



## Assignment of bachelor's thesis

<b>Title:</b>	Logik desktop application
<b>Student:</b>	Kirill Rybakov
<b>Supervisor:</b>	prof. Ing. Pavel Tvrdík, CSc.
<b>Study program:</b>	Informatics
<b>Branch / specialization:</b>	Software Engineering 2021
<b>Department:</b>	Department of Software Engineering
<b>Validity:</b>	until the end of summer semester 2024/2025

### Instructions

Due to its general worldwide popularity, the Mastermind/Logik puzzle game has also had many implementations as a desktop or web application. However, most of them are not flexible enough in terms of user configurability (choice of the number of positions, number of colours, colours themselves, combinations without or with colour repetition).

Make a review of existing desktop applications for the Mastermind/Logic game in which the human user guesses hidden combinations generated by a computer.

Design and implement a platform-independent desktop Mastermind/Logik game with a graphical user interface that allows the user to configure all game parameters: the number of positions, the number of colours and their selection from a palette, possibility of combinations without or with colour repetition. Consult the details of the user interface with the supervisor. Document properly the resulting code. Perform extensive testing of the application.

Bachelor's thesis

# DESKTOP APPLICATION LOGIC (MASTERMIND)

Kirill Rybakov

Faculty of Information Technology  
Faculty of Software Engineering  
Supervisor: prof. Ing. Pavel Tvrđík, CSc.  
May 16, 2024

Czech Technical University in Prague  
Faculty of Information Technology

© 2024 Kirill Rybakov. All rights reserved.

*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

Citation of this thesis: Rybakov Kirill. *Desktop application Logic (Mastermind)*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

# Contents

<b>Acknowledgments</b>	<b>v</b>
<b>Declaration</b>	<b>vi</b>
<b>Abstract</b>	<b>vii</b>
<b>List of abbreviations</b>	<b>viii</b>
<b>1 Analysis</b>	<b>1</b>
1.1 Mastermind . . . . .	1
1.2 Limitations . . . . .	1
1.3 Existing implementations . . . . .	1
1.3.1 Example 1 . . . . .	2
1.3.2 Example 2 . . . . .	3
1.3.3 Conclusions drawn from existing implementations . . . . .	4
1.4 Functional requirements . . . . .	5
1.5 Non-functional requirements . . . . .	7
1.6 Use case analysis . . . . .	8
1.6.1 Use case diagram . . . . .	8
1.6.2 Actors . . . . .	8
1.6.3 UC1: Start game . . . . .	8
1.6.4 UC2: View settings . . . . .	9
1.6.5 UC3: Configure holders . . . . .	9
1.6.6 UC4: Configure attempts . . . . .	9
1.6.7 UC5: Configure repetitions . . . . .	9
1.6.8 UC6: Add a peg colour . . . . .	10
1.6.9 UC7: Delete a peg colour . . . . .	10
1.6.10 UC8: Save or discard settings . . . . .	10
1.6.11 UC9: Check row . . . . .	11
<b>2 Design</b>	<b>12</b>
2.1 Technologies . . . . .	12
2.1.1 Candidates . . . . .	12
2.1.2 Chosen technologies . . . . .	13
2.2 Designing the UI . . . . .	13
2.2.1 The Main Menu scene . . . . .	13
2.2.2 The Game scene . . . . .	14
2.2.3 The Settings scene . . . . .	14
2.3 Project structure . . . . .	15
2.4 Class diagram . . . . .	16
2.4.1 The Gameplay scene . . . . .	17
2.4.2 The Settings scene . . . . .	19
2.4.3 The Menu scene . . . . .	21
2.4.4 The Commons folder . . . . .	22

<b>3</b>	<b>Implementation</b>	<b>23</b>
3.1	Clicking and dragging pegs into place . . . . .	23
3.2	Setting the colour of the pegs . . . . .	24
3.3	Generation of combinations . . . . .	27
3.4	Checking the Player's attempt . . . . .	28
3.5	Zooming in and out in the Game scene . . . . .	30
3.6	Build and Deployment . . . . .	30
3.7	Testing . . . . .	31
	3.7.1 Manual testing of the UI . . . . .	31
	3.7.2 Integration and System tests . . . . .	32
	3.7.3 Unit tests . . . . .	32
3.8	Used third-party libraries and assets . . . . .	33
<b>4</b>	<b>Conclusion</b>	<b>34</b>
	<b>Contents of the attached .zip file</b>	<b>37</b>

## List of Figures

1.1	Existing implementation at <a href="https://webgamesonline.com/mastermind/">https://webgamesonline.com/mastermind/</a> . . . . .	2
1.2	Existing implementation at <a href="https://www.archimedes-lab.org/mastermind.html">https://www.archimedes-lab.org/mastermind.html</a> . . . . .	3
1.3	Colourblind-friendly version of <a href="https://www.archimedes-lab.org/mastermindbis2.html">https://www.archimedes-lab.org/mastermindbis2.html</a> . . . . .	4
1.4	Diagram of the use cases described in this section . . . . .	8
2.1	Sample Main Menu GUI design . . . . .	14
2.2	Sample Game scene GUI design . . . . .	14
2.3	Sample Settings Menu GUI design . . . . .	15
2.4	Class diagram of the Gameplay scene . . . . .	17
2.5	Class diagram of the Settings scene . . . . .	19
2.6	Class diagram of the Menu scene . . . . .	21
2.7	Class diagram of the Commons folder . . . . .	22
3.1	Samples of the colours side-by-side taken from <a href="https://coolors.co/">https://coolors.co/</a> . . . . .	25
3.2	The redesigned Settings menu . . . . .	26
3.3	The Game scene with a large number of attempts at the minimum zoom level. . . . .	30

## List of code listings

3.1	The logic for picking up a peg . . . . .	23
3.2	The logic for dropping a peg . . . . .	24
3.3	The Generator class drag logic . . . . .	24
3.4	The colour highlighting function . . . . .	26
3.5	The data structure used in the combination generation algorithm . . . . .	27
3.6	The combination generation algorithm . . . . .	28
3.7	The data structure used in the checking algorithm . . . . .	29
3.8	The main logic of the first stage of the checking algorithm . . . . .	29
3.9	The main logic of the second stage of the checking algorithm . . . . .	29
3.10	The <code>create_release</code> job, a part of the publish stage . . . . .	31
3.11	Checking that the mouse button used to drag an object is the left mouse button . . . . .	32
3.12	Interface for the attempt checker . . . . .	32
3.13	Interface for the combination generator . . . . .	33

*I would like to thank my supervisor, prof. Ing. Pavel Tvrđík, CSc., for providing his support and advice in the writing of this thesis. I would also like to thank Ing. Zdeněk Rybala, Ph.D. for his advice on modelling the application.*

## Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 16, 2024



## Abstract

The aim of this thesis is the development of the desktop application game "Mastermind" (also referred to as "Logic") for Windows and Linux. The resulting program is intended to improve upon the existing implementations of the game, allowing to select colours from a predefined palette, and providing greater flexibility in the length of combination and the number of guesses and repetitions. The implementation is written in C# and uses the Unity Engine.

**Keywords** Mastermind, Desktop application, Multiplatform, Linux, Windows, C#, Unity, GUI


## Abstrakt

Cílem této práce je vývoj desktopové aplikační hry "Mastermind" (také označované jako "Logic") pro Windows a Linux. Výsledný program má vylepšit stávající implementace hry, umožňuje výběr barev z předdefinované palety a poskytuje větší flexibilitu v délce kombinace a počtu odhadů a opakování. Implementace je napsána v C# a používá Unity Engine.

**Klíčová slova** Mastermind, Desktopová aplikace, Multiplatformní, Linux, Windows, C#, Unity, GUI

## List of abbreviations

GUI	Graphical User Interface
UI	User Interface
CI	Continuous Integration
FIT	Faculty of Information Technology
JRE	Java Runtime Environment
OS	Operating System
API	Application Programming Interface
SDL	Simple DirectMedia Layer
CC	Creative Commons
SA	Share Alike
RGB	Red Green Blue
DLL	Dynamic Link Library



# Chapter 1

## Analysis

### 1.1 Mastermind

The game "Mastermind" is a board game between two Players. The first Player, known as the "code maker", thinks of a combination while the second Player plays the role of the "code breaker" and tries to find the combination until his attempt either matches the target combination, or the number of attempts reaches a certain limit. Each turn, the code maker checks the code breaker's attempt and informs him how closely it matched the target combination.

