ALGONQUIN COLLEGE

ASSIGNMENT 2 – Game Implementation (Piccross[1])

General View

Due Date: prior or on **Week 10 - Nov 13th 2021 (midnight)**

- **2nd Due date** (until 20th Nov) - 50% off.

Earnings: 15% of your course grade.

Purpose: Create the functional Game.

- This is the second task in JAP. We continue developing the "Piccross" game during this semester and the code will be updated to incorporate new functionalities.
- **PART I:** What you need to include to update the previous GUI:
  - MVC Model
  - The controller is responsible for start / restart the game;
  - The model contains basic data (to be used later in the distributed version);
  - The viewer is basically your standalone application, improving the GUI interface with menus, additional functions (ex: color) and some dialogs.
- **PART II:** Additional elements are required to define:
  - Code documentation (using Java code conventions)
  - Javadoc files.
  - Executable Jar.

Note 1: About Teams
The same teams defined in the A1 (GUI definition) can continue working. New teams are not allowed (unless accepted by your lab professor due to exceptional situations).

[1] The name "Piccross" is intentionally changed in comparison with the original "Picross" game since some functionalities will be different. This idea was originally proposed by Prof. Daniel Cormier, instructor in some labs this semester.

Part I – Updating the Piccross Project

1.1. USING MVC MODEL

The new version of **Piccross** (also based on "Picross" game – see, for instance http://picross.net/) will use the MVC (Model-View-Controller), dividing some responsibilities in different layers. Basically, we need to have:

- **Game class**: That just have the main function and declares the following entities:
  - **GameModel** gameModel = new GameModel();
  - **GameView** gameView = new GameView();
  - **GameController** gameController = new GameController(gameModel, gameView).

The idea is that **Game** can implement the MVC model with entities responsible for managing the activities (**Controller**), define the basic data properties and configuration (**Model**) and the GUI interface itself (**View**).

- **GameView class**: Basically, define methods to:
  - Splash screen: During initialization.
  - Initializes the game (creating all components);
  - Refresh and clean components during execution;
  - Show dialogs (ex: "About" or "Color Chooser");
  - Interact with user when he/she is playing with the game (updating colors, points, etc.)
- **GameModel class**: There just a few data that should be maintained:
  - Configuration string (ex: "00100,00100,11111,01110,01010" (or: "4,4,31,14,10") - you see it in the next session).
  - **Dimension** (number of rows / columns).
  - **Board**: The representation of the grid (dim x dim) that can be simply a 2-dimensional Boolean array.
  - Ideally, you can maintain the values for each column and row given by the board (used as labels/texts) in the GUI interface.
- **GameController class**: In this class, we can see basic functionalities such these:
  - Configures (using one pattern of binary string" and start the game;
  - Manage the **timer**;
  - Deal with **actionEvents** (typical from controllers).

Note 2: The MVC implementation
The definitions for all methods are free, since you respect the basic principles about the responsibility about each layer. One strategy is to start defining the interface (based on GUI) and identify the model that describes the game (for instance, the String) and finally, define the actions to be developed.

1.1. BASIC GAME

The new version of **Piccross** (also based on "Picross" game – see, for instance http://picross.net/) is based on the previous GUI application, but it is now including some additional functionalities…

Remember that your game must be played in order that users should be able to select the correct definition of the game.

Preview:

Fig. 1 – Example of execution (intentionally played wrong)

Basically, the game starts with a "blank" grid and some labels indicating what is the sequence of squares that belongs to one specific *pattern*. If you select one specific square, you have two possibilities:

- When you are in the **normal** mode (the "Mark" checkbox is not selected):

  - When the square belongs to the pattern, a color (in the above configuration, "color1 = green") is printed and you receive one additional mark (in the points field).
  - However, if the square is wrongly selected, a different color ("color3 = red in the above image), you receive a penalty (-1 marks).
- If you need to "mark" squares that you believe that are not belonging to the pattern:
  - If correctly marked, you also receive one additional point and a new color ("color2 = yellow" in the above configuration).
  - However, if you marked a square that should not belong to the pattern, you also receive a penalty (another -1 value with color3 = red color).
- The game is recording each action and one square, once selected, is not possible to be clicked again (disabling the action).
- While there are "blank" squares, you can continue playing and the timer of the game is updated.
- Finally, once all squares are selected, the game is ended.

