AP-21 – Programming Assignment #1- v2

November 9, 2021

Exercise 1 (Java Beans) – Tic Tac Toe

Tic Tac Toe is a paper-and-pencil game for two players who take turns marking the spaces in a three-by-three grid with *X* or *O*. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row is the winner.

We want to implement Tic Tac Toe using Java Beans. (*Note: There are several tutorials on the web showing how you can do it, but none satisfies the requirements listed below.*)

The system is made of a graphical dashboard, **`TTTBoard`**, containing the board, a **`TTTController`** label, and a restart button (see Figure 2 below). The board is made of 9 cells, which must be instances of a Java bean called **`TTTCell`**. Each cell has therefore the same appearance and the same logic. Conceptually, the **`TTTBoard`** dashboard displays the board and checks at each move if the game is ended, while the **`TTTController`** label monitors that the two players alternate correctly in a game.

Requirements for the `TTTCell` bean

A cell must be a square (e.g. a **`JPanel`**), completely covered by two buttons (e.g. two **`JButtons`**), initially labelled X and O. A cell has (at least) one property: **`State`**, initially set to "Initial", **which must be both bound and constrained** (in the JavaBean's sense). When the state is "Initial" and the user clicks on button X, an attempt to set **`State`** to "X" is made. If this change is not vetoed, the cell changes state, the background color of the clicked button is changed (you can choose your preferred colors for X and O), the O button is disabled and its label is set to the empty string, making evident that the cell is marked X. Dual effects must be obtained if the user clicks on button O. Figure 1 shows the effects of clicking on each button if the change is not vetoed. Note: clicking on a cell with state different from "Initial" should have no effect.

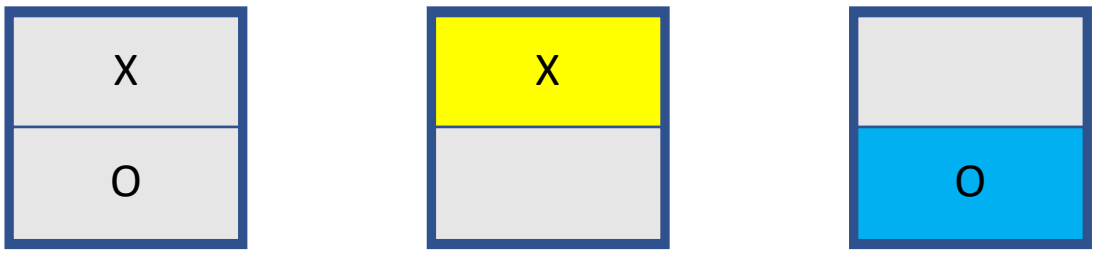

Figure 1: A cell in the initial state, after clicking X and after clicking O.

Cells must also provide support for a "reset" action that restores the initial state, for a "won" action that sets the background color of the enable buttons to green (ignoring the disabled buttons), and for a "disable" action that disables both buttons. Such actions are needed to support the functionalities of **`TTTBoard`**, as explained below, and can be implemented in any way (public methods are accepted, but event-based communication is more appreciated).

Requirements for the `TTTController` bean

The **`TTTController`** is a bean that has the graphical appearance of a label (e.g., it can extend **`JLabel`**). It has to check that the two players alternate correctly in a game. To this aim, it must be

registered as **`VetoableChangeListener`** to all the cells of the board. At the beginning it displays "**`START GAME`**". As for the first move, clicking either X or O on any cell is legal: if X is clicked then the **`TTTController`** displays "**`NEXT MOVE: O`**" (dually if O is clicked). During the game, if the button displays for example "**`NEXT MOVE: O`**", if player X tries to play clicking button X on a cell, then the state change is vetoed (dually for X and O exchanged).
The controller must also provide support for a "reset" action that restores the initial configuration, and for a "won" action that sets the background to green and changes the label text to "**`THE WINNER IS: Z`**" where Z is either O or X. Such actions are needed to support the functionalities of **`TTTBoard`**, as explained below, and can be implemented in any way (public methods are accepted, but event-based communication is more appreciated).