The board has a fixed number of rows, where each row has a fixed number of slots for the code breaker to insert coloured pegs. These rows would correspond to the code breaker's attempts and will be checked by the code maker. Consequently, if the code breaker runs out of rows to place coloured pegs before finding the combination, he has lost the game.

The rows also contain slots for the code maker to correct the code breaker's attempts. Most commonly, the pegs placed by the code maker are of two colours: red and white. Red means that one of the colours placed by the code breaker is in the correct position and white means that the colour placed by the code breaker is correct, but it is in a wrong position. No peg placed by the code maker means that the code breaker's colour is not present at all in the target combination. The code breaker will use this information to improve his next attempts.

### 1.2 Limitations

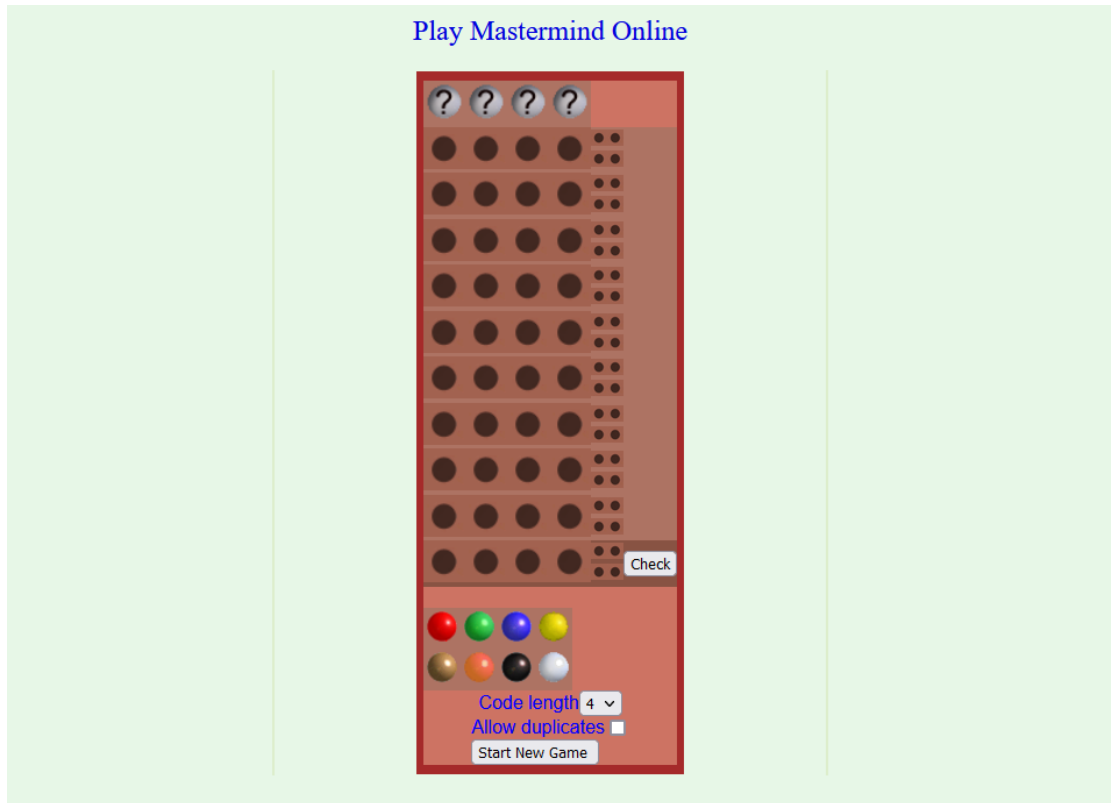
In this setup, the game is confined to the dimensions of the physical board: the number of attempts is limited by the number of rows in the board, the number of colours is limited by the tokens in the set and the length of the combination is limited by the number of slots. Furthermore, the need for a second Player leaves room for human error when checking the other Player's attempt and coming up with predictable combinations.

### 1.3 Existing implementations

Some digital versions of the game were developed. They remove the need for the second Player, thus solving one of the shortcomings of the board game. However, they still inherit some of the limitations of the physical board. Namely, the lack of customization and configurability.

### 1.3.1 Example 1

The following is a screenshot from one of the Mastermind implementations found while conducting research on existing implementations:



■ **Figure 1.1** Existing implementation at <https://webgamesonline.com/mastermind/>

This implementation follows the original layout of the board game, having the slots for the code breaker laid out in rows, with the coloured pegs at the bottom for the code breaker to click and place on the slots. The Player can set up some parameters of the game.

For example, the length of the combination can be chosen to be either 4 or 8. Setting the length to any intermediate value is impossible. Furthermore, the physical board game would provide more flexibility in this case because the two Players can agree to keep the combination of a certain length and not use the extra slots on the board. However, this implementation does not allow the Player to configure the number of attempts, it always remains constant. The fixed number of attempts becomes a problem for combinations of a large length, such as 8.

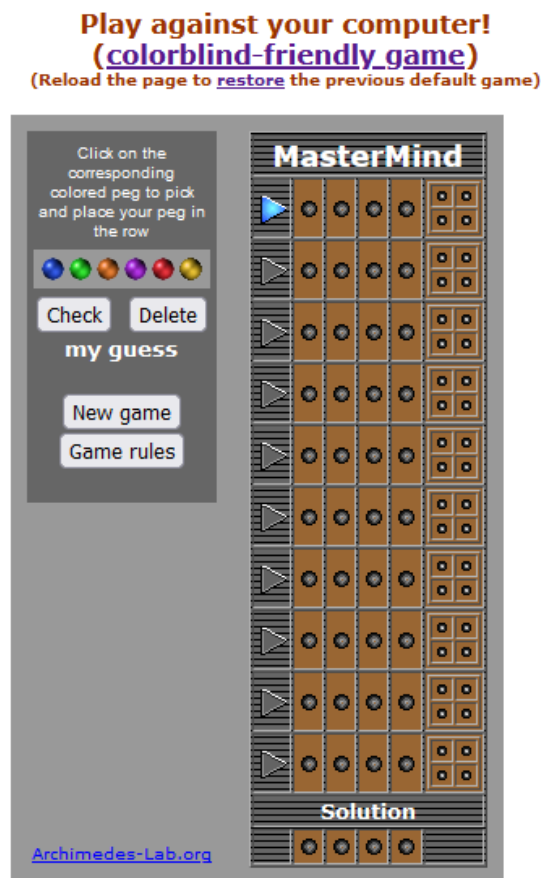
Another configuration option provided here is the "Allow duplicates" check box. If it is enabled, then the colours in the combination can repeat. This is also something that is present in the physical version of the game, as the two Players can agree with each other on the repetition of colours. However, this button does not allow the Player to specify at most how many times can a colour repeat, which is something that can again be done in the original game by agreement.

The last major limitation of this implementation is that it does not allow the Player to select which colours are included in the game. The palette is fixed and may cause problems for the Player, for example if he is colourblind. This is a limitation present in both the digital and physical versions of the game, because the colours are limited by the set of pegs provided with the board game.

This implementation also contains a sub-optimal UI which results in a negative user experience: the coloured pegs cannot be dragged into the slots. They must instead be clicked on, after which they will attach to the cursor and then the Player can click on the slots to place the peg. This is very unintuitive and there are no instructions on how to place the pegs. Furthermore, once a colour is selected, it cannot be "de-selected" and will follow the cursor forever. A different colour can be selected, but it is impossible for the Player to get back to the ordinary system cursor.