In the previous (default) game, the "pattern" to be recognized should be similar to this one:

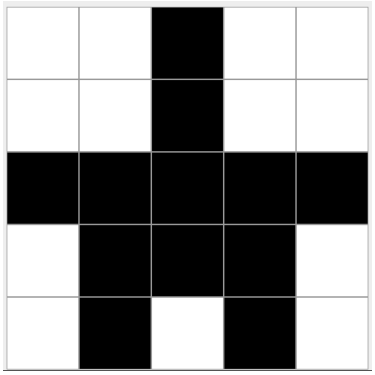


Fig. 2 – Default pattern (game "00100,00100,11111,01110,01010")[2]

1.2. REVIEWING THE INTERFACE

Now you can understand the behavior of the game:

- On the image, the left-up corner shows the checkbox (mark) that should be selected when you believe that one specific square should not belong to the mark.
- In the right side, you should update the points according to the previous definition.
- Each user action should also update one specific text (see the area in the right side), as a kind of "history" of the game. The list of messages is sequential and the text is

[2] The explanation about these numbers will be given in section 2.1, but alternative configuration using decimal numbers are also possible. Ex: "4,4,31,14,10".

completely free (ex: "NNN_set" or "NNN_clicked", where "NNN" is the name / identification of one specific component).

- Example (shown in the A1 specification[3]):

```
…
Mark set;
Pos 1,3 clicked;
Mark reset;
…
```

- During the game (while it is not finished), the timer is updated showing the seconds.
- It is also possible to "restart" the game (cleaning all the grid, cleaning the points and resetting the timer).
- The values for labels to be put in the game to help the user to identify the number of squares to be selected will be described later (section 2.1).

1.3. NEW ELEMENTS

The new version of the game must also include some menu options (also using **icons**):

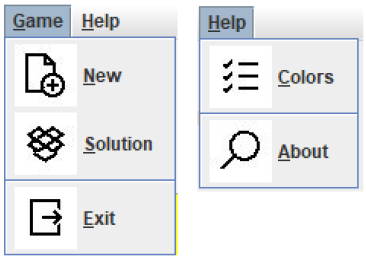


Fig. 3 – Menu options

- **Game**: Some functionalities about the game.
  - **New**: Creates a new game (with new configuration).
    - Ex: "01111,11100,10111,10110,01110" (see Fig. 7).
    - ***Note:*** *A new game must obey one specific configuration that will define the "initialization protocol" (see section 2.1).*
  - **Solution**: Shows the answer for the game.
    - ***Note:*** *The solution will release the correct answer for one specific game configuration using the colors defined – for instance, considering my current color configuration, we have the colors color1 = green or color2*

[3] The text message can be completely arbitrary.

= yellow (the color3 = red will not appear because we are showing the solution).

- **Example**:

Fig. 4 – Solution shown (game "4,4,31,14,10")

- **Exit**: Finalizes the execution.

- **Help**: Additional functionalities:
  - **Colors**: Change color configuration for execution (using Color Chooser dialog).
    - **Color1**: Used by correct pattern;
    - **Color2**: Used by marked pattern (squares that do not belong to pattern);
    - **Color3**: Error selected squares.

