ITI 1121. Introduction to Computing II
Winter 2017

Assignment 3
(Last modified on February 14, 2017)

Deadline: March 6th, 2016, 11:59 pm

[PDF]

Learning objectives

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

Background information

We are going to create our own implementation of the game "FloodIt". Several versions of this game can be found, for example a version for android, on which this description is based, here. Several Web versions are available, for example here.

In this game, a square board is filled with dots, each having one of six possible colours. The players initially "captures" the top left dot. The player keeps choosing one of the six colours. Each time a colour is chosen, the dots that have already been captured turn into that colour. Each dot that is a neighbour[1] of a captured dot, and whose colour matches the newly selected colour becomes itself captured. The game ends once the entire board is captured (the board is "flooded" because all the dots have the same colour). As the game can only progress (a captured dot cannot be lost), the issue of the game is certain. The goal is thus to flood the board in as few steps as possible. Figure 1 shows two examples of the initial game configuration screen[2].

Figure 2 shows one step of the game. On the left, the currently captured dots, in green, before the player selects "purple" as the next colour (right). The neighbouring dots that were purple are now captured as well. The player wins when the entire board is captured (Figure 3).

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

[1] A neighbour is a dot with is one step over on a line or on a column.
[2] The UI is based on "Puzzler" by Apple.

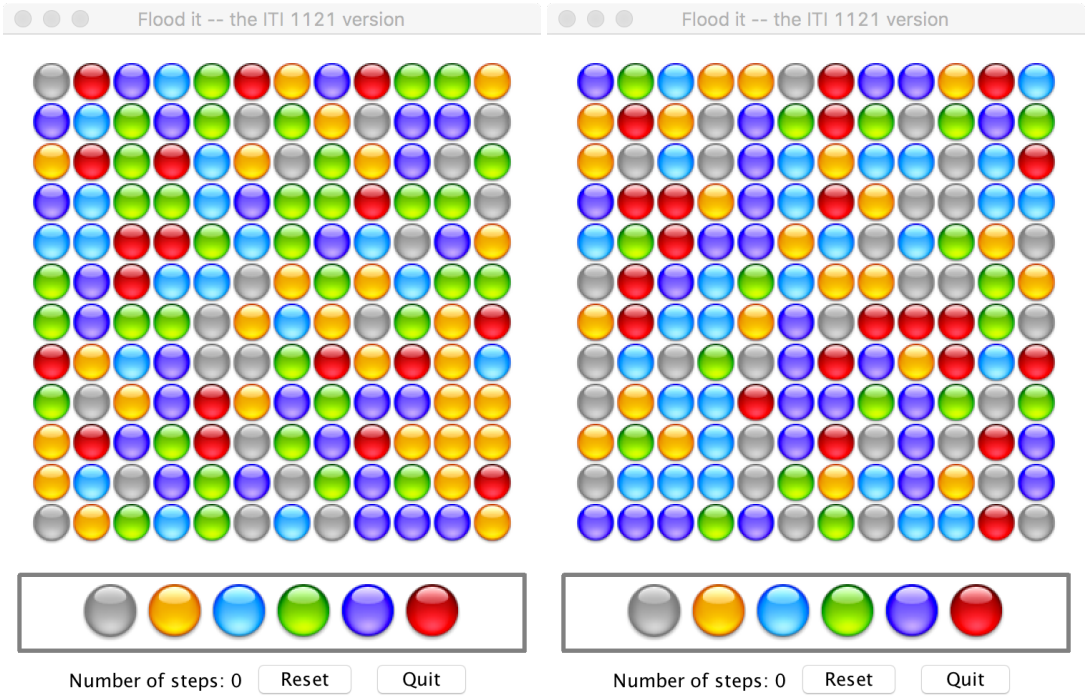

Figure 1: Two examples of the initial configuration of the game.

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 4). In our case, it works as follows:

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

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 colour and status (captured 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 dots captured so far
- the currently selected colour