### 1.3.2 Example 2

This is a screenshot from another implementation of the game found on the web:



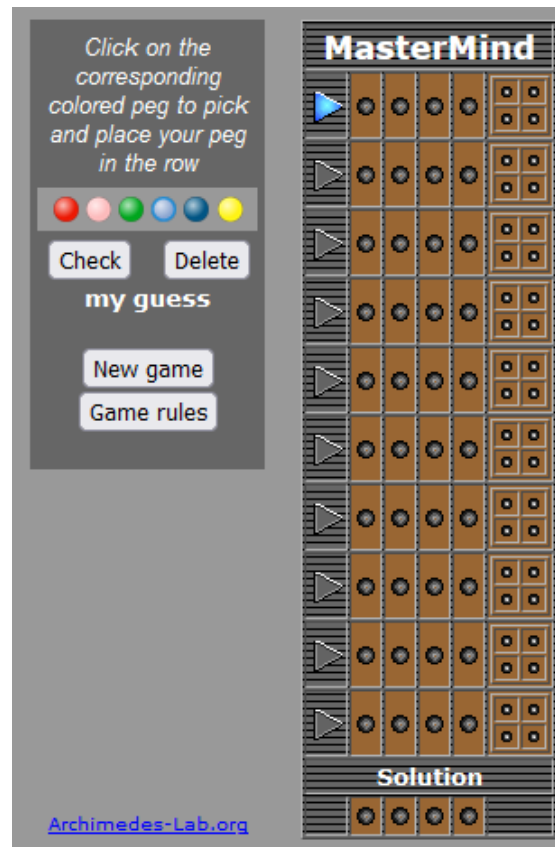
■ **Figure 1.2** Existing implementation at <https://www.archimedes-lab.org/mastermind.html>

This implementation provides minimal customization features. The length of the combination and number of attempts are fixed. Furthermore, it does not provide any information to the Player if the repetition of colours is allowed or not, which will make it more difficult for the Player to find the combination. If the Player does not want the difficulty of the game to be too high, this presents a big drawback for the implementation.

Furthermore, the GUI implementation is not great. In the previous example, the Player had some form of dictating in which slot will the selected colour be placed. In this example, the Player must click on the colour and it will be filled in from left to right. This means that the Player must always solve the combination from left to right, which may not always correspond to some Players' strategies. In addition to that, if a Player makes a mistake, there is no way

of removing a specific colour from his row. The only way to do so is to click on the "Delete" button, which will delete the entire row. This further worsens the user experience because the Player must be able to memorize their current input if he wishes to make a correction.

The minimal configurability consists in some possibility of customizing the colour set. This feature is present in the form of a colourblind-friendly version of the game and contains a slightly different colour set:



■ **Figure 1.3** Colourblind-friendly version of <https://www.archimedes-lab.org/mastermindbis2.html>

The colours here are slightly different, however the game still is not as flexible as desired.

### 1.3.3 Conclusions drawn from existing implementations

In order to improve on the existing implementations, the new implementation of Mastermind described in this thesis will need to include the following features:

- An option for the Player to set a reasonable number of attempts.
- The possibility for setting which exact colours are used for combinations.
- The possibility for adjusting the number of repetitions. The Player must be able to specify how many times a single colour is allowed to repeat.
- The possibility for setting the length of the combination. It must only be limited by the number of colours the Player has set. For example, if colour repetitions are not allowed then the length of the combination should not be greater than the number of colours.

- The large amount of customization calls for a way for the Player to save his settings for subsequent launches of the game. Thus, some form of persistence is required.
- Following from the previous point, in order to correctly implement persistence, the game must be implemented as a desktop application. Persistence can be achieved online, however this would require the Player to create an account, which is unnecessary for just one specific use case.
- Furthermore, since it is a desktop application, the application must be multiplatform. Because Windows and Linux are the most popular operating systems and the author does not have a suitable MacOS development environment, the application will only be developed for these operating systems.

The settings of the game will consist of a list of configuration values, for example the number of attempts, number of holders, number of repetitions and a list of colours.

## 1.4 Functional requirements

### The Main menu

#### F1.1 Starting the game

The Player should be able to start the game from the main menu with the currently saved settings.

#### F1.2 Accessing the Settings menu

The Player should be able to access the Settings menu from the Main menu to configure the game.

#### F1.3 Loading settings from file

The settings should be loaded from a file when the application is started, so that they take effect immediately when the Player clicks on the "Play" button.

### The Settings menu

#### F2.1 Exiting without saving

The Player should be able to exit the Settings menu without saving his changes. This should effectively rollback any changes made.

#### F2.2 Exiting and saving

If the Player is satisfied with his configuration, he may choose to save his changes and exit the Settings menu.

#### F2.3 Saving to file

The changes should be saved to a file to persist across program runs.

### **F2.5 Setting the number of slots (holders)**

The Player should be able to configure the number of slots (holders) for the coloured pegs, hence determining the length of the combination.

### **F2.6 Setting the number of attempts**

The Settings menu should contain some way for the Player to choose how many attempts he has at finding a combination.

### **F2.7 Setting the number of repetitions**

The Player can exactly specify at most how many times can a colour repeat. The default setting is one, which implies that there are no repetitions.

### **F2.8 Adding a new coloured peg**

The Player can add a new coloured peg which will be included in the generated combination.

### **F2.9 Editing an already created peg**

Some way of changing the colour of an existing peg should be possible.

### **F2.10 Validating input data**

The settings configured by the Player should be validated in order to prevent data that does not make sense. For example, a combination length greater than the total number of colours and no repetitions.

## **The Game**

### **F3.1 Placing coloured pegs into slots**

The game should provide the Player a way to place a specific colour into a specific slot.

### **F3.2 Removing coloured tokens from slots**

The Player should be able to remove a coloured token from a specific slot in a similar way as he placed it. He should also be able to do this without clearing the entire row.

### **F3.3 Checking the result of an attempt**

The Player should be able to check his latest attempt and receive feedback from the program.

### **F3.4 Returning to main menu**

The Player should be able to return to main menu without having to win/lose the current game.

### **F3.5 Notification of the Player if he has won or lost**

The game should provide a clear message to the Player, if he has won or lost the current game.



### **F3.6 Reveal the correct combination**

Once a game is finished with a win or a loss, the program should reveal the combination to the Player.

### **F3.7 Review the finished game**

The Player should be able to review all his attempts after finishing a game. He should not be able to edit the game (i.e., any of his attempts) during this stage.

## **1.5 Non-functional requirements**

### **NF1 Multi-platform**

The compiled application should be able to run on Windows and Linux.

### **NF2 Unity**

The application should be developed using the Unity framework.

### **NF3 C#**

Following from the previous non-functional requirement, the application should be developed in C#.

### **NF4 Building**

The application should have the possibility to be built automatically, preferably through CI pipelines.

### **NF5 Click-to-run**

The user should be able to download the compiled application and immediately run it without having to download any additional dependencies.

### **NF6 Versioning**

The application should be versioned on the FIT GitLab server.

### **NF7 Responsiveness**

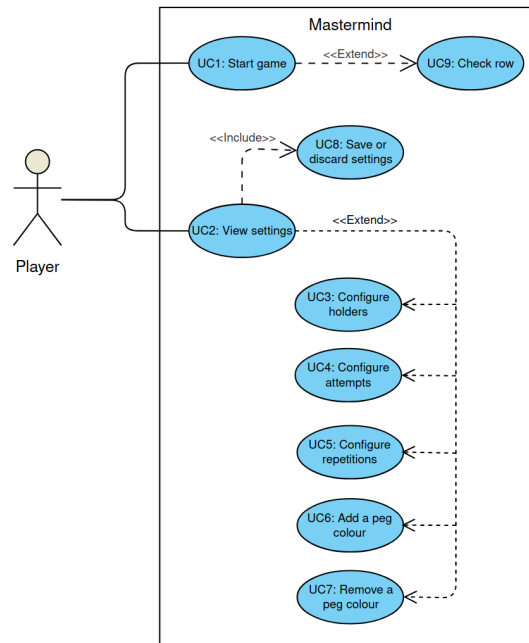
The GUI should be responsive to user input and window resizing.

### **NF8 Intuitiveness**

In order to be as intuitive as possible, the GUI will look as close to the original board game as possible.

