ASSIGNMENT 2.2 – MVC Model (Battleship[1])

General View

Due Date: prior or on **Jul 9th 2023 (midnight)**

- **2nd Due date** (until **Jul 16th 2023**) - 50% off.

Earnings: 16% of your course grade.

Development: Activity can be done **individually** or in teams (**only 2 students** allowed).

Purpose: Define the MVC model for the Battleship Game.

- This is the second implementation in the JAP course. The purpose is to implement the game using DP (Design Patterns), specially the MVC (Model-View-Controller) structure for the game.
  - **TIP:** In the following sections, check the **TODO** activities in the following sections.
- **PART I:** Considering the GUI defined in the **A11** (and implemented in the **A12**), define the MVC pattern to be used:
  - What is the **controller** (use this part to define behaviors)?
  - What is the **view** (use this part to update the interface)?
  - What is the **model** to be used (consider all data that / properties are relevant to the game?
- **PART II:** Considering your application:
  - Detail the differences between your application and the previous configuration.
  - Evaluate any other pattern that could be used by your application. Additional elements are required to define:
    - Code documentation (using Java code conventions)
    - Javadoc files.

[1] Check the A11 specification for more details.

- Executable Jar

Part I – Updating the Game Implementation

1.1. Game Basics

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

The new version of **Battleship** must use **DP** (Design Patterns) and, specifically, the **MVC** (Model-View-Controller). 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**).

Note 1: **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.

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: "0444400000000010030220020300000203010000000003010022030010000300" - you see it in the next session).
- **Dimension** (that indicates the maximum size of a ship).
  - *Remember:* the board size (number of rows / columns): dimension*2.
- **Board**: The representation of the grid (board x board) that can be simply a 2-dimensional array.

- **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).

1.2. Interface Review

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

Preview:

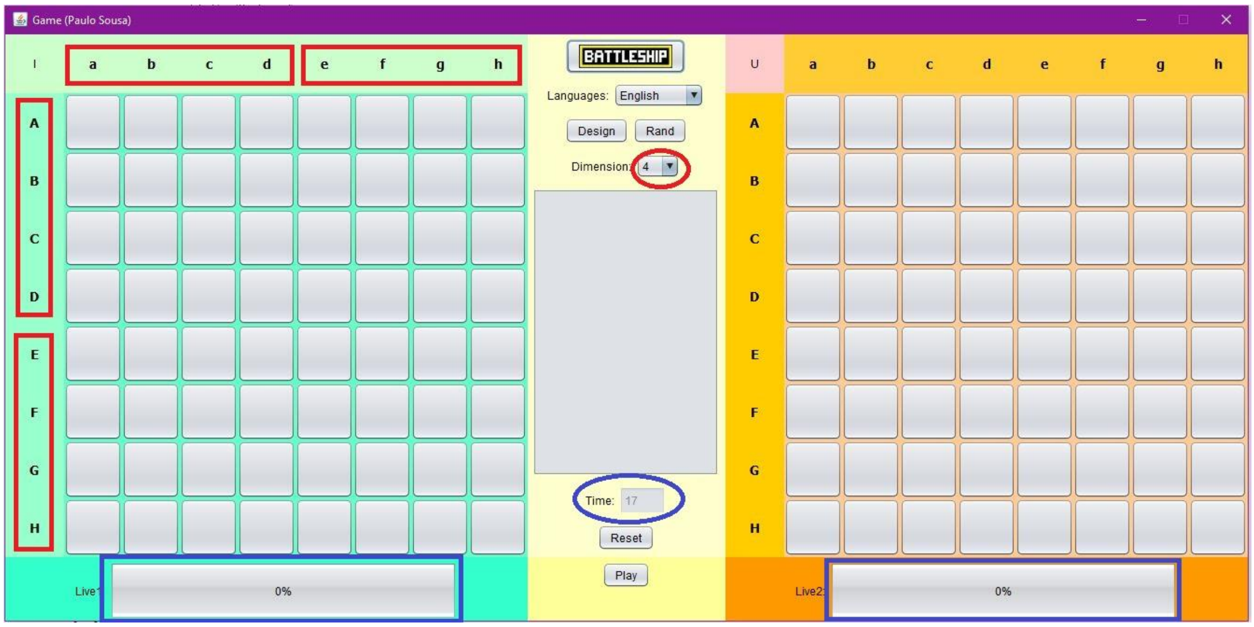


Fig. 1 – Example of initial interface

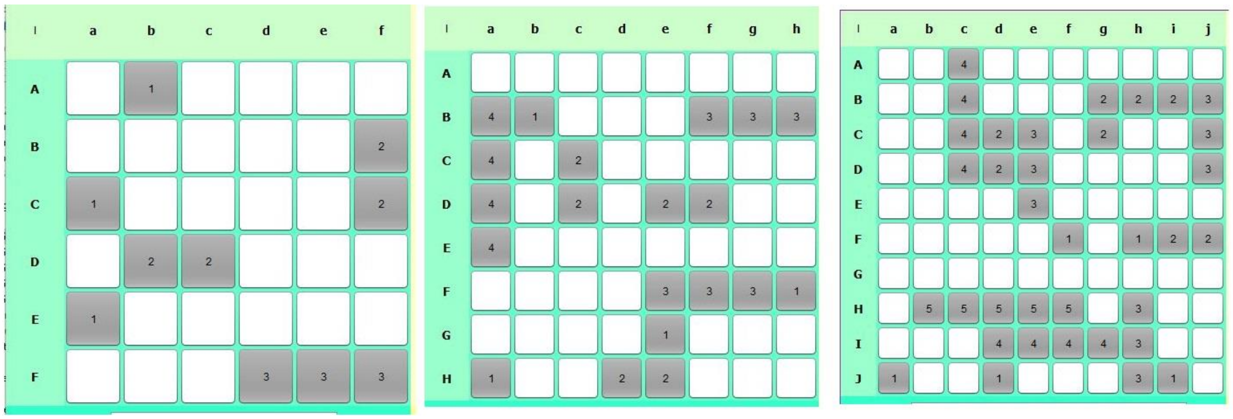
Basically, the game starts with a "blank" grid and some labels indicating what is the sequence of squares that belongs to one specific *pattern*. In this game, we have two different boards: the human (left / green) and the computer (right / orange):

- Remember that the game starts with the "**design**" mode: you define the configuration (ex: language), but most importantly, the dimension (**dim**).

- Be sure that once defined, you should start playing (“**play**” mode).
    - Remember that we have two players: the human and the computer.
    - The human selects position manually, while computer selects random positions.
    - Correct selections (or “firing” boats) gives points, otherwise, just water (no points).
    - The game ends when all ships are found and you can save the points.

❖ **TODO**:

The Game can accept different dimensions (and therefore, a more dynamic configuration). For instance:



I	a	b	c	d	e	f
A		1				
B						2
C	1					2
D		2	2			
E	1					
F				3	3	3

I	a	b	c	d	e	f	g	h
A								
B	4	1				3	3	3
C	4		2					
D	4		2		2	2		
E	4							
F					3	3	3	1
G					1			
H	1			2	2			

I	a	b	c	d	e	f	g	h	i	j
A			4							
B			4				2	2	2	3
C			4	2	3		2			3
D			4	2	3					3
E					3					
F						1		1	2	2
G										
H		5	5	5	5	5		3		
I				4	4	4	4	3		
J	1			1				3	1	

Fig. 2 – Additional examples using different dimensions (D=3 and D=7).

➢ You need to be able to create a splash screen before the game itself. For example, you can show a kind of image such as:

Fig. 3 – Game splash

Basic features should also include:

- **Timer**: During the game (while it is not finished), the timer is updated showing the seconds.
- **Design**: Where you can put ships manually in the game.

- **Ex**: One suggestion is to create one window interface where user can select ship sizes, direction and click over specific positions.

