CS1006 Lights Out Practical 200014517/200020128/200015381 Tutor: Ognjen Arandelovic

04 Apr 2021

Summary

Our program can create a window with the java swing GUI that runs allows the user to choose between 3 different modes of lights out. Those modes being 15 levels of normal, 15 levels of coloured mode and a board editor allowing the user to create custom levels by editing individual buttons and then solve them.

The program features a "Moves made" counter and display showing the time taken on each level. It supports two variants – A coloured board using 3 different colours and a time limit which is set for each level.

The program reads in csv files for the levels so the levels can be easily changed by just changing the csv files. The GUI presents the user with simple options to select the different modes and displays the grid of lights as buttons to click.

Design

For the general class structure:

- The board class handles buttons grid, updating and displaying buttons.
- The button classes handle user presses, storing button state and displaying appropriate colours
- The game class handles a single play of several levels, and any applicable state
- The ColouredGame class is a specialised game for the coloured variant.
- The BoardConsume allows different classes to interact and create boards
- The level reader class reads and validates level files
- The MainFrame class is the entry point for the program and provides the main menu and the interactions between classes at the top level

We produced an initial class design (See Appenix A). This was revised over time until we reached our final program structure (See Appendix B)

Board Class

We mainly considered two ways of representing the game board:

- The first was using a 2D array of boolean or integer values within some larger class. This approach would allow coloured buttons to be easily implemented but seemed less object oriented and would require a separate list of UI Button instances to be held.

- The second was storing a 2D array of Button instances. These would contain both the state of any individual button, and its UI representation. This is the solution we decided to use, as it was more object oriented and would allow us to encapsulate more button related logic into a single class.

Most of the board related logic is within the board class. We aimed to make it as generic as possible – so it could easily be extended and used in other applications. This allowed us to efficiently represent both a normal and editable board in a single class, which could be easily extended it to the coloured variation.

We opted to make use of a BoardConsumer interface as part of this goal, as it allowed us to create and have boards interact with different parent classes – as it forced these classes to have some common methods it could use.

The board features multiple constructors to suit different use-cases such as the board editor mode which starts with a board with all lights off. If a board consumer is provided, the board is added to it's UI. One is not always provided, for example in cases where it is being tested for solvability the board is cloned and no consumer is provided.

The board provides several public methods to streamline interaction with the board, such as getButton which retrieves a button from a given row/column position.

We added a print method which was used during testing to ensure the internal representation of the board matched the one shown in the user interface.

Game Class

The game class implements the board consumer interface. The game class can represent one 'play' of multiple levels by updating its internally held board.

This game class is responsible for keeping track of moves made, time elapsed and for using the LevelReader lass to read each level from file. We considered doing this file operation with the board class but decided to do it within the game class so the board could be used for both edited and loaded games.

The game class also features several constructors to suit the different use-cases and can have an initial board to display passed to it during initialisation. This is used after a board has been edited, so that board can be played. This class also takes a parent JPanel object which it appends its UI instances to. We decided to use composition instead of inheritance here as we were not overriding any of the methods within the JPanel class. The game class also customises its UI instances to make them look better.

The game class displays which level is currently being played. We chose not to require the user to complete levels in order as they may wish to play up to a certain level and then return to it a later date – and having to replay those levels would be a poor user experience.

The start method starts the timer resets the board's internal state for the start of a new level. This s combined with the setWon method which loads in the next level upon level completion.

Victory is determined to be when all lights in the board are out, via. The board.isSolved method – this is run every time a click is made.

Board Editor Class

This class is largely the same as the game class but with some different display options and has the update siblings set to false for its board so that buttons can be individually edited.

It also implements BoardConsumer so it has the required methods to interact with the board – it's constructor and general structure is very similar to that of the game class. We considered using inheritance to reduce code duplication, but we felt this was not suitable as a BoardEditor is not a specialisation of game and there was no appropriate way to share this code.

This class includes an isSolvable method to determine if the board is solvable before the user attempts to play it. To implement the solvability algorithm, we considered using either a mathematical approach or implementing a version of the commonly used 'Chase the lights' algorithm which is used by players to solve the game.

The is solvable method does the chasing the lights method of solving a lights out puzzle but first it clones the board so as to not change the users board and chases the lights down by hitting the button below every lit button. Then if certain configurations of the bottom row are left then it is solvable otherwise return false.

We decided to implement the 'Chase the lights' algorithm as the mathematical approach appeared complex and would require the use of converting our class-based approach to a 2D array of integers which would be used as a matrix. We made use of the LightsOut tutorial (Metzler, 2006) "Pattern lookup" to determine solvability by first chasing the lights and then checking the final row.

This method is called on every board update and the game can only be started if it is solvable.

We used a lambda function to trigger the MainFrame switching to the game as it allows for this state to easily be passed up without adding an additional method to the MainFrame class.

Button Abstract Class

The button abstract class contains common behaviour between the NormalButton and ColouredButton. We originally made this an interface before moving common logic to this abstract class. The sub-classes NormalButton and ColouredButton then implement features to store and update their own internal state (and colour) but implement some methods which are used by the board to determine if it is solved (such as isOn) and the methods update and activate which are used to either update a single button's state or update the state of a single button and those surrounding it.

This class also implements ActionListener and calls the activate method whenever it is clicked – this method is defined by the subclasses.

The normal button's state is either on or off, so it's activate method reflects this and only toggles it's state and colour between 'on' (green) and 'off' (gray).

The coloured button acts in the same way using the activate and update methods, however instead of switching between two states it switches between several different colours using a colours array. It also features a setState method which validates the new colour value.

Making use of this abstract class with certain methods overridden allowed us to maintain encapsulation and an object-oriented style while keeping duplicated code to a minimum.

Colored Game Class

This class extends the standard game board and acts in much the same way, except it has methods overridden to allow it to read the current level files and indicate that it is a coloured board within the level display.

Level Reader Class

The level reader file includes two extra classes – an exception and a level class which stores a 2-dimensional array of buttons and accommodates the time limit. The time limit is determined to be optional, but if a time limit is not specified and the time limit variant is enabled a level cannot be played.

The main level reader class has two methods for reading in different variant boards that work in much the same way. A 2D array of buttons is created, then a scanner object is instantiated that scans the level file in. An integer is made that is used as an index for the while loop (specifically for declaring the button objects) that loops through each line in the file then splits it into columns. Non-numerical values are removed, and then valid values are converted into button instances.

The method for reading in coloured level is the same except it uses coloured button objects and the current button state is an int. And depending on if the file reads the data as 1, 2 or 3 it will set the current button state to 1, 2 or 3, respectively.

Both methods perform validation to ensure that only valid values are present and that the correct number of rows/columns are present.

Main Frame Class

The main frame is the entry point for the program. It creates the main menu interface and manages the creation and clean-up of subclasses such as Game or BoardEditor sessions.

We moved several values such as dimensions to static variables so that they could be shared and re-used – such as button dimensions.

This class also contains some theme colours that are used throughout the program.

It is the only class that remains throughout the lifetime of the program and stores information such as which variants are currently enabled, and which game type is currently running.

For the main menu panel, a BoxLayout is used so that the components of the main menu can be stacked on top of each other in a centred column.