## 1.6 Use case analysis

### 1.6.1 Use case diagram



■ **Figure 1.4** Diagram of the use cases described in this section

### 1.6.2 Actors

Since the application is designed to be a singleplayer game and not designed to be integrated with any other external systems, there shall be only one actor, the "Player".

### 1.6.3 UC1: Start game

#### Preconditions

- The Player is in the Main menu.

#### Postconditions

- The Player is in game.

#### Path

1. The Player clicks on "Play".
2. The system displays the Gameplay scene.

## 1.6.4 UC2: View settings

### Preconditions

- The Player is in the Main menu.

### Postconditions

- The Player is in the Settings menu.

### Path

1. The Player clicks on "Settings".
2. The system displays the Settings menu.

## 1.6.5 UC3: Configure holders

### Preconditions

- The Player is in the Settings menu.

### Path

1. a. The Player drags the slider to configure the number of holders.  
b. The Player directly inputs the number of holders into the text box next to the slider.
2. a. If the Player used the slider, the system changes the configuration to the value of the slider.  
b. If the Player typed the value directly in the text box, the system will validate the input, assign the value of the configuration, and move the slider to the corresponding position.
3. The system will configure the maximum number of repetitions, because it is dependent on the number of holders.

## 1.6.6 UC4: Configure attempts

### Preconditions

- The Player is in the Settings menu.

### Path

1. a. The Player drags the slider to configure the number of attempts.  
b. The Player directly inputs the number of attempts into the text box next to the slider.
2. a. If the Player used the slider, the system changes the configuration to the value of the slider.  
b. If the Player typed the value directly in the text box, the system will validate the input, assign the value of the configuration, and move the slider to the corresponding position.

## 1.6.7 UC5: Configure repetitions

### Preconditions

- The Player is in the Settings menu.

## Path

1.
  - a. The Player drags the slider to configure the number of repetitions (i.e., how many times each colour can appear).
  - b. The Player directly inputs the number of repetitions into the text box next to the slider.
2.
  - a. If the Player used the slider, the system changes the configuration to the value of the slider.
  - b. If the Player typed the value directly in the text box, the system will validate the input, assign the value of the configuration, and move the slider to the corresponding position.

### 1.6.8 UC6: Add a peg colour

#### Preconditions

- The Player is in the Settings menu.
- The number of colours is less than the maximum, defined in the implementation.

#### Postconditions

- The number of colours is incremented.

#### Path

1. The Player clicks on "Add peg".
2. The system creates a new peg of the default colour.
3. The Player sets the colour of the new peg.
4. The system reflects the change of colour on the screen.

### 1.6.9 UC7: Delete a peg colour

#### Preconditions

- The Player is in the Settings menu.
- There is a colour to be deleted.

#### Postconditions

- The number of colours is decremented.

#### Path

1. The Player clicks on "Delete" at a specific peg.
2. The system deletes the colour from the configuration and removes the peg from the screen.

### 1.6.10 UC8: Save or discard settings

#### Preconditions

- The Player is in the Settings menu.

## Postconditions

- The Player is in the Main menu.

## Path

- a. If the Player clicks on "Exit without saving", the use case ends. The changes are discarded and the system loads the Main menu.
  - b. If the Player clicks on "Save and exit", then the system will validate the data that the Player has set.
- a. If the data is valid, the system will save the configuration to file and to memory and load the Main menu.
  - b. If the data is invalid, the system will inform the Player on the screen and return back to the Settings menu in the same state as the Player has left it.

### 1.6.11 UC9: Check row

#### Preconditions

- The Player is in game.

#### Postconditions

- The current row is changed.

#### Path

1. The Player clicks on the button "Check row".
- a. If the row is filled, then the system provides feedback to the Player in the Result box.
  - b. If the row is not filled, the system reports to the Player that the row is not filled and the use case ends here.
- a. If the current row is the last row and it was not filled in correctly, the system reports to the Player "You lost".
  - b. If the current row was filled in correctly, the system reports to the Player "You won!"
  - c. If the current row is not the last row, the current row is set to the next row to allow the Player to fill it in next.
4. The token generators and the check row button are disabled to allow the Player to review their game without modifying it.

## Chapter 2

# Design

### 2.1 Technologies

#### 2.1.1 Candidates

##### Java

Due to the author's experience in programming in Java, this was the first programming language to be considered. There are several game engines available, notably:

- JMonkeyEngine<sup>1</sup>. One of its drawbacks is the lack of convenient IDE integration.
- LibGDX<sup>2</sup>. Like JMonkeyEngine, it does not have IDE integration.

However, Java as a language has a big drawback. It requires a JRE to run .jar files. This violates NF5 (Click-to-run) because if a user does not have the proper JRE installed and configured, he will have to do extra installation steps.

##### C++

Another language considered was C++. It could be compiled directly into an executable and there were several available APIs and game engines to choose from:

- SDL<sup>3</sup>. This library provides a layer to interact with the underlying OS APIs and OpenGL. Despite providing an extra layer, it is still fairly low-level and as a result has a steep learning curve. For example, many of the features that would be needed to implement Mastermind are not present and would need to be implemented almost from scratch, increasing development time. A lack of proper editor makes it difficult to design the GUI layout. Furthermore, when experimenting with the library, valgrind reported several leaks upon termination which were caused by the library itself, despite calling all the necessary functions provided by the documentation to free the memory.
- Unreal Engine 5<sup>4</sup>. This game engine is intended for 3D games with high-quality graphics. As a result, despite being very popular, well documented and supported by many platforms, it would be too much for the purposes of a 2D desktop game and will result in an application that requires too much computational resources for what it achieves.

---

<sup>1</sup><https://jmonkeyengine.org/>

<sup>2</sup><https://libgdx.com/>

<sup>3</sup><https://www.libsdl.org/>

<sup>4</sup><https://www.unrealengine.com/en-US>

## 2.1.2 Chosen technologies

### Unity

Unity is a cross-platform engine developed by Unity technologies. It is popular for both 3D and 2D game development, which makes it well suited for the task.

Unity uses the Gameobject-Component pattern [1], allowing to attach different components to different game objects, which represent instances of classes in the game. This helps to reduce code duplication by writing the same code, for example for collision detection, once and adding it to several game objects that need collision detection. One such example is the Horizontal Layout group [2], which allows the engine to automatically center and arrange UI items without the programmer having to write the source code for that again, from scratch.

The main features of Unity which will be used in the development of the application will be "scenes" and "prefabs". A scene is used to contain everything that is expected to be loaded in the game at one time [3]. For example, a Game scene would load game-related objects such as players and enemies, while a Settings scene would only load UI-related objects. Prefabs are a special type of component that allows fully configured GameObjects to be saved in the Project for reuse. These assets can then be shared between scenes, or even other projects without having to be configured again [4]. They function as a form of template with which one can instantiate game objects.

Furthermore, the Unity Editor provides a convenient and easy-to-use GUI to modify game scenes. It also integrates with the JetBrains Rider IDE to facilitate debugging. Additionally, the same Unity project can be recompiled for multiple platforms without the need to change the source code. It produces an executable alongside a DLL and resources directory, which allows the user to run the application immediately after downloading.

### C#

The programming language was chosen to be C#, with the primary reason being that Unity uses C# for its scripting API. It is also a memory-safe language, which will help prevent errors such as memory leaks, which were one of the reasons the SDL library was not chosen. Lastly, because the language is popular amongst the public and developed by Microsoft, it is thoroughly documented [5] and has a considerable amount of community support.

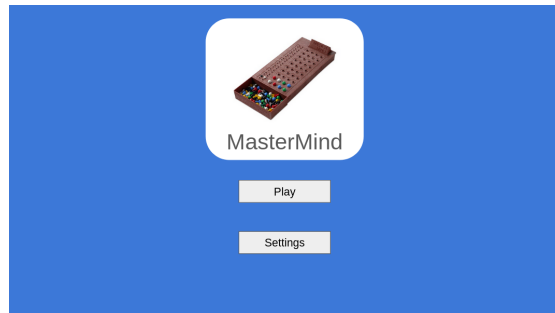
## 2.2 Designing the UI

As per NF8, in order to make the program as intuitive as possible, the UI will be as similar as possible to the physical version of the board game and the existing web versions of the game. However, since the existing implementations do not have any Main menu or Settings menu, the only part of the game that is limited by this constraint is the Game menu. Other menus such as the Main menu or the Settings menu can be designed without any constraints.