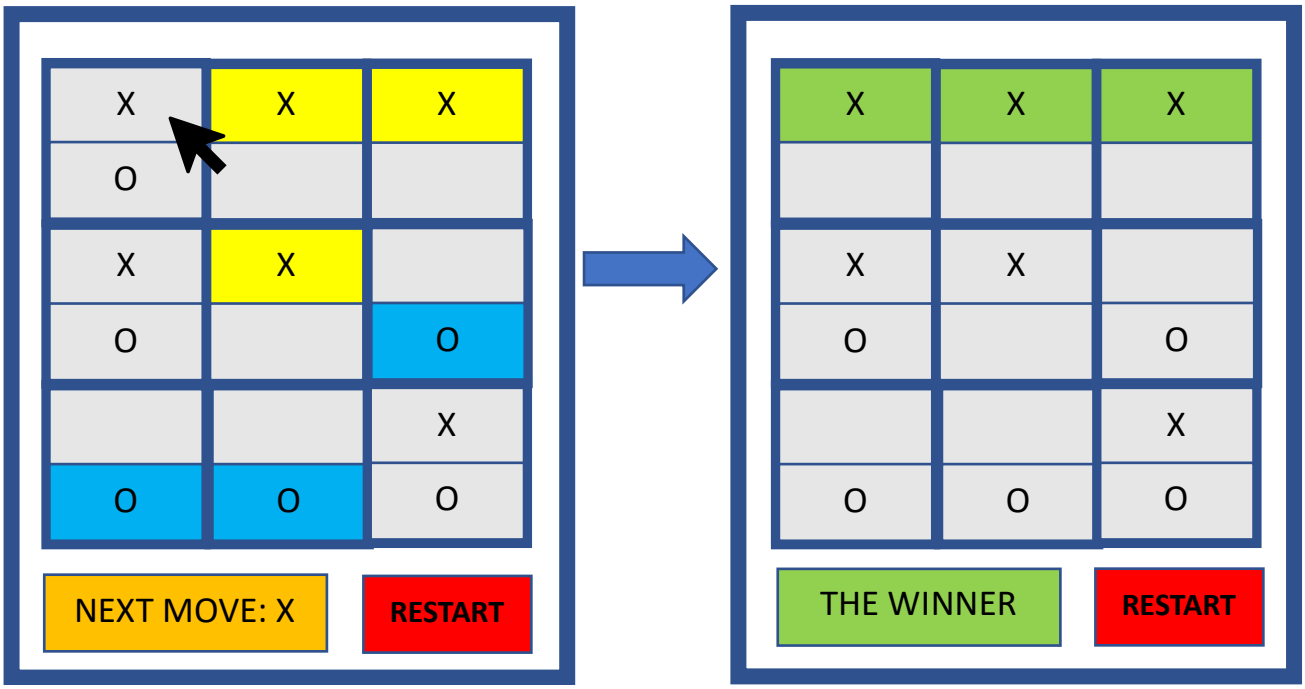

Figure 2: Player X clicks and wins.

Requirements for the `TTTBoard` dashboard

The main class of the application is the **`TTTBoard`** dashboard, that must be defined as extending **`JFrame`**. It dispays a grid of 3x3 **`TTTCells`**, a **`RESTART`** button, and a **`TTTController`** label (see Figure 2). The dashboard is registered as **`PropertyChangeListener`** to all the cells. After each move, the dashboard checks if the game is ended because either a (vertical, horizontal, or diagonal) row has been completed by one player, or because the board is full. In the first case the winning row is emphasized by setting the background color of the cells in the row to green (interacting with the relevant cells), the label text is changed to display the winner (interacting with the controller), and all the buttons of the grid of cells (but those of the wining row) are disabled.
Clicking on the **`RESTART`** button at any time must reset all the cells, the controller, and the dashboard to their initial configurations.