There are two buttons, one which starts a game with the current selected settings and one which takes the user to the board editor. Depending on the settings, the start button will call the game or colored game constructor to create a new game. It will then set the main menu panel to invisible, this is so that it can be set to visible later if the user wishes to go back to the main menu. The board editor button will call the board editor constructor and also hide the main menu.

There is also a back button that is created in the main menu so that it can access every component in the main frame. It is set to be invisible at first as it is not needed at the main menu. When a game or board editor is created, the back button is set to be visible so that the user can return to the main menu. When clicked, the back button will remove all the components of either the game or the board editor and set the main menu panel to be visible again.

Box.createVerticalGlue() is used to create gaps between components in the main menu so that the UI looks nicer. These gaps can increase or decrease depending on the window size so the window can be resized with no issues.

Bugs/Errors

The program does not have any major errors that stops it running or anything. It works as intended. But it has a few minor issues:

- The level number for the custom level displays as Level –1
- Occasionally when tabbing out/clicking off the game the win message for levels disappears and the user must hit a button to activate lights then turn them all off again to fix it. They can still hit the back button, however.
- The timer for the levels continue even during the win screen.

Examples of Program Running

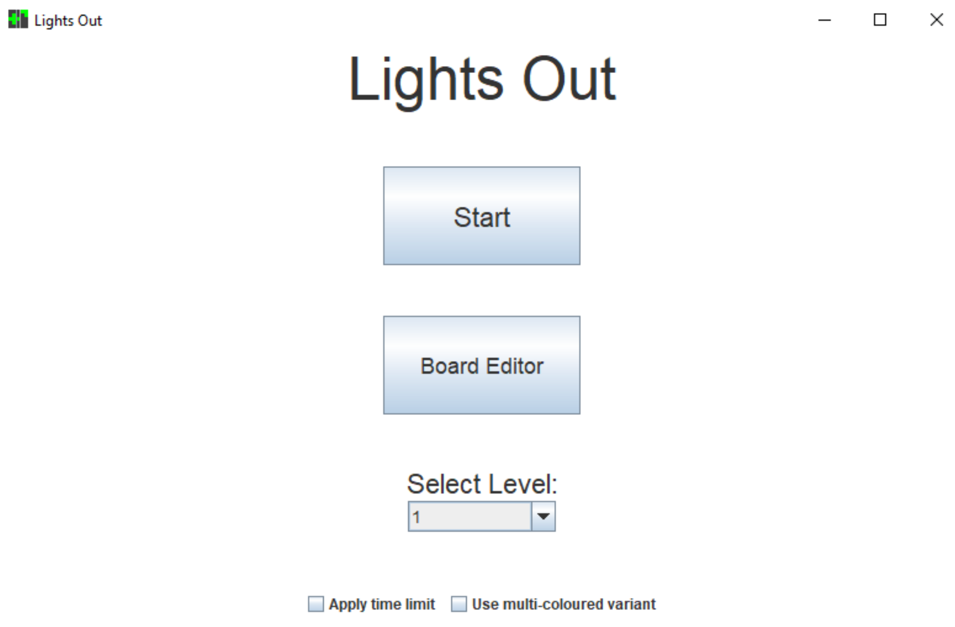


Figure 1 – The main menu when first loading the game.

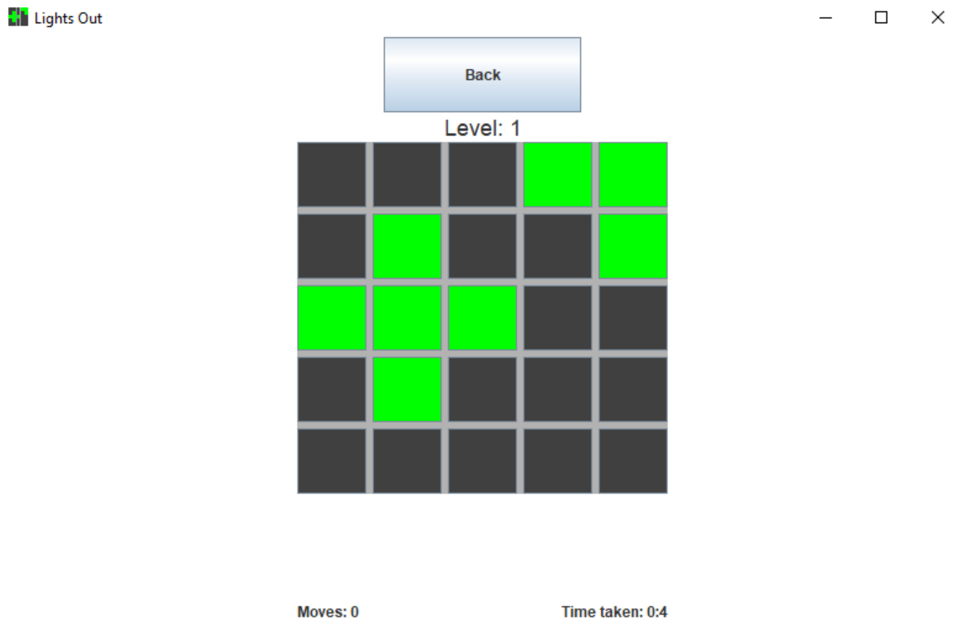


Figure 2 – The first level of the main mode is loaded after hitting the start button.