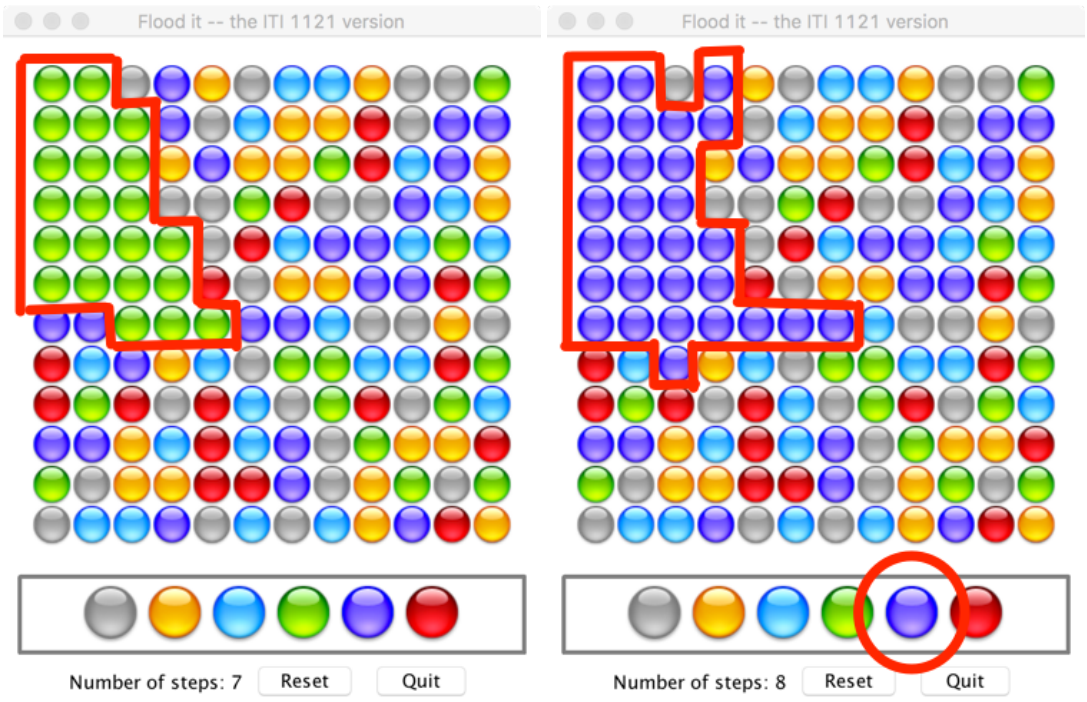

Figure 2: One step of the game.

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 set the current colour to a new value. Finally, it provides a way to (re)initialize the game, reallocating the initial colours randomly.

A detailed description of the classes can be found here:

- GameModel Documentation
- GameModel.java
- DotInfo Documentation
- DotInfo.java

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 selected colour (probably as a number). 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 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 in charge of the computing the consequences of the user selecting a new colour. The controller's **selectColor()** method is used to inform the controller that a **new** colour has been selected. It should then compute the new state of the game, and the see if the game is over or not. If it is, a message is being displayed (see Figure 3).

"Flooding" strategy

There are various ways that can be used to compute the next state of the game once a new colours is selected. In this assignment, we are going to rely on a stack. 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.

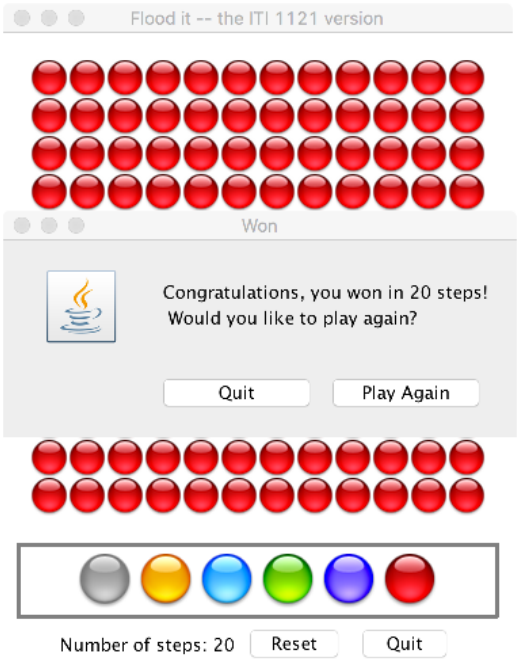

Figure 3: The player won in 20 steps.

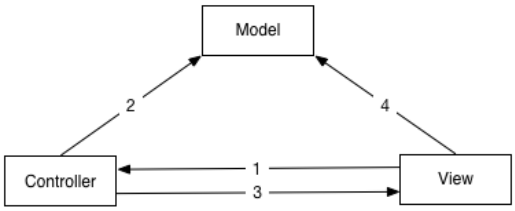

Figure 4: The collaboration between the Model, the View and the Controller.

Here is a sketch of an algorithm that you will use to implement the stack-based flooding strategy. The parameter **newColor** is the newly selected color.

```
Stack-Based-Flooding(newColor).
   create an empty stack.
   push every captured 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 captured and the color of n is newColor then
            capture n
            push n to the stack
         End If
      End For
   End While
```

The instance of the class GameController is created by the class **FloodIt**, which contains the **main**. A runtime parameter can be passed on to the main, to specify the size of the board (at least 10). If a valid size is not passed, a default size of 12 is used.

