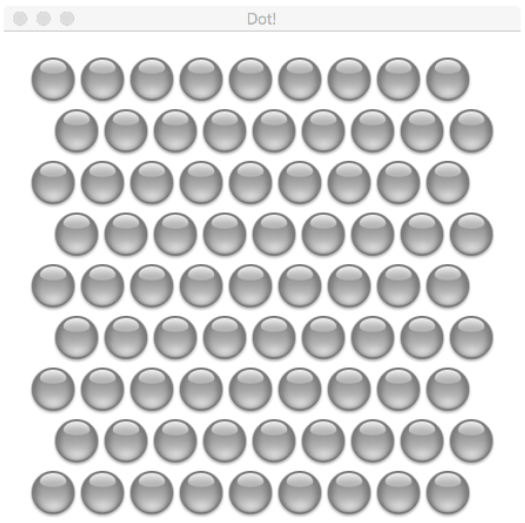


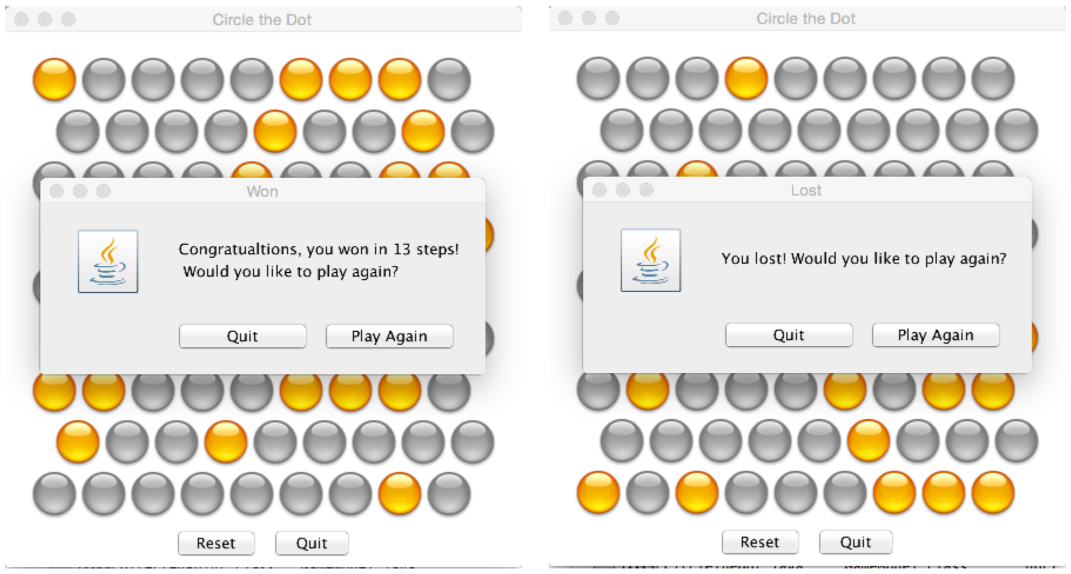
Figure 7: The actual disposition of the board.

```
Breadth-First-Search(start, targets, blocked)

 create a empty queue.

 add the path ``{start}'' to the end of the queue.

 While the queue is not empty Do
   Remove the path q from the head of the queue. Let c be the last
         position of that path
   For all positions p neighboring c
     If p is not in blocked Then
       If p is in targets then
         return the path ``q + {p}''
       Else
         add the path ``q + {p}'' to the end of the queue
         add p into blocked
       End If
     End If
   End For
 End While

 // The queue is empty and we have not returned a path. The targets
 //  are not reachable from position start.
 return FAIL
```

The instance of the class GameController is created by the class **CircleTheDot**, which contains the **main**. A runtime parameter can be passed on to the main, to specify the size of the board (at least 4). If a valid size is not passed, a default size of 9 is used.

A detailed description of the classes can be found here:

- GameController Documentation
- GameController.java
- CircleTheDot Documentation
- CircleTheDot.java



Congratualtions, you won in 13 steps!
Would you like to play again?

You lost! Would you like to play again?

Figure 8: Winning and losing the game.

Bonus (10 marks)

The pseudo code for the breadth-first search is deterministic. A smart player will always be able to anticipate the next move of the blue dot. Try to modify your game so that the blue dot next move is chosen randomly between all the possible shortest path (that is, the choice is still always the shortest path, but if there are more than one choice, and the one that is taken is chosen randomly).

Rules and regulation (10 marks)

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

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.

Files

You must hand in a zip file containing 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 "data" with the icon images in it
- The corresponding JavaDoc doc directory.
- StudentInfo.java, properly completed and properly called from your main.

As usual, we should be able to compile your code by simply extracting your zip file and running "javac CircleTheDot.java" from the extracted directory.

Last Modified: February 10, 2016