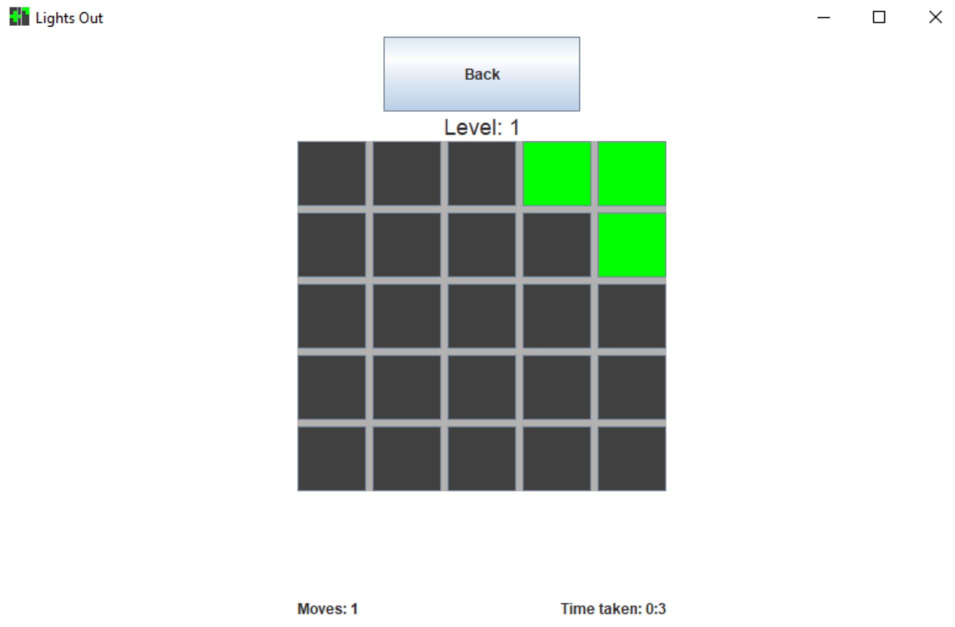


Figure 3 – After hitting a button (specifically B3) the board has updated

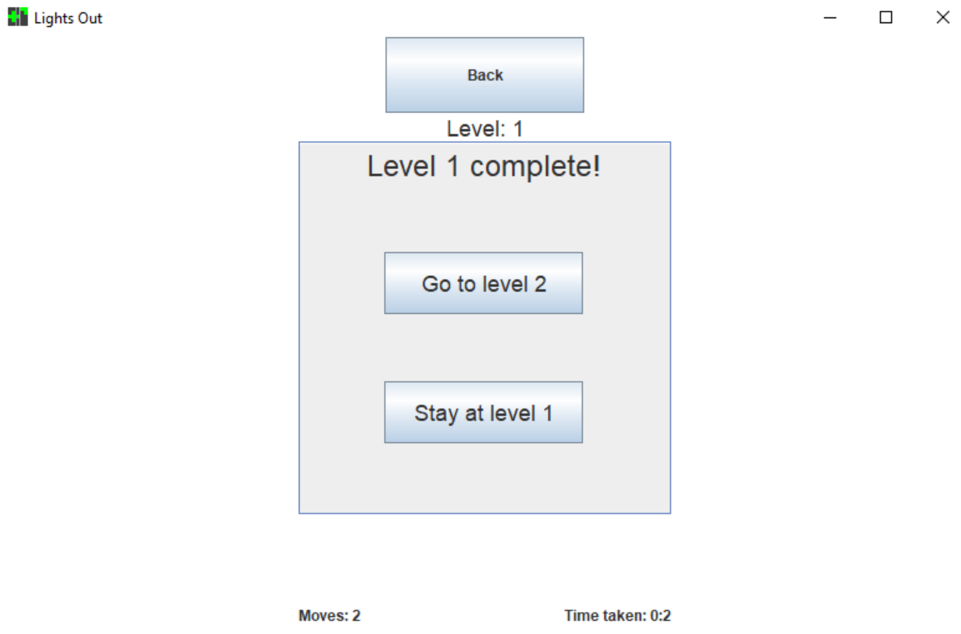
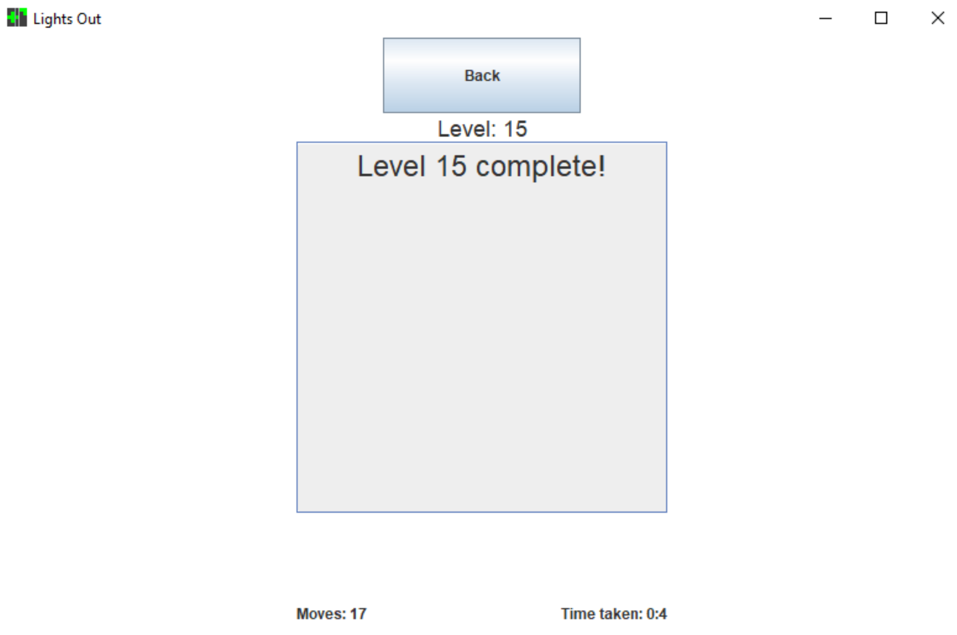


Figure 4 – After hitting the button E1 the win message was displayed.



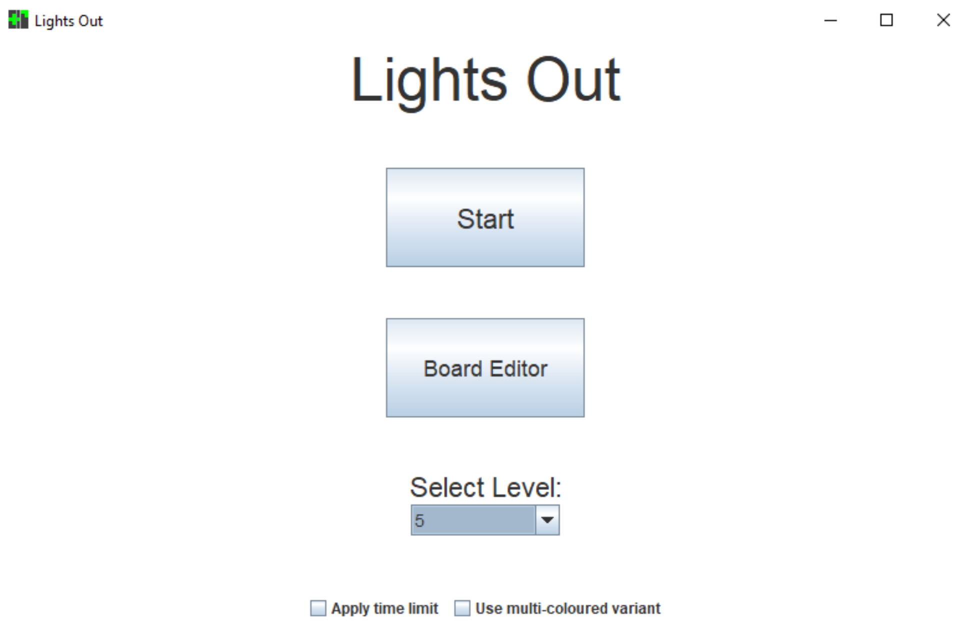
Level: 15

Level 15 complete!

Moves: 17

Time taken: 0:4

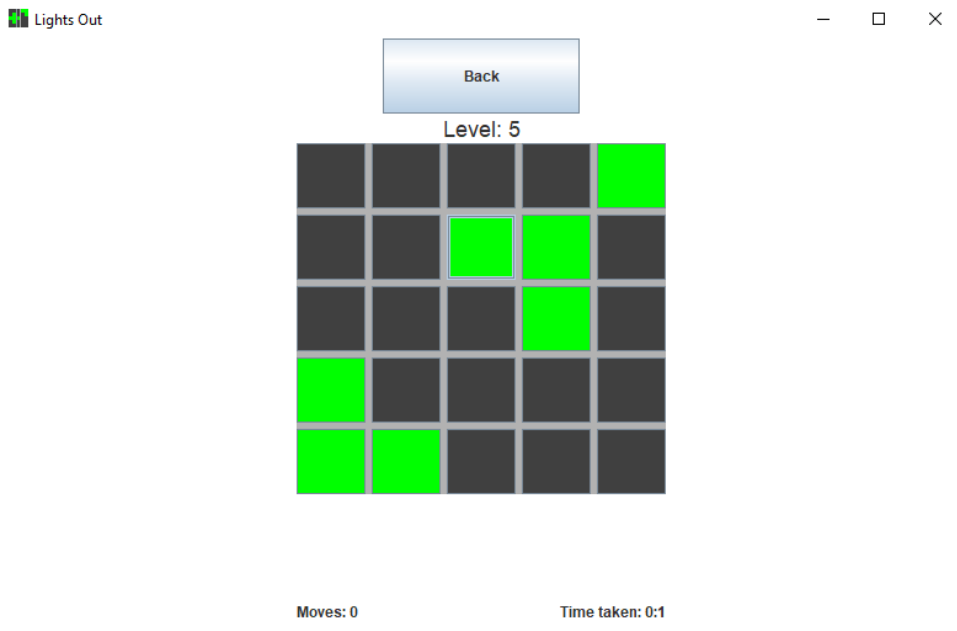
Figure 5 – The win message displays after beating the final level.