A detailed description of the classes can be found here:

- GameController Documentation
- GameController.java
- Stack Interface Documentation
- Stack.java
- FloodIt Documentation
- FloodIt.java

The view (40 marks)

We finally need to build the UI of the game. This will be also be done with a single 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 is a row of six large dots, which are used by the player to select the colours. Then, still above that, the board is displayed. The board is made of a series of dots , n lines and n columns of them (where n is the size of the board). Figure 5 shows the full GUI.

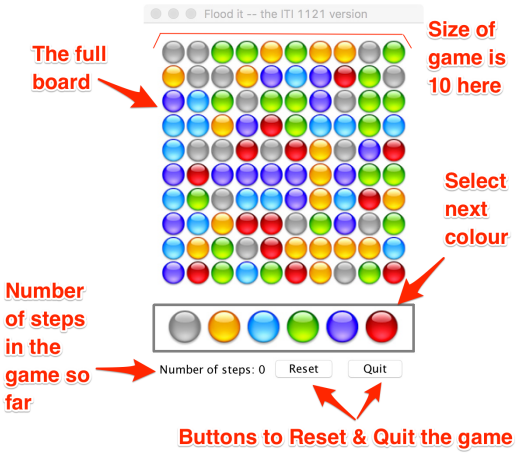

Figure 5: The GUI with a game of size 10.

If the size of the board is between the minimum of 10 and 25, icon of medium size should be used. If a size larger then 25 is used, then small icons should be used (Figure 6).

To implement the dots, 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**. 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 here:

- GameView Documentation
- GameView.java
- DotButton Documentation
- DotButton.java
- A zip file containing the icons. Taken from Puzzler by Apple.

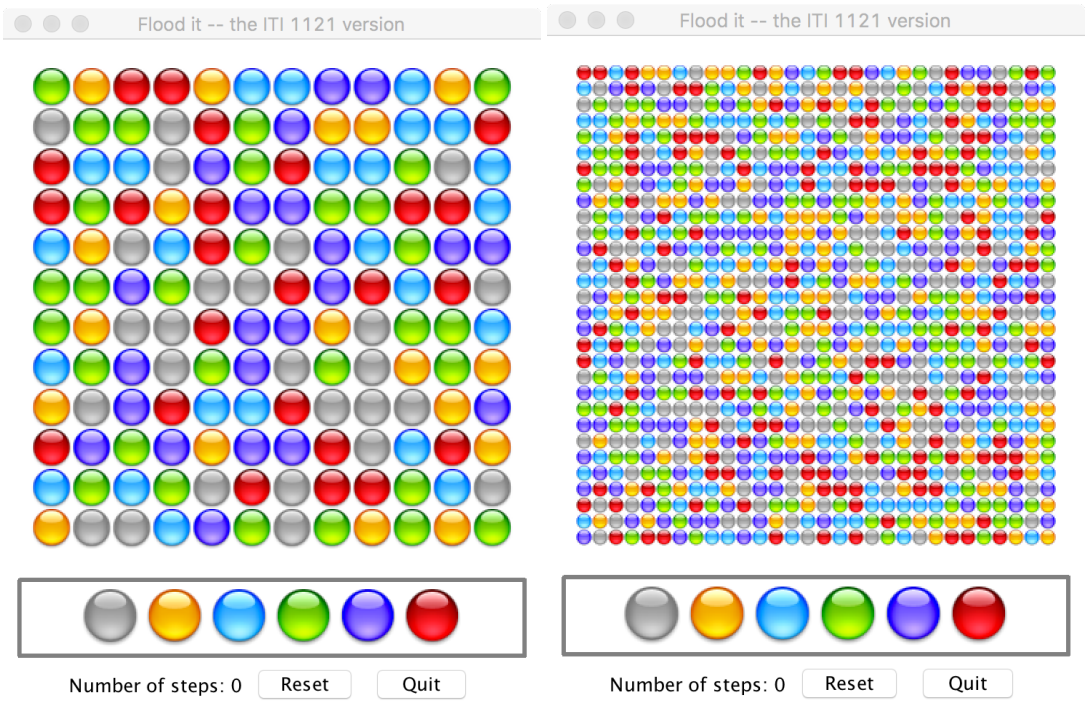

Figure 6: The GUI with a game of size 12 (left), and of size 30 (right)

Rules and regulation

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. You **cannot** change the signature of the provided methods.

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.

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 14, 2017