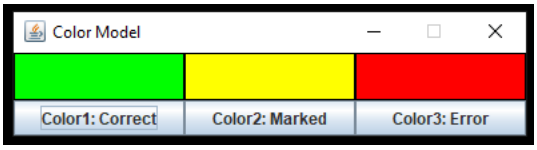
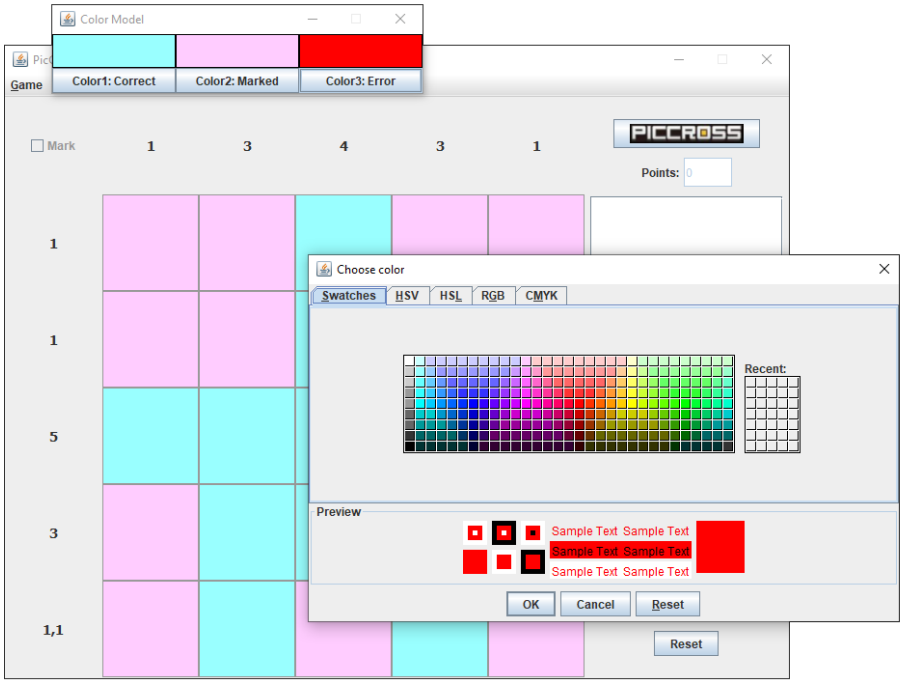


Fig. 5 – Default colors used in game

Note that when using different colors, the configuration should be changed. These colors can be adjusted according to the user preference and, once selected, it should be possible to change the presentation of the board. Ex:



- **About**: Basic info about the game (similar to Splash function defined in the previous assignment): using Show Message dialog.

Fig. 6 – About message.

Part II – The Game Logic

This is the most interesting activity in this assignment: how to configure the game and the logic to be used in the “Piccross” game.

2.1. BASIC REPRESENTATION (MODEL)

Here are the basic rules:

- The game is using a square grid. For instance, 5x5[4].
- It means that, in binary representation, the way to represent each row (line) vary from "00000" to "11111".
- For instance, the game shown in the Fig. 2 and 4 should be represented by the following values (in binary and also decimal):

$$(00100)_2 = (4)_{10}$$
$$(00100)_2 = (4)_{10}$$
$$(11111)_2 = (31)_{10}$$
$$(01110)_2 = (14)_{10}$$
$$(01010)_2 = (10)_{10}$$

- For this reason, considering that we are using 5 squares, the configuration is given by one specific binary representation: "00100,00100,11111,01110,01010", where the dimension (5) can be calculated using ***tokenizer*** functions over the binary configuration.
- It means that the game must be created when reading one specific sequence of binary numbers.
- Now, considering these values, we need to define the set of contiguous sequence of numbers 1 (the key to represent the columns and rows to our game. Let's see some examples:
  - The first row (00100) that has only one 1, will show only the value "1";
  - Similarly, the second column is (00111) and will indicate the number "3" for this column.
  - Finally, checking the last row, we have (01010), since we have 2 number 1's not contiguous, the value to be put in this label should be "1,1" (or similar representation).
- These values must be shown as labels to define the logic from the game.

Note that this binary representation is also important to be defined as a **board** in the game that you can test the matching during the execution (when user is clicking over one specific tile / square and is getting points.

[4] To get **bonus**, as explained in section 2.4, this number can vary (**suggestion**: between 2 and 10), which will imply in a different range of binary values and, therefore, multiple possible configuration for grid, columns and rows.