Select Level: 5

☐ Apply time limit ☐ Use multi-coloured variant

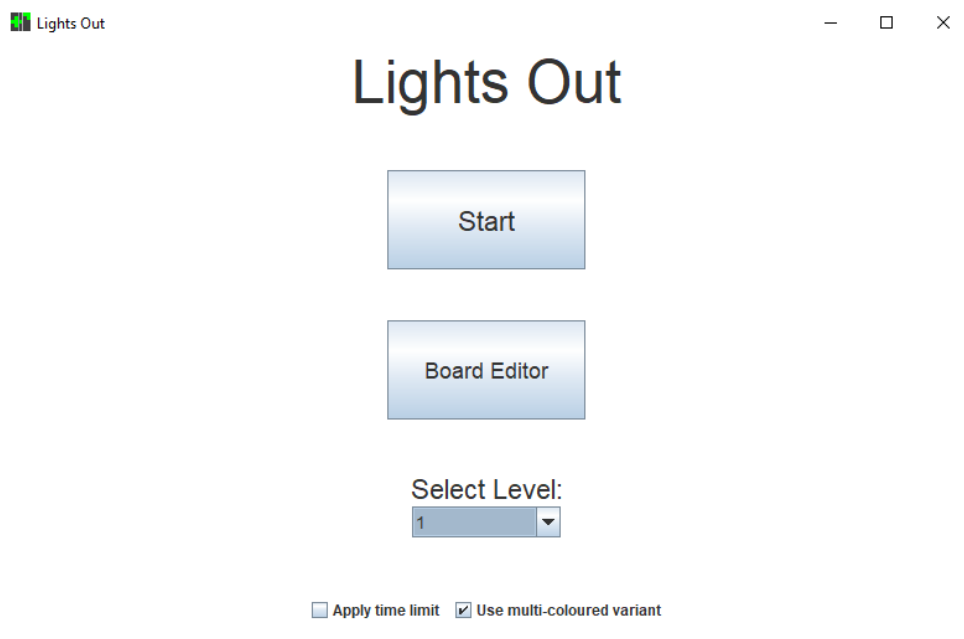
Figure 6 – The main menu with the level select at 5.



Level: 5

Moves: 0 Time taken: 0:1

Figure 7 – After hitting start with the 5th level selected the 5th level loaded.



Apply time limit [✓ Use multi-coloured variant]

Figure 8 – The main menu after checking the coloured mode.

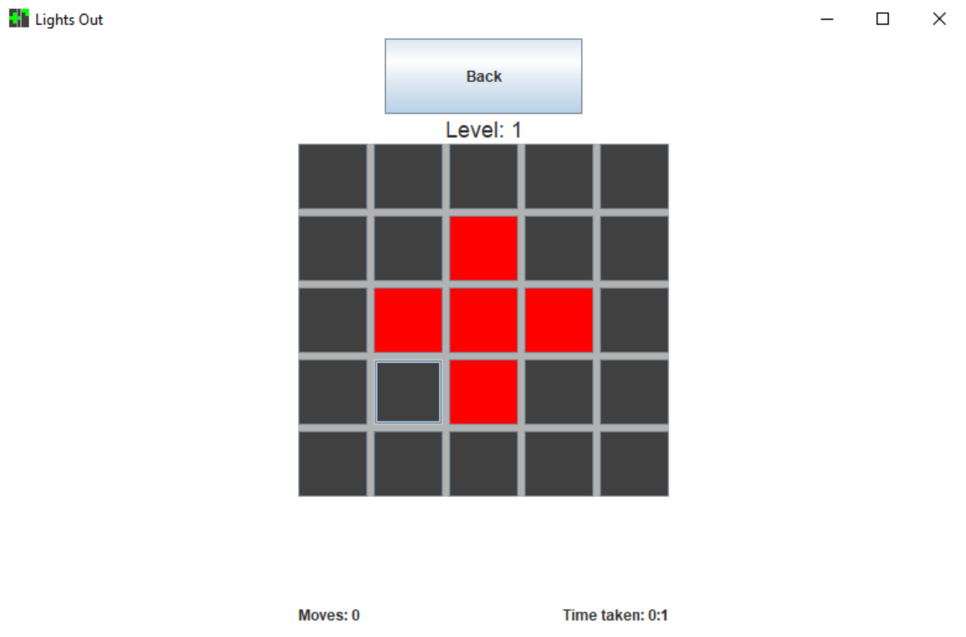


Figure 9 – The first coloured level has loaded after checking the coloured mode.

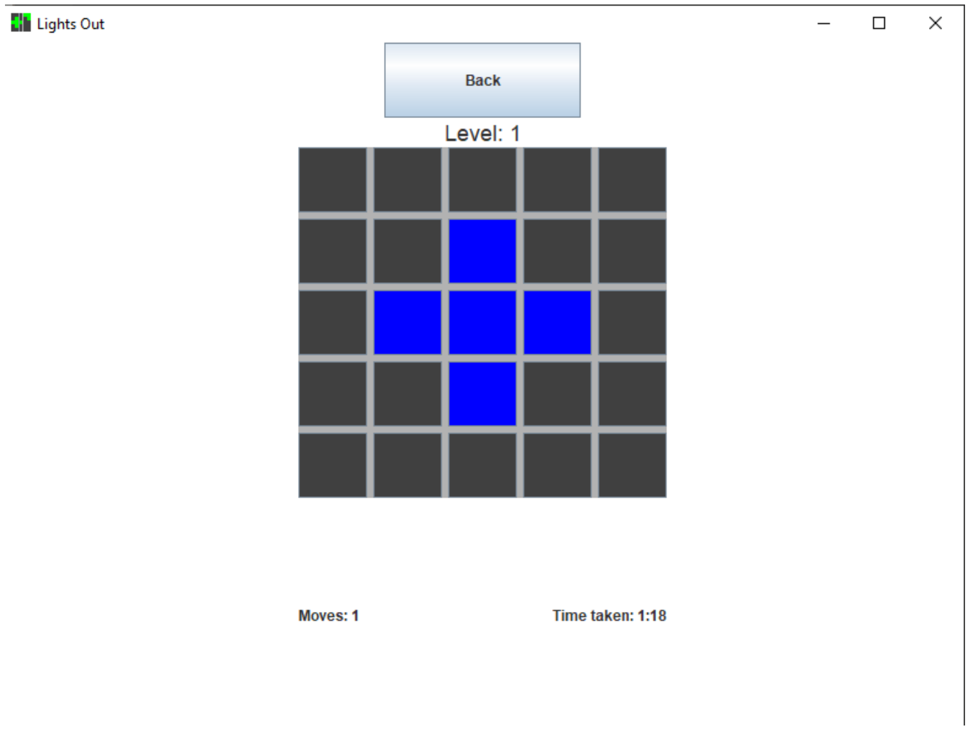


Figure 10 – The button C3 has been hit cycling the adjacent buttons to blue.