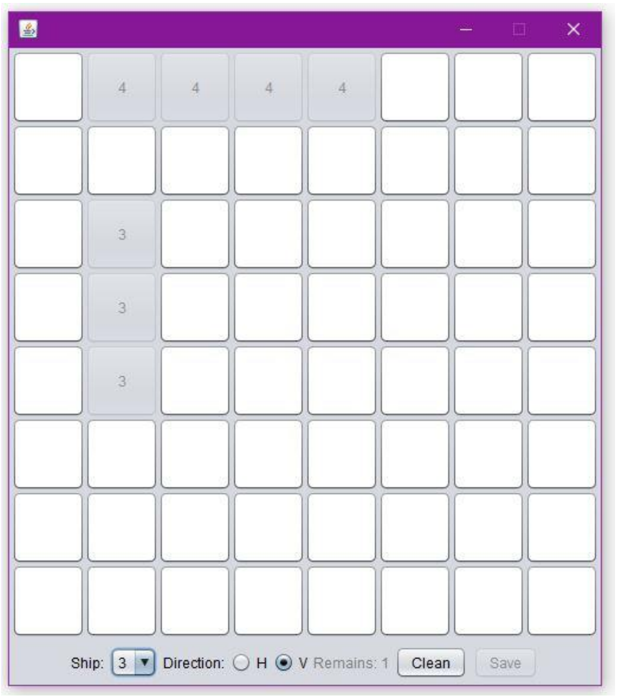


Fig. 4 – Menu options

- **Reset**: It is also possible to "restart" the game (cleaning all the grid, cleaning the points and resetting the timer).

1.3. NEW ELEMENTS

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

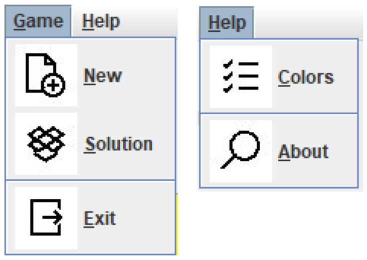


Fig. 5 – Menu options

- **Game**: Some functionalities about the game.
  - **New**: Creates a new game (with new configuration).
    - ***Note:*** *Remember that the game must follow the rules about ship distribution.*
  - **Solution**: Shows the answer for the game.

- **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.

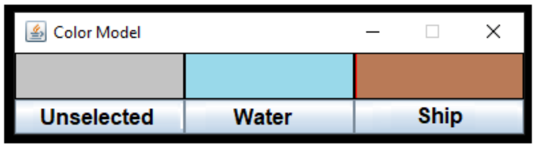


Fig. 6 – Default colors used in game

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:

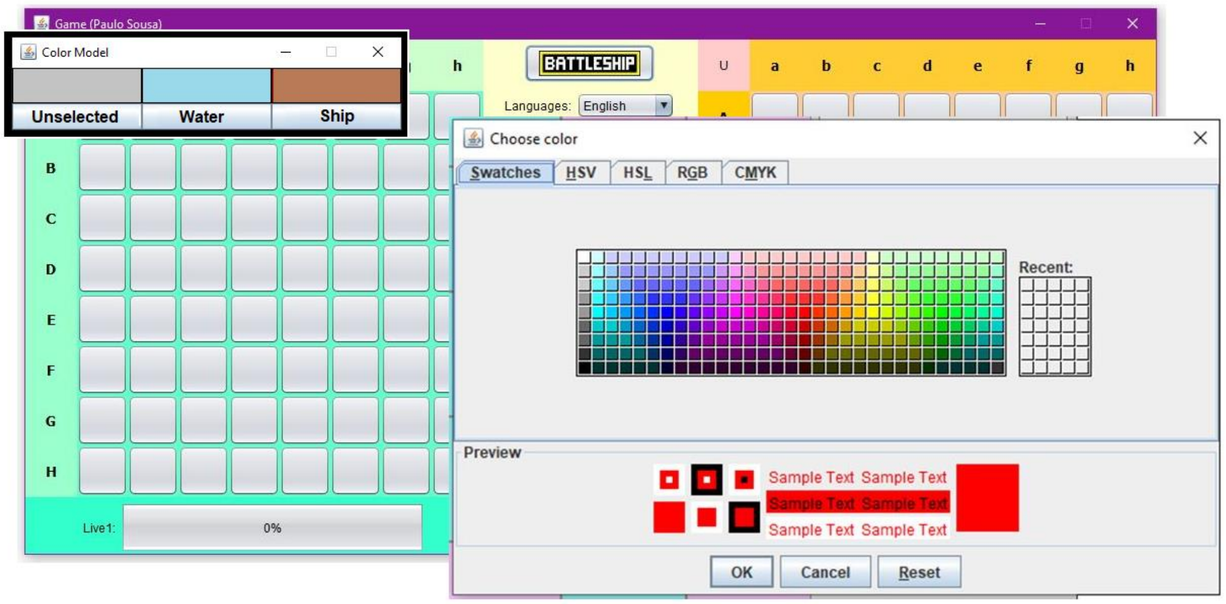


Fig. 7 – Color dialog

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

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 "**Battleship**" game.