2.2. INITIAL CONFIGURATION

Basically, the game can configuration can be defined by the following steps:

1. Define the initial dimension (ex: **D = dim** = 5)[5];
2. Generate one specific (**random**) input composed by binary string. Remember that each binary number must have values are between 0 and 2^D-1.
3. Using these values in a binary representation, define the numbers that must appear in the rows and columns. These numbers will give the necessary information to user about the sequence of squares that must be selected.

For instance, a new possible configuration (using the same “dim = 5”) can be given by:

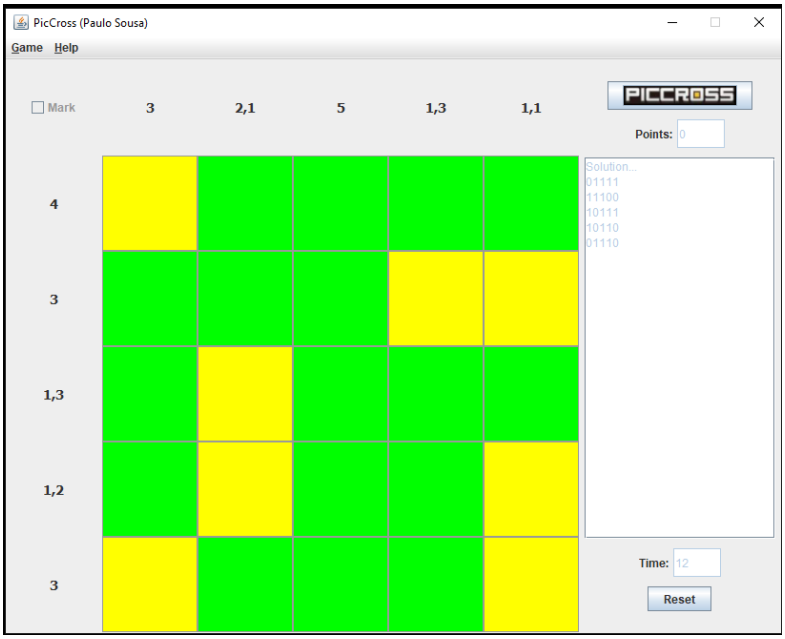


Fig. 7 – Example of new game configuration (game "01111,11100,10111,10110,01110").

2.3. UPDATING CONTROLLER

The **GameController** has already been started before, in order to deal with actions by listeners as a kind of event handler. Now, we need to use the controller to create the configuration to the game, using different methods.

- **Before the game:** Note that the creation of game and new configuration is the action required by the “**New Game**” option, following the steps defined in the previous section.

[5] Remember that if you do not want to get bonus, **D (dim)** can be fixed (D = 5), but the values must be randomly generated.

- ***Note:*** *Remember that it is required to respect the ranges, using the binary representation to adjust the labels that can represent the values of columns and rows in the game.*

- **During the game**: When user is selecting one specific square, the actions to show the colors (if they are matching or not with a pattern) must also be treated as events.
  - After each initialization (or refresh from the game), the **timer** will be activated and the seconds will be shown in the panel (see annex 1).
  - About the **points**:
    - While the game is not finished (when all tiles from grid are selected), if the user is clicking over one specific square, that matches with the image, he/she is getting one additional point.
    - Similarly, if he has "marked" appropriately, he is receiving another point. However, wrong selection will imply in losing one point (remember to avoid to give a negative score).
  - At any time, the game can be **reset** and the grid returns to original configuration (without selection).
  - The functionality to show the solution (as indicated in the Fig. 4 and 7) is one action required by the "**Solution**" menu.
  - Remember that, you can also **change the colors** of the game, according to the color chooser that you can use.
- **When game ends**: After selecting all squares, the points and timeout are shown. Eventually, a new game can be started.
  - **Perfect game**: If the player gets a full (complete) solution a kind of winner message must be shown. Ex:

Fig. 8 – Example of image showing perfect score.