### 2.2.1 The Main Menu scene

After considering all these requirements, it was decided that the main menu will contain the title of the game, a small picture, and two buttons: "Play" and "Settings". The former button will start the game with the current settings, while the latter button will open the Settings menu for the user to configure the parameters of the game. For the picture, it was decided to use an image of the original board game version of Mastermind. An image authored by the user "ZeroOne" from the Wikimedia commons, however it was covered by the CC BY-SA license [6]. This means that the software must be released under the same license, because it would be considered a derivative of the work. Therefore, a different image under the CC0 license [7]

was selected, which did not place any constraints on the licensing of the resulting software. The resulting scene is designed to look as follows:



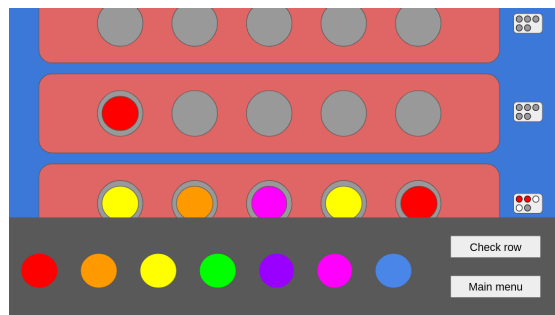
■ **Figure 2.1** Sample Main Menu GUI design

### 2.2.2 The Game scene

In order to have a similar layout to the existing implementations, the horizontal rows will contain slots for colours and smaller slots on the side to show the Player the correctness of his attempt. At the bottom there will be a row of "generators" and there will be a generator for each colour that the user has added. Next to the row of generators are two buttons: check the current row and return to the main menu.

Each generator will generate a new peg of its colour when the Player starts dragging it. The newly created peg will continue the dragging operation, i.e. the generator will remain in place while the peg will follow the mouse. This peg can then be dropped into place by the Player or, if the token is released in an invalid place, it will be destroyed.

Because the desktop screen usually has more horizontal space than vertical and the number of attempts is generally much higher than the length of the combination, the UI will need to account for this. The rows of holders will be placed in a scroll area to allow the user to fit more attempts than the height of the window/screen. Below is the visualization of the draft GUI:



■ **Figure 2.2** Sample Game scene GUI design

### 2.2.3 The Settings scene

The Settings scene will need to have a clear way for the Player to:

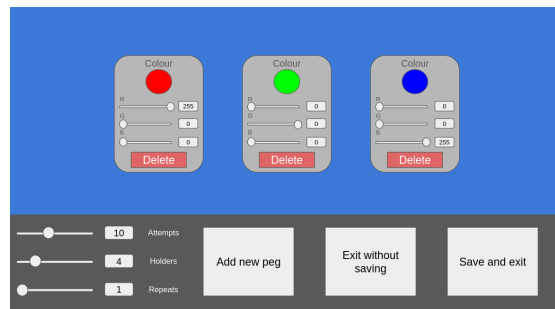
1. Save and exit.
2. Exit without saving.



3. Set the number of attempts.
4. Set the number of holders (combination length).
5. Set the number of repetitions.
6. Add a new token colour.
7. Edit a token colour.
8. Remove a token colour.

For points 1, 2, and 6, the most clear and intuitive way would be to have big visible buttons for the Player to click, with text stating the purpose of the button. For points 3, 4, and 5, as described in the use cases, sliders, and text input fields are available. The sliders and text input fields will be clearly annotated with text describing what each slider sets up.

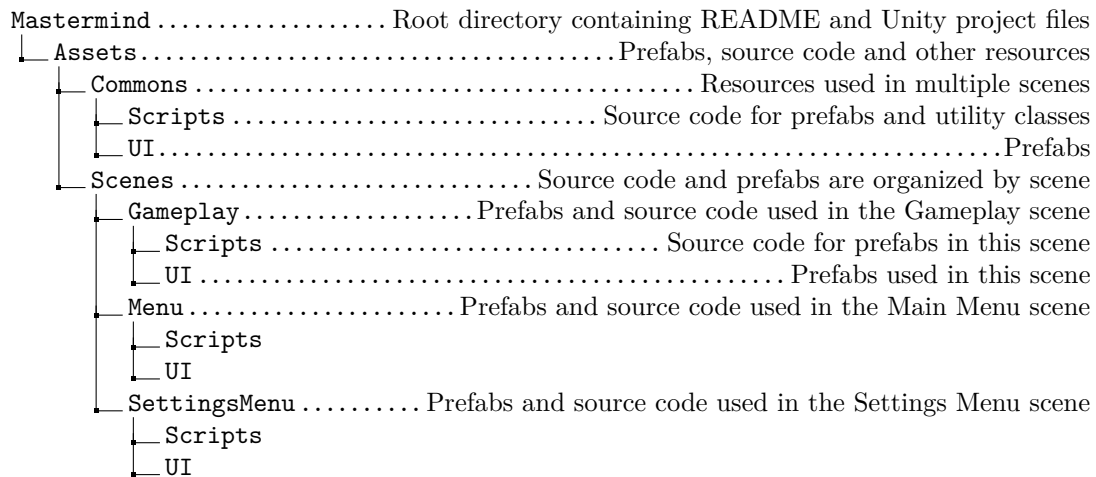
For points 7 and 8, it would make sense to have a separate delete button for every colour. To make it obvious that the delete button corresponds to the colour, they will be located in the same panel. The most intuitive way to provide extensive colour customization is through directly inputting the RGB values of the colour. Since RGB values are in the range from 0 to 255, it would make sense to have a slider for those as well. Similarly to the delete button, the RGB sliders will make it obvious which colour they are affecting because they will be placed in the same panel as the colour itself. This results in the following GUI layout for the Settings menu:



■ **Figure 2.3** Sample Settings Menu GUI design

## 2.3 Project structure

The project will be structured according to the three scenes, such that assets (source code and prefabs) used in the scene are contained in the scene directory. Some assets will be used across multiple scenes, for example static utility classes. For such cases a Commons directory is created, encompassing all such assets. The resulting project structure is as follows:



## 2.4 Class diagram

### The problem

Since Unity uses the Gameobject-Component pattern, it is an exception - rather than the norm - for objects inheriting from `MonoBehaviour` [8] to use inheritance. As a result, class diagrams of Unity projects will result in many classes all inheriting from one base class, which is not very helpful in analyzing or understanding the design.

### The solution

The solution is to ignore such framework classes and interfaces. Furthermore, prefabs play an important role in the code that uses them, so it has been decided to model these prefab assets as classes and consider any other class/object holding a reference to them as an association. If the prefabs have some special behaviour, in the form of a class extending `MonoBehavior`, attached to them using Unity's component system, this behaviour will be modelled as a generalization. Such an approach has helped to identify the large amount of references to prefabs for the `GameManager` class, which would have made it difficult to keep track of them.