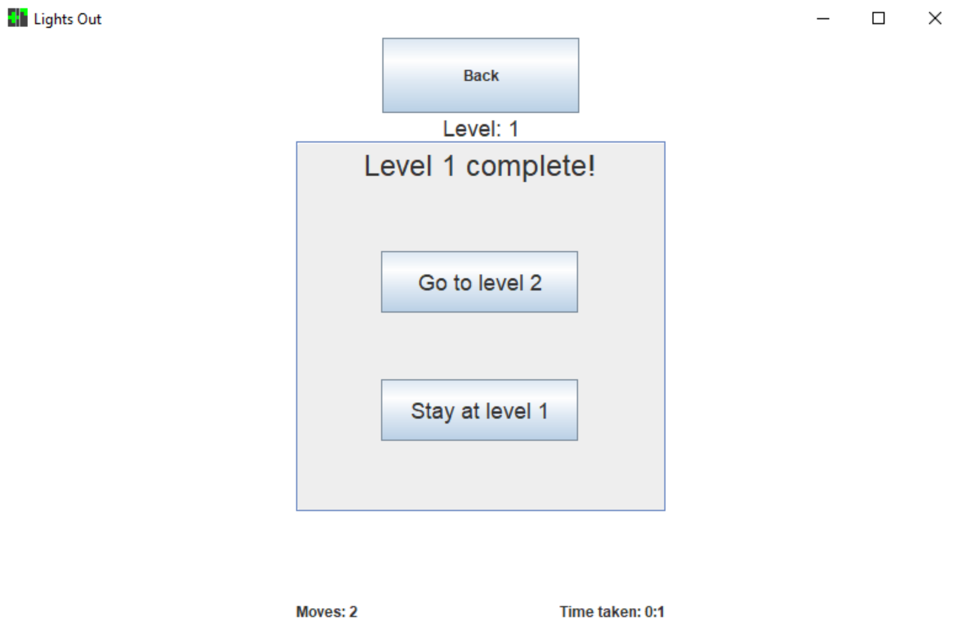


Figure 11 – After hitting the button C3 again the win message is displayed.

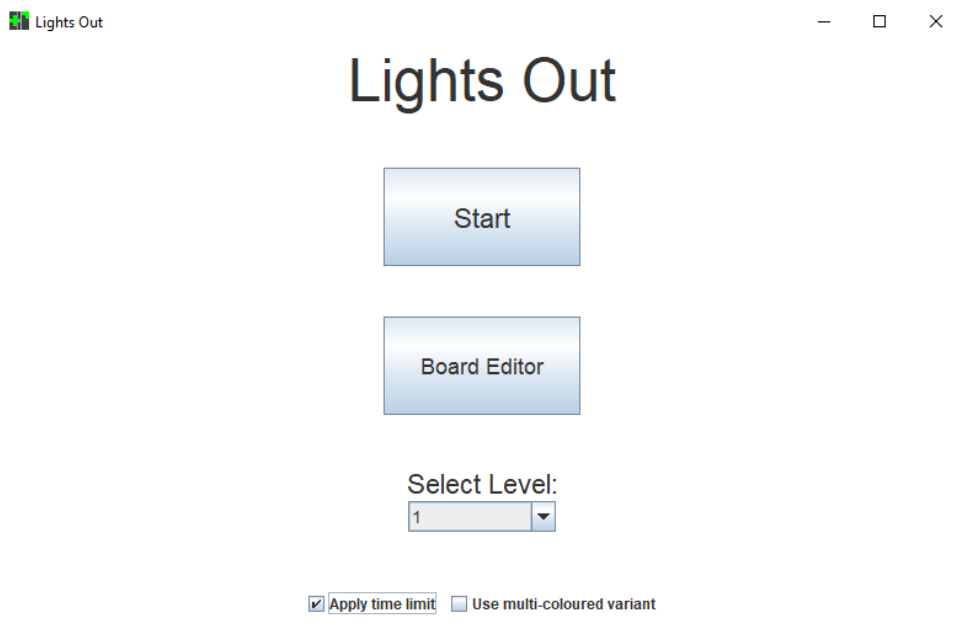
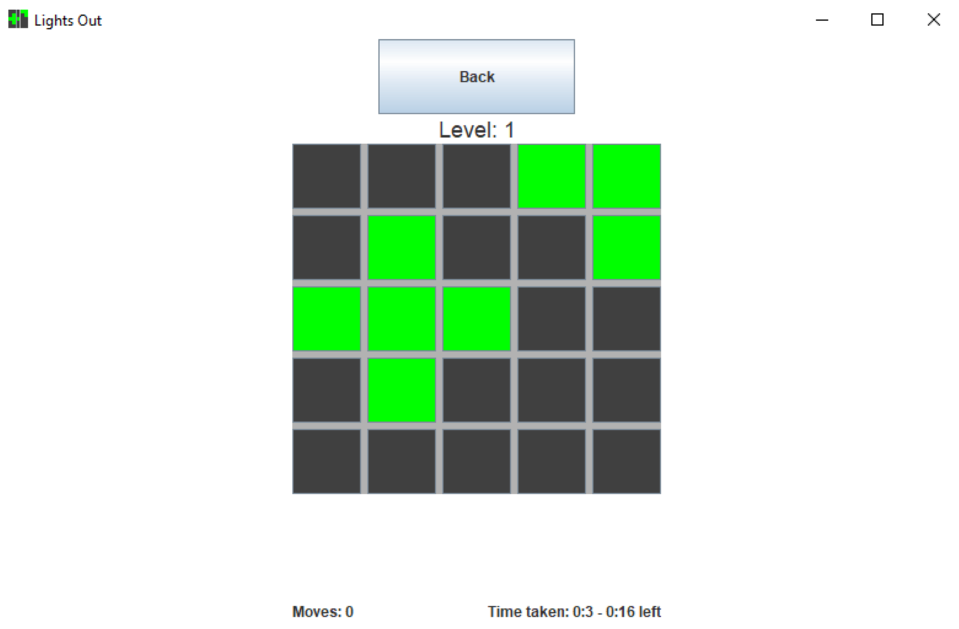


Figure 12 – The main menu before loading the 1st level with the time limit applied.



Moves: 0 Time taken: 0:3 - 0:16 left

Figure 13 – Hitting the start button shows that the Level now has a time limit counting down in the corner.