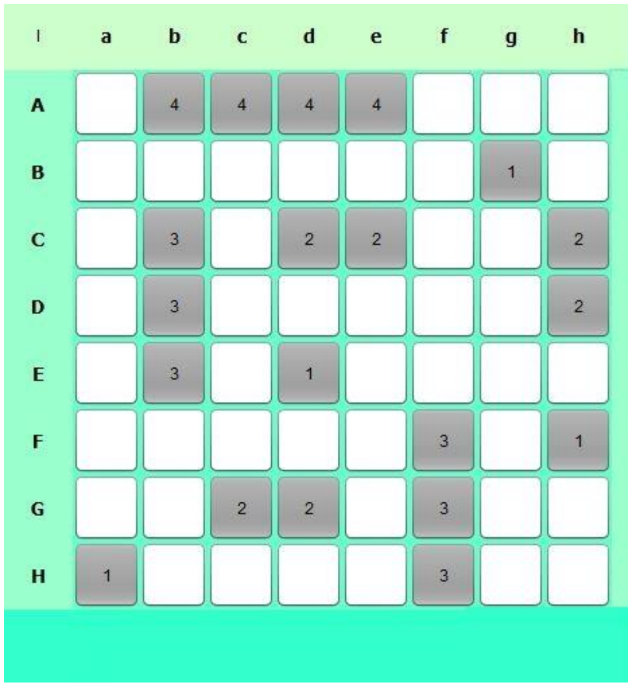
2.1. BASIC REPRESENTATION (MODEL)

Here are the basic rules:

- The game is using a square grid. For instance, dim = 4.
- It means that, in numerical representation, given by one proper configuration: “0444400000000010030220020300000203010000000003010022030010000300”.
- The model used was based on this configuration was:

```
04444000
00000010
03022002
03000002
03010000
00000301
00220300
10000300
```

- This is equivalent to the following view:



I	a	b	c	d	e	f	g	h
A		4	4	4	4			
B							1	
C		3		2	2			2
D		3						2
E		3		1				
F						3		1
G			2	2		3		
H	1					3		

Fig. 8 – From model to view example

2.2. INITIAL CONFIGURATION

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

1. Define the initial dimension (ex: **D = dim** = 4)[2];
2. Generate one specific configuration for human player (randomly **or** manually) input composed by numerical values (zero means no boat, while numbers – from 1 to D – indicates the ship).
3. Note that the computer configuration is always random.
4. Remember the formulas:
    - D = Dimension (ex: 2 to 9)[3]
    - B = Board (2*D)
    - T = Tiles (B*B)
    - S = Sum of boats (given by the formula): D*1 + (D-1)*2 + … 2*(D-1) + 1*D
        - Solution: ∑(n-i+1)*i = n(n+1)(n+2)/6 = (D*(D+1)*(D+2))/6
    - R = Ratio: S / T [Range: 0.25 > 0.5)

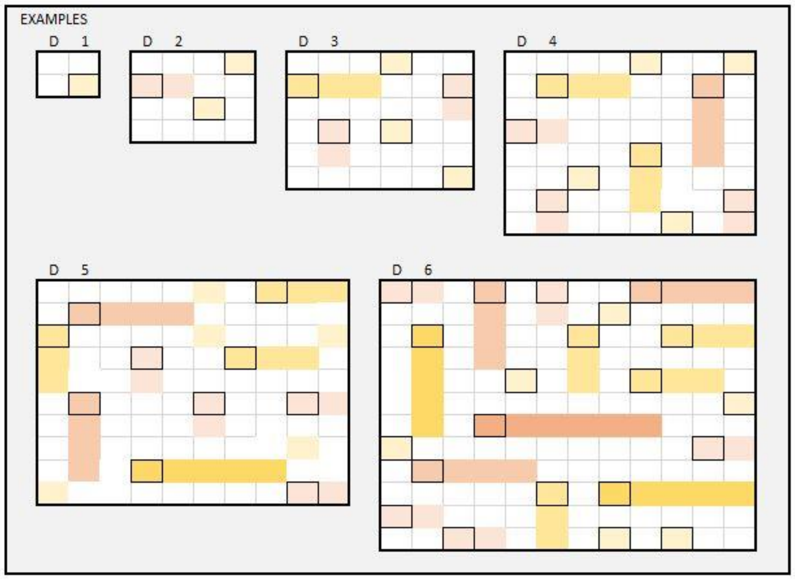


Fig. 9 – Game examples (random dispositions)

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.