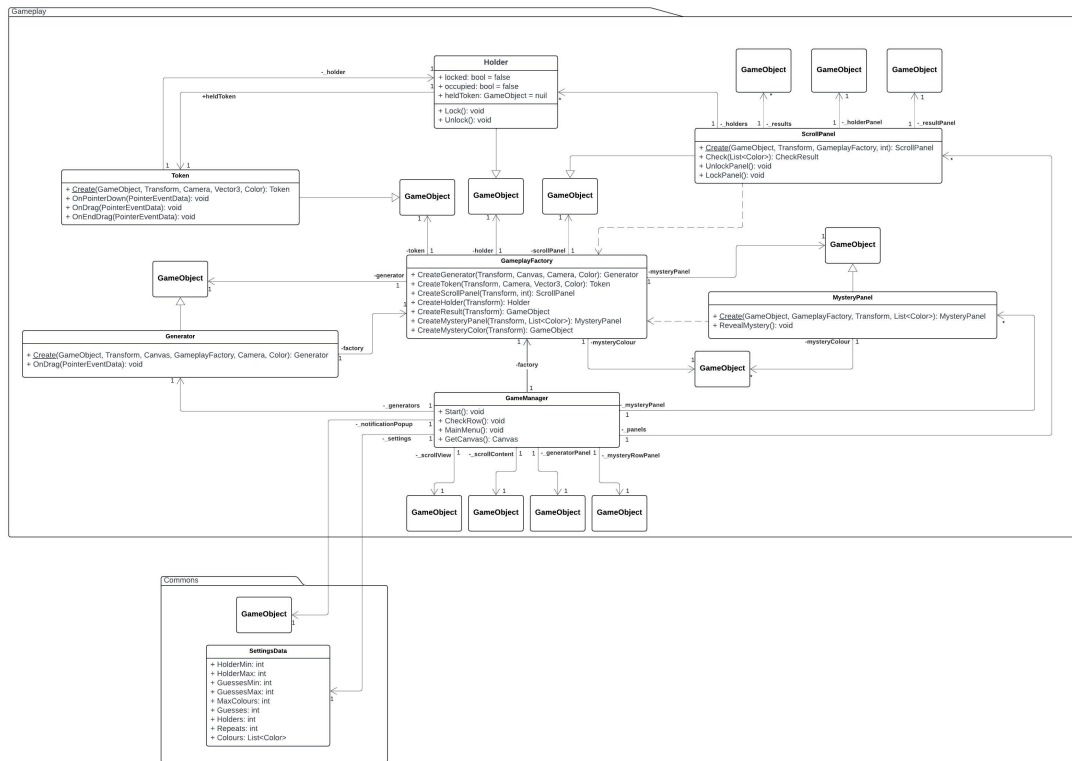
To tackle the large number of references, an approach similar to the Factory pattern was used. Normally, Factory classes guarantee to provide an implementation of an interface when requested; which concrete implementation is selected depends on many factors, but the key point is that the Factory decides which implementation to provide. However, in this case, the Factory guarantees to provide only one concrete implementation. Instead, the role of the Factory class is to decide which prefab reference to use when instantiating the object. This helps offload the reference keeping from the Manager class to the Factory class, so that the Manager class can only keep references to static game objects in the scene.

This section includes the class model for each scene alongside a general overview, and descriptions of objects in the scene and their public methods.

For clarity and simplicity:

- `GameObject` classes not referenced by any other class in the scene are omitted from the models.
- The descriptions of classes omit factory methods and static initializer methods.

## 2.4.1 The Gameplay scene



■ **Figure 2.4** Class diagram of the Gameplay scene

The Gameplay scene has one circular dependency between the **Holder** and **Token** classes. This is because the two classes are very closely related, however both expect that at initialization their references to each other are null and are only updated later on, if the objects exist. Therefore, this circular dependency will not affect the application.

### The GameManager class

This class is responsible for the internal state of the game, it holds important information such as the generated combination and which row is currently being solved.

- **public void Start()** is a method inherited from Unity's **MonoBehavior**. In the context of **GameManager**, it will be responsible for initializing most of the prefabs used, such as the generators and scroll row panels.
- **public void CheckRow()** will be responsible for checking the current row being solved, if valid, and updating the state of the game accordingly. Depending on the result, this may end the game. It will also update the current row. This function is intended to be called by a button in the scene.
- **public void MainMenu()** is also a function designed to be called by a button in the scene, its function being to load the main menu scene and end the game immediately.
- **public Canvas GetCanvas()** will allow all objects with a reference to the **GameManager** to access the scene's canvas without having to invoke expensive functions such as **GameObject.find()** [9].

## The GameplayFactory class

This class is designed to simplify reference-keeping to all the different prefabs intended to be instantiated dynamically. For some game objects that do not have any behaviour attached to them, the factory methods have a return type of `GameObject`. Furthermore, despite some of the methods ending up as simple wrappers for Unity's `Instantiate()` [10], the objects are still instantiated through the factory in order to simplify reference keeping and for the sake of consistency.

## The ScrollPanel class

This class is designed to be attached to a `GameObject` representing a row of holders and result icons on the screen. Its main objective will be to operate on all the contained holders and results as a whole, instead of one-by-one.

- `public CheckResult Check(List<Color>)` will check the colour combination stored in the holders of this scrollPanel against the passed colour list.
- `public void UnlockPanel()` will unlock the panel and subsequently all holders, to allow the user to interact with them; dragging objects tokens in or out.
- `public void LockPanel()` will lock the panel, disabling any interactions

## The Holder class

This class is designed to be attached to a `GameObject` representing a holder. Its job is to help `ScrollPanels` interface with the held tokens. `Holder` is simple enough to not require a dedicated initializer method - Unity's built-in `Instantiate()` method is sufficient.

- `public void Lock()` will lock the currently held token in place, if there is one.
- `public void Unlock()` will unlock the currently held token in place.

## The Token class

This is the behaviour of the token object which is click-and-dragged into place by the user. It symbolizes the coloured pegs in the real game.

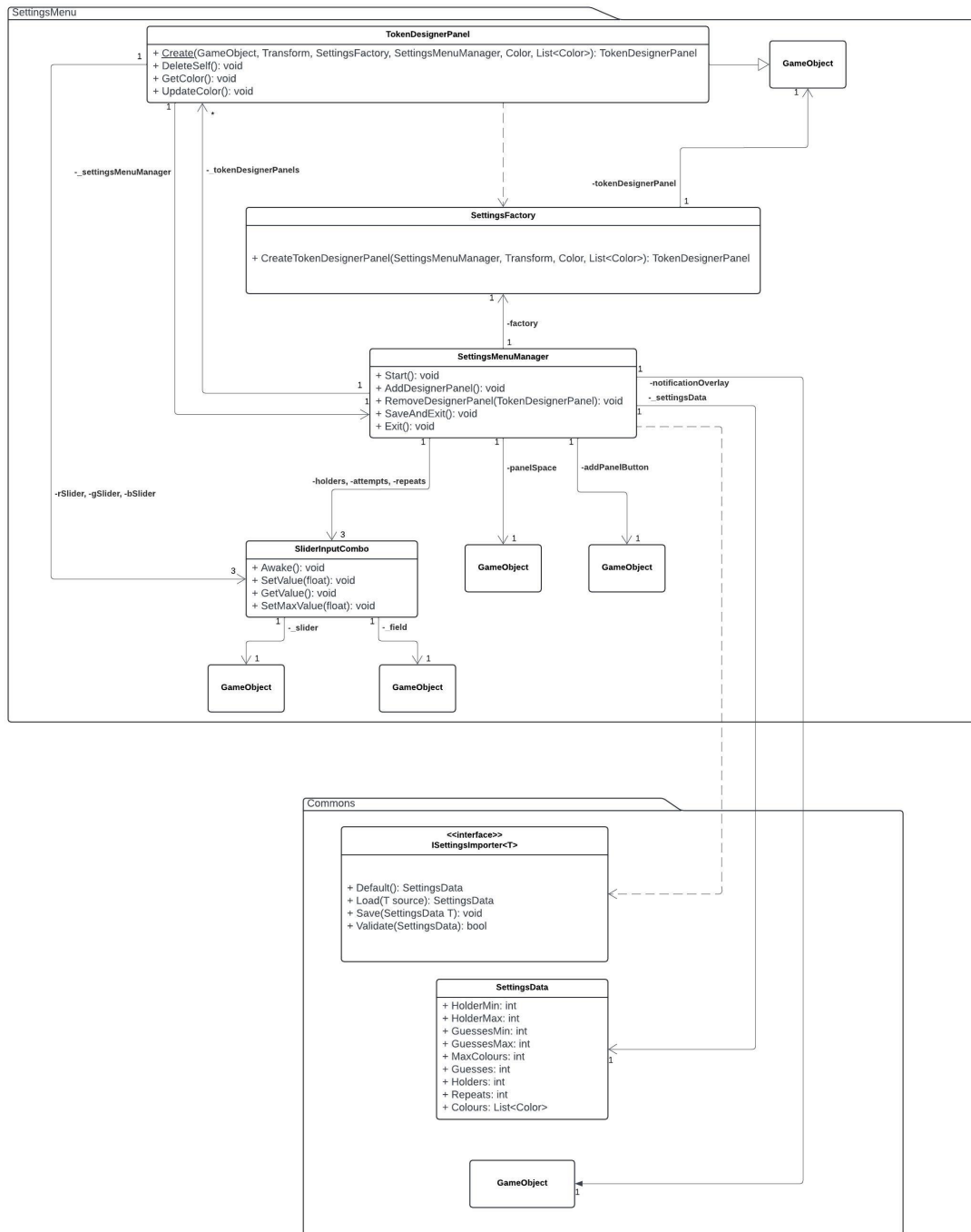
- `public void OnPointerDown(PointerEventData eventData)` Function provided by Unity's `IPointerDownHandler` [11] will verify that the token is not being held and should not be dragged before beginning the dragging operation.
- `public void OnDrag(PointerEventData eventData)` Function provided by Unity's `IDragHandler` [12] will handle the movement of the token.
- `public void OnEndDrag(PointerEventData eventData)` Function provided by Unity's `IEndDragHandler` will handle the end of the drag. If the dragged token is not released over a holder, it should be destroyed. Otherwise, it should be placed into the holder.