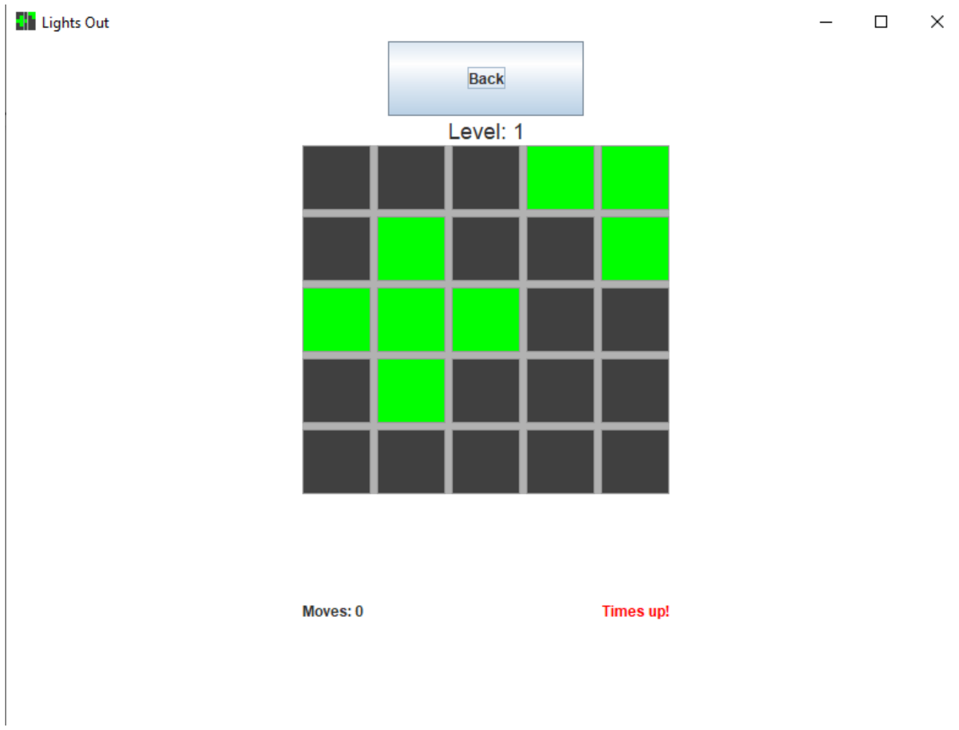


Figure 14 – the time now has run out and the user cannot click any of the board buttons and must go back to the main menu.

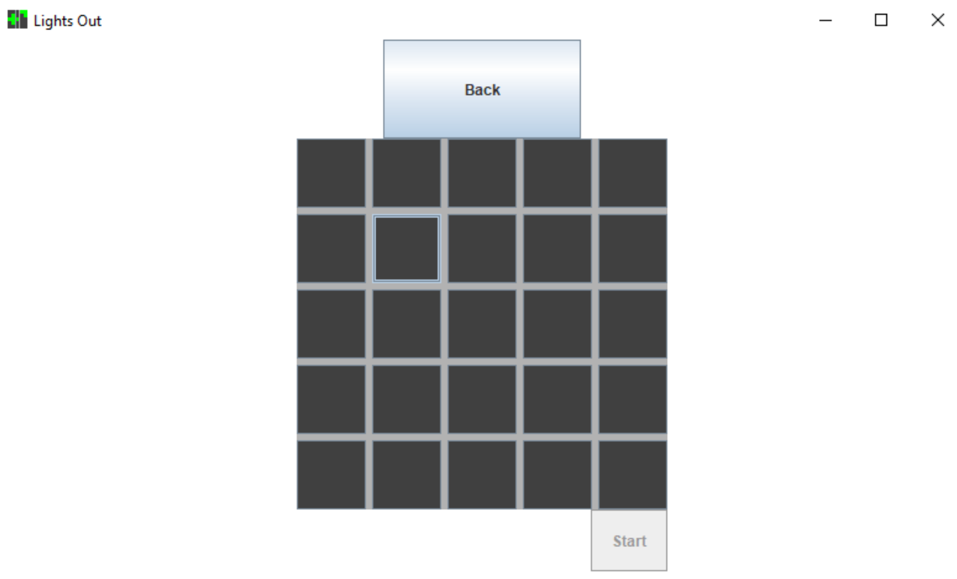


Figure 15 – After hitting Board editor it shows the blank editable board with the start button greyed out as no level has been made yet.

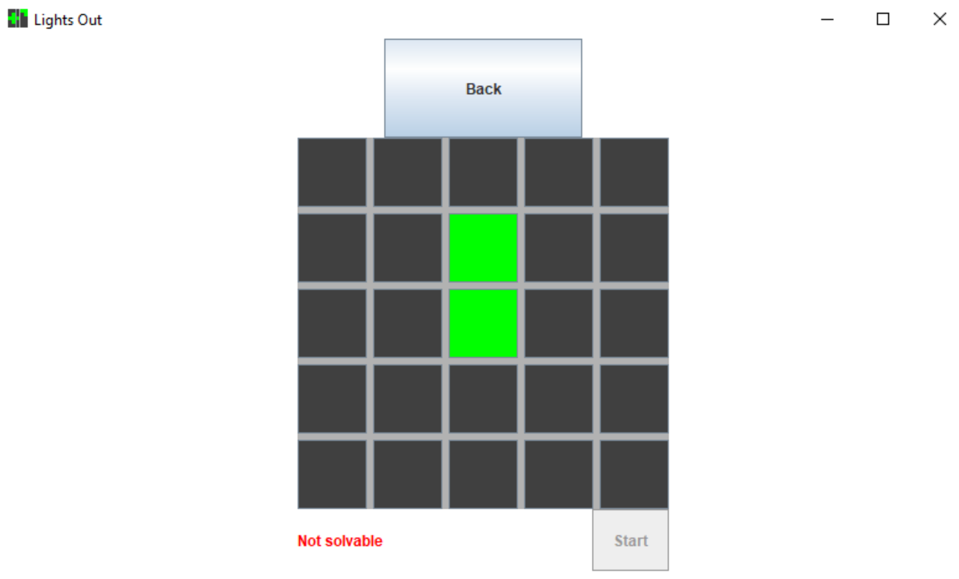


Figure 16 – After hitting buttons C2 and C3 the level is not solvable, so the start button is still greyed out.

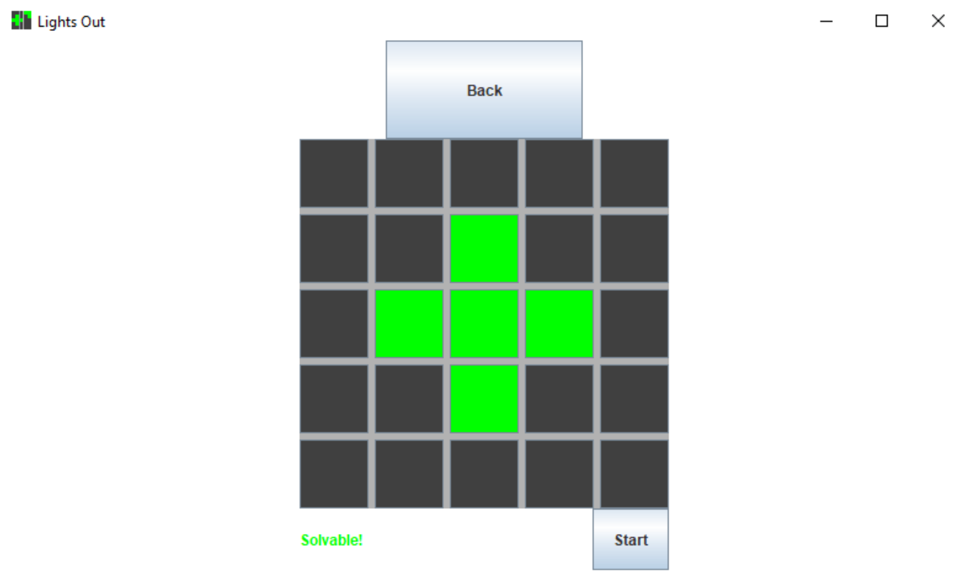


Figure 17 – After making a solvable pattern the user can now hit the start button.