Solution format

Three adequately commented source files (**`TTTCell.java`**, **`TTTController.java`** and **`TTTBoard.java`**) and three corresponding **`jar`** archives, plus a short document (in PDF) reporting the main design decisions.

Exercise 2 (Java Reflection and Annotations) – Testing algorithms

Note: For this exercise you need to download and unzip the `crypto.zip` archive distributed with these instructions.

As you need to protect communications within your software company from hackers, you encharged an employee to implement a set of cryptographic algorithms. Each algorithm should be implemented by a Java class providing

1. A constructor taking an encryption key of type String as parameter
2. An *encryption* method taking a String as parameter and returning the encrypted string
3. A *decryption* method taking a String as parameter and returning the decrypted string

Unfortunately the employee decided to leave your company for a better job. Before leaving, he delivered to you the attached folder **`crypto.zip`** containg the **`.class`** files only and some auxiliary files, claiming that his obligation was fulfilled. He told you that, at least for some of the algorithms, the encryption method starts with "enc" and the decryption method with "dec", and that the encryption keys to be used were recorded in a file.
Clearly you don't trust completely the programmer anymore, thus you want to test his code. Do the following:

1. Define a **`KeyRegistry`** class, with methods:
    - **`add (Class c, String key):`** to add a new key for the crypto algorithm class **`c`**
    - **`get (Class c)`**: to get the last key associated with the class **`c.`**
2. Write a Java program **`TestAlgs`** that takes the parent directory of **`crypto`** as a command line argument (i.e., via **`argv`**) and:
    - Loads the list of keys in the registry: each line in the file **`crypto/keys.list`** is a space-separated pair made of the class name of the algorithm and the key needed for such an algorithm
    - For each class files in **`crypto/algos`**, first check if it is an encryption algorithm, i.e. it has (1) a public constructor, (2) a method starting with enc and (3) a method starting with dec, all three with one String parameter.
    - If it is not an encryption algorithm just print a corresponding warning on the standard output.
    - Otherwhise test the class as follows, using the secret words in file **`crypto/secret.list`**:
    - For each secret word **`wrd`** (i.e., each line in file **`crypto/secret.list`**), create an instance of the algorithm using the correspoding key as argument to the constructor. Next call the encryption method on **`wrd`**, producing say **`encwrd`**, and then the

decryption method on **`encwrd`** obtaining a string **`decwrd`**. If **`decwrd`** is different from **`wrd`** and it is also not equal to **`wrd`** followed by one or more *padding characters* **`#`**, then print on the standard output, **`KO: wrd -> encwrd -> decwrd`**.

Solution format: The Java source files implementing the described algorithm, including at least **`KeyRegistry.java`** and the main class **`TestAlgs.java`**.

Sample output: You can compare the output of your solution with file **`SampleOutput.pdf`**, which is the output of a correct solution (containing some non-printable characters).

Exercise 3 – Testing encryption functions, again, with annotations

Even if the testing program developed in the previous Exercise worked pretty well on some algorithms, some classes in **`crypto/algos`** were not identified as encryption algorithms. From a colleague of you former employee you learn that he liked very much to use annotations. Therefore, after inspecting folder **`crypto/annot`**, you suspect that also the remaining classes could implement an encryption algorithm: they use the annotations **`@Encrypt`** and **`@Decrypt`** to tag the encryption and the decryption method, respectively.

1. Write a Java program **`TestAlgsPlus`** that enriches the testing framework of the previous exercise in order to test also classes which contain (1) a public constructor, (2) exactly one method annotated with **`@Encrypt`** and (3) exactly one method annotated with **`@Decrypt`**, all three with one String parameter. If necessary, refactor the code of the previous exercise in order to reuse it as much as possible here.
2. Rewrite the source code of the **`@Encrypt`** and **`@Decrypt`** annotations, making them visible at runtime

Solution format: The Java source files **`Encrypt.java` and `Decrypt.java`** defining the two annotations, and the Java source files implementing the algorithm, including at least **`TestAlgsPlus.java`**.