## The MysteryPanel class

This class is the behaviour of the object containing `MysteryColours`, intended for storing the secret combination until it needs to be revealed. Designed as a single point of access for all the `MysteryColours`, it provides a method to reveal the secret colours.

- `public void RevealMystery()` will reveal the secret combination by un-hiding the colours in all the `MysteryColour` instances referenced by this object.

## 2.4.2 The Settings scene



■ **Figure 2.5** Class diagram of the Settings scene

The Settings scene contains a circular dependency between `SettingsMenuManager` and `TokenDesignerPanel`. However, the `SettingsMenuManager` class is guaranteed to be present at all times, so it is not a problem.

## The `SettingsMenuManager` class

This class is responsible for maintaining the state of the settings scene, as well as using the factory to instantiate other elements on scene.

- `public void Start()` initializes and sets all variables and values to their defaults.
- `public void AddDesignerPanel()` is a method triggered by the "Add new Token" button, which will use the factory to create a new panel to configure (Design) a token.
- `public void RemoveTokenDesignerPanel(TokenDesignerPanel item)` will remove the token designer panels and hence remove the colour from the configuration, by reference. Triggered from the delete button in the panel itself.
- `public void SaveAndExit()` will write all the changes to file and return back to the main menu.
- `public void Exit()` will immediately return back to the main menu scene without saving.

## The `SettingsFactory` class

This class is responsible for reference-keeping to all the prefabs that are supposed to be instantiated dynamically in the scene. For the settings scene, this means keeping track of the `TokenDesignerPanel` and `ColorSample` prefabs.

## The `TokenDesignerPanel` class

This class defines the behaviour associated with customizing and creating a new token colour. It will have a way of configuring the colour using 3 RGB sliders, show a preview of the created colour, and also contain a button to delete the colour.

- `public void DeleteSelf()` will call the `RemoveDesignerPanel()` method in the settings menu manager in order to delete the current panel.
- `public Color GetColor()` will return the current color shown by the preview and stored in this class.
- `public void UpdateColor(Color color)` will update the color stored in this class and update the preview image as well to reflect the change.

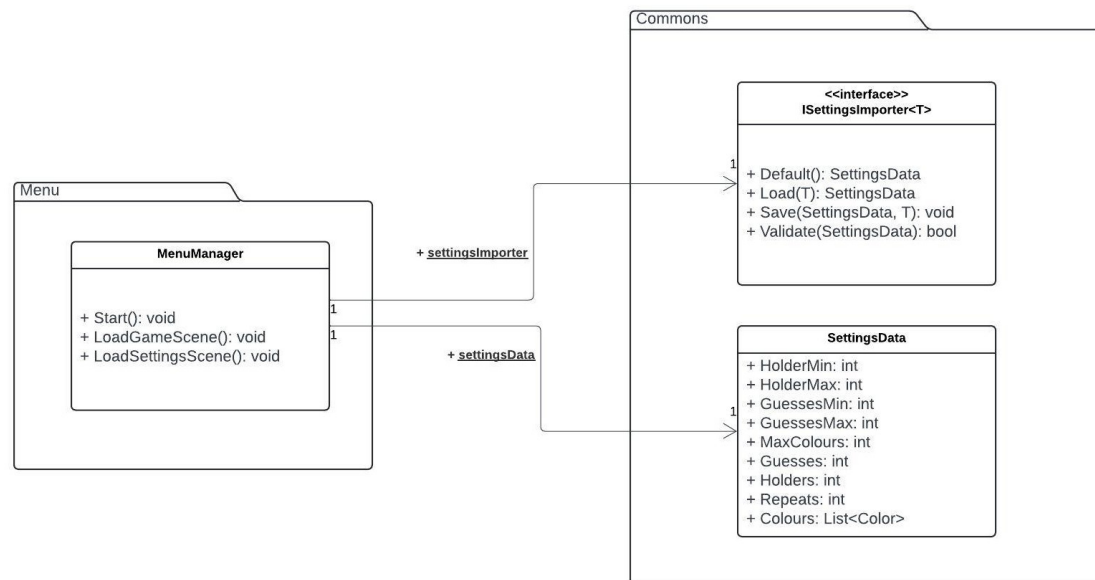
## The `SliderInputCombo` class

This class is designed to help group together and abstract over some collective operations with Unity's `Slider` [13] and `TMPText` [14] components. Such a class is needed because sliders are used throughout the settings menu, and it is usually more convenient for the user to also have a method of inputting the value manually instead of using the slider - particularly for large number ranges where fine movement of the slider is impossible.

- `public void Awake()` is a method inherited from `MonoBehavior` similar to `Start()`, however it is called before the first frame [15] so that it is able to get references to child game objects in time.
- `public void SetValue(float value)` will set the numeric value in both the slider and text fields associated with this object.
- `public void GetValue()` will return the numeric value held by both the slider and the text field. They should be the same, so there is no difference which one is returned.

- `public void SetMaxValue(float value)` Unity's `Slider` has a maximum and minimum value. This method allows to directly access the underlying maximum value of the slider from the outside.