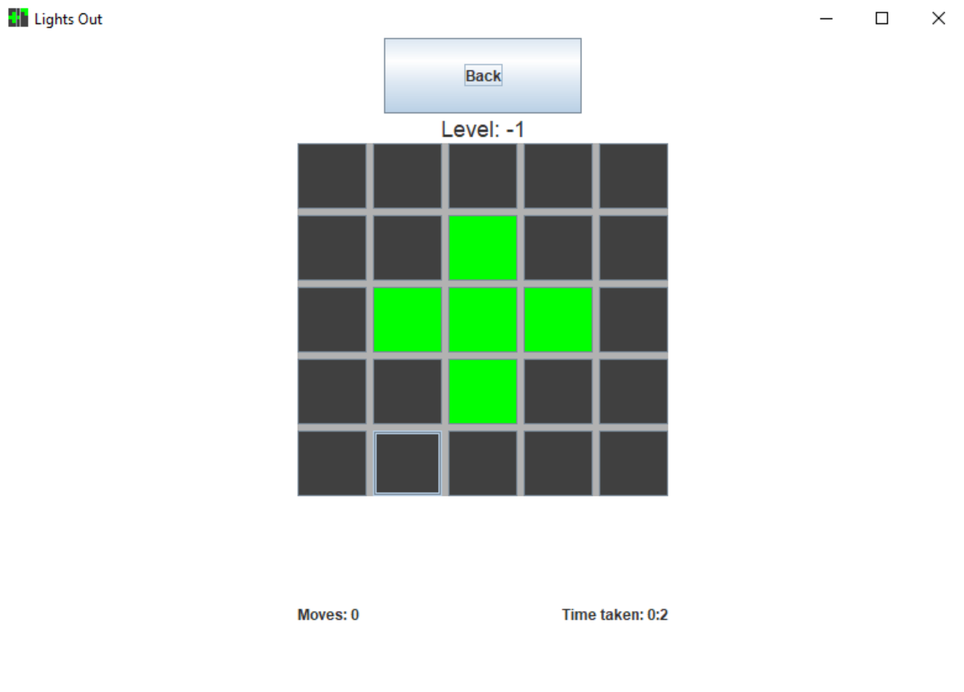


Figure 18 – The subsequent level has now been loaded and can be solved.

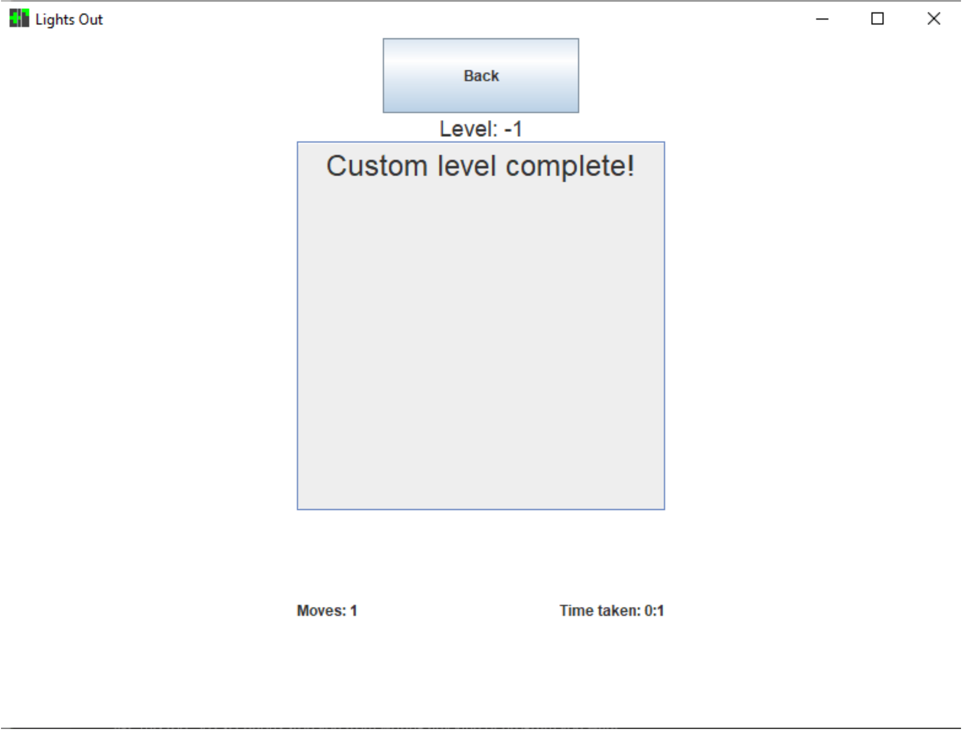


Figure 19 – The subsequent win message when the level is solved.

References

Vidu (2014) Java – How to set icon image for swing. Available at: https://stackoverflow.com/questions/7194734/how-to-set-icon-image-for-swing-application (Accessed: 5 April 2021).

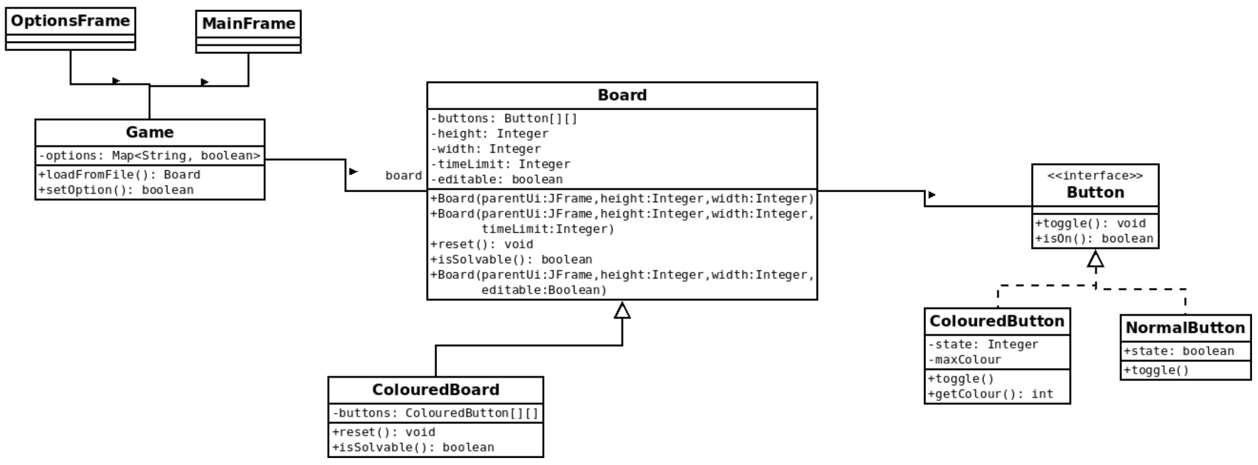
Javapoint.com (n.d.) How to Read CSV file in java. Available at: https://www.javatpoint.com/how-to-read-csv-file-in-java (Accessed: 20 March 2021).