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

Note 2: The Controller implementation

By principle, you can define as much as controllers by listeners as you want, including anonymous classes. However, the recommendation is to centralize all actions in one single controller.

Basically, we need to have:

- **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.
- **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**[4] will be activated and the seconds will be shown in the panel.
    - About the **points**:
        - While the game is not finished (when all ships are discovered), if the user is clicking over one specific square/tile, that matches with the image, he/she is getting one additional point (or the "life" expectancy of the adversary is decreased).
    - At any time, the game can be **reset** and the grid returns to original configuration (without selection).
    - The functionality to show the solution 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[5]. Ex:

Fig. 10 – Example of image showing perfect score.

[4] Different versions can be applied. Ex: countdown timer that can be used as a different difficult levels.
[5] Game images are provided as suggestions for students

- **Correct game solution**: If in the end, the adversary (game) wins, you can simply show a dialog like this:

Fig. 11 – Example of image showing the game when finished without perfect score.

- Note that you can win / lose before selecting all tile positions.

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: "tips / hints".*

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.

Part III – Documentation and Submission

Since you are in another document proposal, focus on answer the "**TODO**" activities mentioned in this specification. Remember: right now, you do not need to implement nothing.

- **SUMMARY:** In short, create a document, where you specify the details of your MVC **Battleship** version.

Note 4: About Teams
Only teams already defined are allowed (and just one member should submit). It means that, if you decided to work alone, you will continue until the end of the course.

Evaluation

❖ 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...

Submission Details

- **Digital Submission: Compress** into a **zip** file with all files (including document).
- 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: **A21_Sousa123.zip**.
  - If you are working in teams, please, include also your partner. For instance, something like: **A21_Sousa123_Sousa456.zip**.
- ***IMPORTANT NOTE:*** **Assignments will not be marked if there are not source files in the digital submission.** Assignments could be late, but the lateness will affect negatively your mark: see the Course Outline and the Marking Guide. All assignments must be successfully completed to receive credit for the course, even if the assignments are late.

Marking Rubric

Maximum Deduction (%)	Deduction Event
-	**Plagiarism:**
Check	3-strike policy[6] (**AA32**, **SA07** and **IT01**)
-	**Severe Errors:**
8 pt (50%)	Late submission (after 1 week due date)
16 pt (100%)	Missing demo (zero[7])
-	**Execution Problems:**
4 pt (25%)	Script execution failure (unable to create Jar/Javadoc)
4 pt (25%)	Unable to play the game (using "default" / initial configuration)
2 pt (10%)	No internationalization (unable to change language)
2 pt (10%)	No design mode (unable to change configuration – ex: new dimensions)
2 pt	Unable to save/load game (configuration / design or play mode)
1 pt	Missing splash screen
1 pt	Unable to change colors
-	**Assignment Elements:**
-	**Part I – MVC**
2 pt	Missing architectural elements (View)
1 pt	Missing event handler strategy (Controller)
1 pt	Missing structure / data definition and analysis (Model)

[6] The plagiarism detection will imply in the "3-strike" policy: starting with ZERO, then moving to course failure or program cancelation (see the Algonquin College documents: https://www.algonquincollege.com/policies/).

[7] If a course requires demos, they are not optional. If a student does not demo their work, they should receive a grade of 0 on that assessment, not a grade reduction.

2 pt	Problems with MVC integration
0.5 pt	Other errors
-	**Part II – Additional elements**
0.5 pt	Missing evolution details
0.5 pt	Missing DP aspects
1 pt	Other errors (ex: missing menu, functionalities)
1 pt	Github utilization
-	**Bonuses**
1 pt	Bonus: interesting ideas, enhancements, discretionary points (2%).
Final Mark	**Formula:** 16*((100- ∑ penalties + bonus)/100), max score 18%**.**

File update: Feb 18th 2023.

Good luck with A22!