    - Note that in this case, the number of points must be equal to D*D.
  - **Common finalization**: If in the end, the player has committed one or more mistakes, your game is "over" and you can simply show a dialog like this:

Fig. 9 – Example of image showing the game when finished without perfect score.

Note 3: Functions to be implemented
*All these functionalities should be implemented by appropriate **functions** in one specific class (**Model** – **View** – **Controller**). You can also develop "**auxiliary**" functions. Examples: How to create random binary values between ranges (during the configuration) and also several getters/setters considering the value for D = dim).*

Finally, the purpose of this game developing is that you can exercise your logic to create this game, using the knowledge and experience that you already have in Java and using them in this scenario.

2.4. BONUS MARKS

Bonus marks will be given if the new Game can accept different dimensions (and therefore, a more dynamic configuration). For instance:

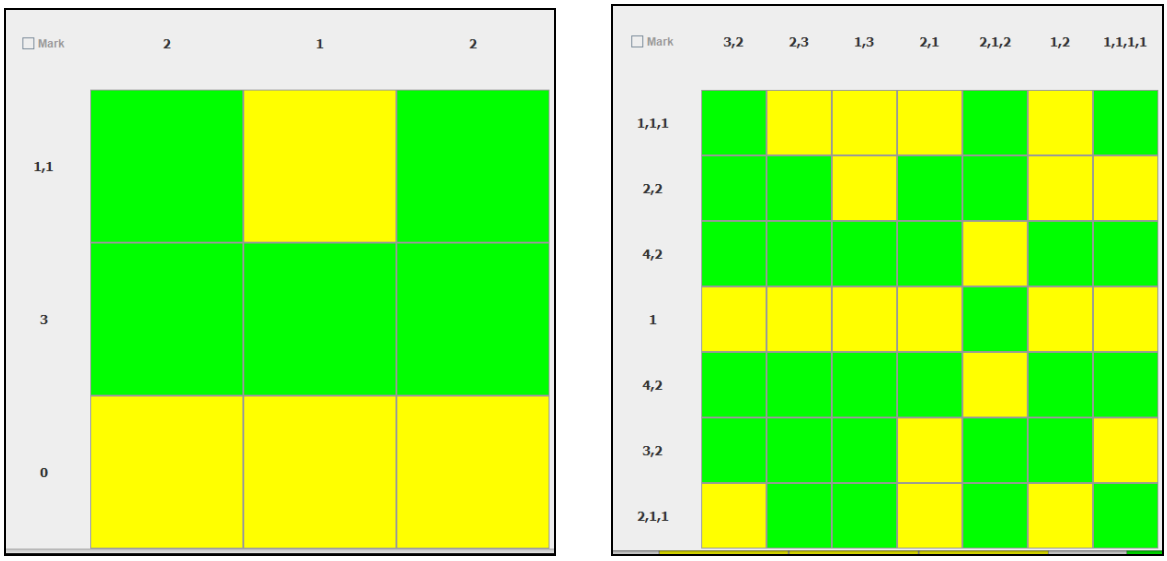


Fig. 10 – Additional examples using different dimensions (D=3 and D=7).

- **TIP**: Use the approach of using **lists** to any component (ex: buttons, labels, etc.) to let you create games with different dimensions.

2.5. WHAT WILL COME LATER…

Note 4: Future Game Behavior
The activity now is only be able to execute the game in a standalone mode. The following description is anticipating some aspects of the game that will be updated / changed in future assignments.

- *In the future implementation, the game is supposed to be played by **networking**. For this reason, we will use some definitions such as initial string configuration.*
- *Later, each user will be invited to play the game (in a scheduled procedure) and some ranks will be defined to show the results and create a rank about champions.*
- *Databases will be used to record some information from the game (coming from **Model**).*

Part III – Documentation and Packing

3.1. RESOURCES PROVIDED

We are providing the images used in the version of Piccross shown, but you are free to make modifications, including new the icons, different images for dialogs, etc.