Metzler, J (2006) LightsOut Tutorial. Available at: https://www.logicgamesonline.com/lightsout/tutorial.html (Accessed: 25 March 2021).

Mercurial Link: https://rdn1.hg.cs.st-andrews.ac.uk/LightsOut

Appendices

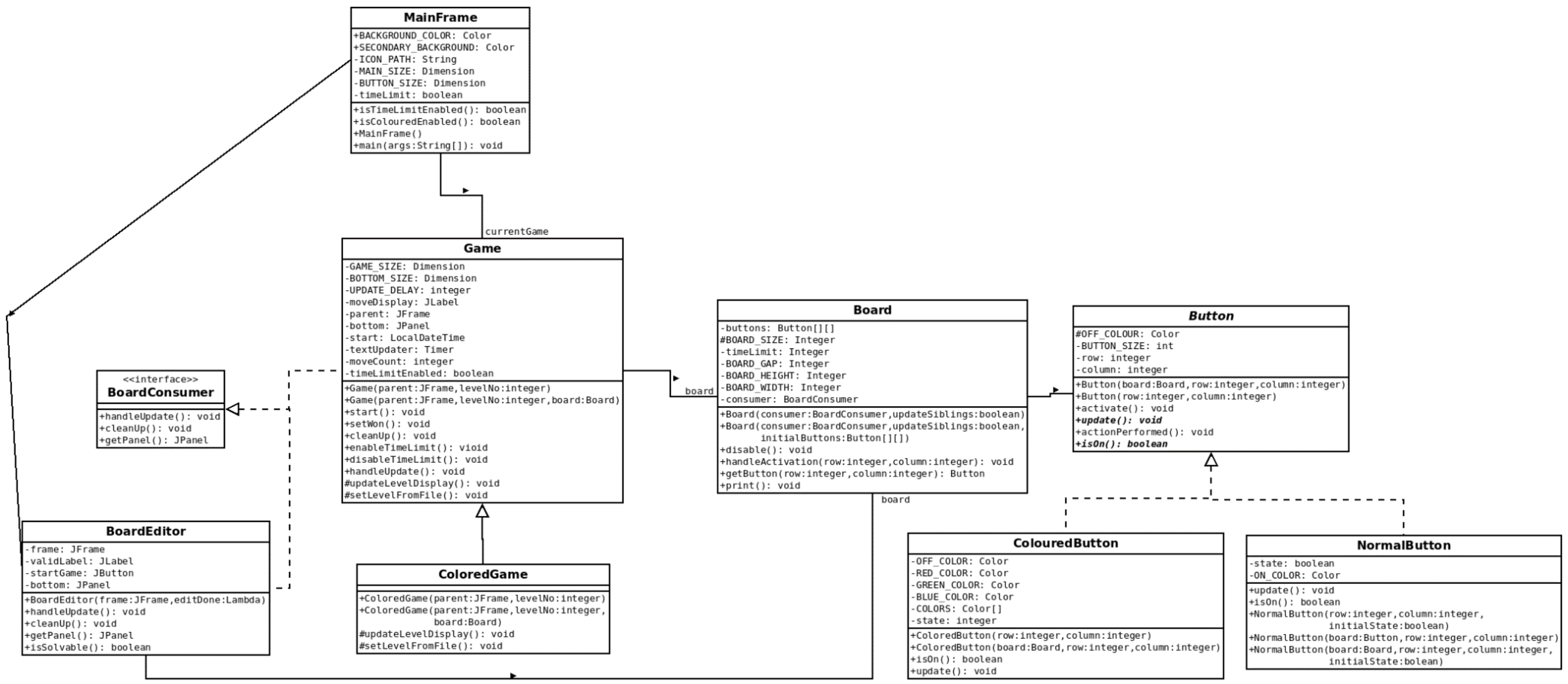
Appendix A – Initial program design



Appendix B – Final program design

CS1006 Lights Out Practical 200014517/200020128/200015381 Tutor: Ognjen Arandelovic

04 Apr 2021


MainFrame
+BACKGROUND_COLOR: Color
+SECONDARY_BACKGROUND: Color
-ICON_PATH: String
-MAIN_SIZE: Dimension
-BUTTON_SIZE: Dimension
-timeLimit: boolean
+isTimeLimitEnabled(): boolean
+isColouredEnabled(): boolean
+MainFrame()
+main(args:String[]): void
currentGame
Game
-GAME_SIZE: Dimension
-BOTTOM_SIZE: Dimension
-UPDATE_DELAY: integer
-moveDisplay: JLabel
-parent: JFrame
-bottom: JPanel
-start: LocalDateTime
-textUpdater: Timer
-moveCount: integer
-timeLimitEnabled: boolean
+Game(parent:JFrame,levelNo:integer)
+Game(parent:JFrame,levelNo:integer,board:Board)
+start(): void
+setWon(): void
+cleanUp(): void
+enableTimeLimit(): vioid
+disableTimeLimit(): void
+handleUpdate(): void
#updateLevelDisplay(): void
#setLevelFromFile(): void
board
Board
-buttons: Button[][]
#BOARD_SIZE: Integer
-timeLimit: Integer
-BOARD_GAP: Integer
-BOARD_HEIGHT: Integer
-BOARD_WIDTH: Integer
-consumer: BoardConsumer
+Board(consumer:BoardConsumer,updateSiblings:boolean)
+Board(consumer:BoardConsumer,updateSiblings:boolean, initialButtons:Button[][])
+disable(): void
+handleActivation(row:integer,column:integer): void
+getButton(row:integer,column:integer): Button
+print(): void
Button
#OFF_COLOUR: Color
-BUTTON_SIZE: int
-row: integer
-column: integer
+Button(board:Board,row:integer,column:integer)
+Button(row:integer,column:integer)
+activate(): void
+update(): void
+actionPerformed(): void
+isOn(): boolean
<<interface>>
BoardConsumer
+handleUpdate(): void
+cleanUp(): void
+getPanel(): JPanel
BoardEditor
-frame: JFrame
-validLabel: JLabel
-startGame: JButton
-bottom: JPanel
+BoardEditor(frame:JFrame,editDone:Lambda)
+handleUpdate(): void
+cleanUp(): void
+getPanel(): JPanel
+isSolvable(): boolean
ColoredGame
+ColoredGame(parent:JFrame,levelNo:integer)
+ColoredGame(parent:JFrame,levelNo:integer, board:Board)
#updateLevelDisplay(): void
#setLevelFromFile(): void
board
ColouredButton
-OFF_COLOR: Color
-RED_COLOR: Color
-GREEN_COLOR: Color
-BLUE_COLOR: Color
-COLORS: Color[]
-state: integer
+ColoredButton(row:integer,column:integer)
+ColoredButton(board:Board,row:integer,column:integer)
+isOn(): boolean
+update(): void
NormalButton
-state: boolean
-ON_COLOR: Color
+update(): void
+isOn(): boolean
+NormalButton(row:integer,column:integer, initialState:boolean)
+NormalButton(board:Button,row:integer,column:integer)
+NormalButton(board:Board,row:integer,column:integer, initialState:bolean)