3.2. BASIC DOCUMENTATION

CODE ELEMENTS (*Basic criteria*)

- Code well organizes, methods and functions grouped logically, classes in order, etc.
- Label naming (variables, constants, methods, functions, etc.) consistent and meaningful.
- Code consistently indented and spaced (define a consistency /organization).
- Good commenting which includes block comments.
- In the case of teams, Git files (or documents / images that can show the collaborative development) are required to be include.

JAVADOC (*following Java Standard*)

- Javadoc outputs are required, and the pages / folders must be included.

JAR (*Executable file*)

- Finally, it is required to create and include JAR files (that can open the binary using Java)

3.3. STANDARD

- ❖ Please read the Assignment **Submission Standard and Marking Guide** (at "Assignments > Standards" section).
- ❖ **About Plagiarism**: Your code must observe the configuration required (remember, for instance, the "splash screen" using your name. Similarly, we need to observe the policy against ethic conduct, avoiding problems with the 3-strike policy...

3.4. RUBRICS

Marking Rubric

Maximum Deduction (%)	Deduction Event
up to 100	**Severe Errors:**
100	Late submission (after 1 week due date)
up to 100	Does not compile with javac
up to 100	Compile but crashes after launch
up to 100	**Does not comply with the problem specifications**
up to 50	Game is not functional: impossible to play and to get actions.
up to 50	The GUI is not including the new elements (menus, dialogs, colors), including icons.
up to 30	Exception treatment not done (ex: if images are not found)
up to 20	No timer, no points updates.
up to 10	Inadequate comments (headers, code explanation), insufficient javadoc
up to 10	Missing or incorrect classes and package
up to 10	Other minor errors
up to 20	Bonus: uses different values for D (dim).
Final Mark	**Formula:** 15*((100- ∑ penalties + bonus)/100), max score 15%**.**

Submission Details

- **Digital Submission: Compress** into a **zip** file with all files (including source code, images, Javadoc, Jar).
- Upload the **zip** file on Brightspace. The file must be submitted prior or on the due date as indicated in the assignment.
- ***IMPORTANT NOTE:*** The name of the file must be **Your Last Name** followed by the last three digits of your student number followed by your lab **section number**. For example: **A11_Sousa123.zip**.
  - If you are working in teams, please, include also your partner. For instance, something like: **A11_Sousa123_Cormier456.zip**.
- ***IMPORTANT NOTE:*** **Assignments will not be marked if no source files are included.** All assignments must be successfully completed to receive credit for the course, even if the assignments are late.

File update: Oct 10th 2021.

Good luck with A2!

Annexes

ANNEX 1 – About Board Configuration (to be used in the Model)

No matter the model (if you use a binary or decimal representation), when you are configuring the board, you will receive a sequence of numbers. Ex: "00100,00100,11111,01110,01010".

- **TIP**: The simpler strategy to get one specific sequence (considering that you are separating them with commas (',') is using default tokenizer methods.

```
import java.util.StringTokenizer;
```

Use one variable to do it:

```
StringTokenizer st;
String valStr;
```

Example of code:

```
st = new StringTokenizer(args, ",");
valStr = st.nextToken();
```

- **Note**: The same pattern (comma separated values) is supposed to be followed when you are creating a random game (when "New Game" is selected).

ANNEX 2 – About Timer (to be used in the Control)

There are multiple ways to implement a **timer**. Here, I am showing one basic that uses classes from **Java.util**:

```
import java.util.Timer;
import java.util.TimerTask;
```

Declare local variables. Ex:

```
// Timer initialization
private int seconds = 0;
Timer timer = new Timer();
TimerTask timerTask;
```

Then, in a class (ex: **GameController**), initialize the timer invoking the run() method (that will be discussed later in **Threads**):

```
public void startTimer() {
    // Timer task
    timerTask = new TimerTask() {
        public void run() {
            seconds++;
            // Update your interface
        }
    };
    try {
        timer.scheduleAtFixedRate(timerTask, 0, 1000);
    } catch(Exception e) {
        // Eventual treatment
    }
}
```