### 2.4.3 The Menu scene

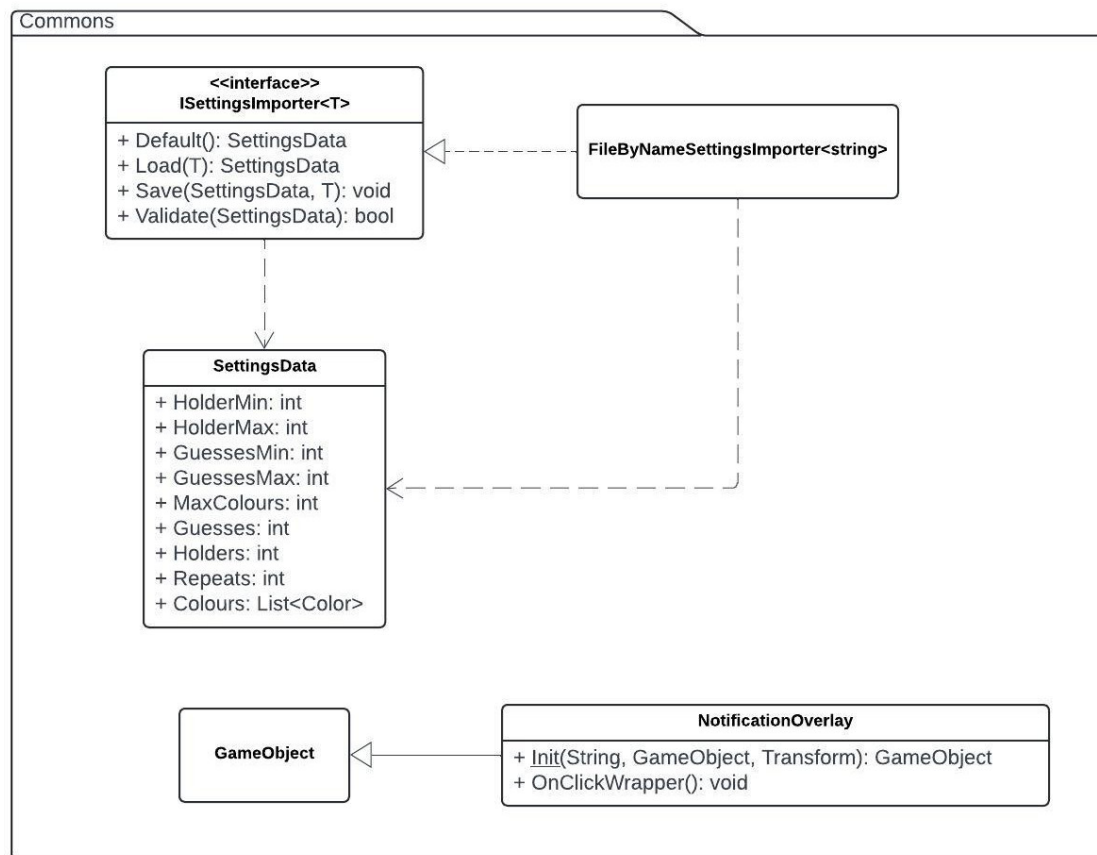


■ **Figure 2.6** Class diagram of the Menu scene

The Menu scene is fairly simple, so as a result it does not have any circular dependencies. It only has one class, whose job is to:

- facilitate switching between scenes with the click of the relevant button.
- Instantiate the Settings importer into a static field to allow it to be accessed by other classes.
- Use the settings importer to perform an initial import of settings upon game start.

### 2.4.4 The Commons folder



■ **Figure 2.7** Class diagram of the Commons folder

The Commons folder contains utility classes used in all parts of the application. It contains an interface for a settings importer and a basic implementation, and a notification overlay. The methods of the importer are as follows:

- **public SettingsData Default()** will return the default setting parameters. This can be used, for example, as a fallback in case importing fails.
- **public SettingsData Load(T source)** will load the setting parameters from a given source specified by the implementation and return them in the format of a **SettingsData** object.
- **public void Save(SettingsData data, T destination)** will write the given **SettingsData** object to the destination.
- **public bool Validate(SettingsData data)** will allow each implementation to provide its own specification of what is considered a valid data.



# Implementation

## 3.1 Clicking and dragging pegs into place

In order to implement the click-and-drag functionality, Unity provides several interfaces: `IDragHandler`, `IBeginDragHandler` and `IEndDragHandler`. It was decided to use `IBeginDragHandler` instead of `IPointerDownHandler` for consistency. Simple click and drag functionality can be implemented using only one of the interfaces - `IDragHandler`. However, the click-and-drag action needed for the game requires some logic to check at the beginning and end of the drag. This logic will remove the peg from the holder, if it is currently being held, at the beginning of the drag:

```
if (!_isBeingHeld)
{
    if (!_holder.locked)
    {
        _holder.occupied = false;
        _holder.heldToken = null;
        _isBeingHeld = false;
    }
}
```

■ **Listing 3.1** The logic for picking up a peg

At the end of the drag, it will check if the peg is above a holder and that the holder is free - placing the peg in the holder:

New pegs will need to be created by attempting to drag a Generator. To do this, the Generator will have to implement `IBeginDragHandler`. When the drag is initiated, the Generator will instantiate a new Token object - representing the newly created peg - and pass on the dragging action to the new Token object. To accomplish this, it is possible to change which object is currently being dragged by the mouse pointer by directly changing the `pointerDrag` value in the `eventData`:

```

if (_holder.locked)
{
    _holder = null;
}
else
{
    if (_holder.heldToken)
    {
        Destroy(_holder.heldToken);
        _holder.heldToken = null;
    }

    _transform.SetParent(collider.gameObject.transform);
    _transform.localPosition = new Vector3(0, 0, 0);
    _holder.occupied = true;
    _holder.heldToken = gameObject;
    _isBeingHeld = true;
    return;
}

```

■ **Listing 3.2** The logic for dropping a peg

```

public void OnDrag(PointerEventData eventData)
{
    var instantiated =
        _factory.CreateToken(_gameManager.GetCanvas().transform,
            _gameManager, _camera,
            _camera.ScreenToWorldPoint(Input.mousePosition), _color);

    eventData.pointerDrag = instantiated.gameObject;
}

```

■ **Listing 3.3** The Generator class drag logic

## 3.2 Setting the colour of the pegs

The initial design called for three sliders - red, green, and blue - to control the colour of each peg. In order to prevent invalid input, identical colours were not allowed to be present in the configuration. In the first iteration, the RGB values were compared directly, which meant that an RGB value of (255, 255, 255) was considered different from (254, 254, 254) despite being almost the same colour to the naked eye. Therefore, some advanced method of colour comparison was needed.

The second attempted method for colour comparison was calculating the Euclidean distance between two colours using  $\sqrt{r_2^2 + g_2^2 + b_2^2} - \sqrt{r_1^2 + g_1^2 + b_1^2}$ . The further the distance between the two colours, the bigger the difference between the two colours. The drawback of this method is that the RGB colour space is not perceptually uniform; that is, a change in length in any direction of the colour space will not be perceived by the human eye as the same change. For example, the two colours "Pakistan green" (RGB 0, 64, 0) and "Coquelicot" (RGB 255, 64, 0) are both the same distance (128 units) from "French rose" (RGB 255, 64, 128). However, Coquelicot

and French rose appear very similar to the naked eye. This is demonstrated by the image below:



■ **Figure 3.1** Samples of the colours side-by-side taken from <https://coolors.co/>

The final attempted method for colour comparison was to use a different colour space - one that is better suited to reflect the caveats of human colour perception. The chosen colour space was "CIELAB", as it best reflects the human colour perception. Because there is no direct conversion from RGB to CIELAB, it is necessary to perform an intermediate conversion into another colour space: "CIEXYZ". There is no built-in method in Unity to convert between RGB and CIEXYZ, so it is necessary to multiply the RGB column vector by the following matrix:

$$\begin{bmatrix} 0.49 & 0.31 & 0.20 \\ 0.18 & 0.81 & 0.01 \\ 0.00 & 0.01 & 0.99 \end{bmatrix}$$

From CIEXYZ, CIELAB can be obtained using:

$$L = 116f(Y/Y_n)$$

$$A = 500(f(X/X_n) - f(Y/Y_n))$$

$$B = 200(f(Y/Y_n) - f(Z/Z_n))$$

where

$$f(t) = \begin{cases} \sqrt[3]{t} & \text{if } t > \delta^3 \\ \frac{1}{3}t\delta^{-2} + \frac{4}{29} & \text{otherwise} \end{cases}$$

$$\delta = \frac{6}{29}$$

The values for  $X_n, Y_n, Z_n$  vary depending on the illumination. However, under "standard" illumination (natural daylight), their values are:

$$X_n = 95.05$$

$$Y_n = 100$$

$$Z_n = 108.88$$

This approach, using a different colour space which better reflects human colour perception, yielded significantly better results. However, upon testing, it was noted that the perception of colour difference can be affected by the Player's own colour perception - e.g. colour blindness - or the quality of their computer screen. This meant that running the program on different hardware could result in different outcomes: some colours may appear identical on one screen but different on another. This could cause confusion for the Player because the check for identical colours could appear random to him, when in reality it was the fault of the hardware.

The solution to this was to limit the choice of colour to a predefined colour palette. The game would display a grid of different colours, and the Player would select his desired colour by clicking on the colour in the grid. The colours in the palette will be specifically chosen such that no two colours are too similar, allowing for the input check to do the simple direct comparison without the need to have complicated conversion and comparison algorithms.

The colour grid is implemented to have the the colours highlight if a Player is hovering their mouse over them. To achieve this, Unity's `IPointerEnterHandler`[16] and `IPointerExitHandler`[17] interfaces were used. Each interface introduces methods which are executed when the mouse pointer enters the bounds of the object. In this case, the methods are implemented to brighten the displayed grid colour. However, for already bright colours this did not appear to change anything - on the other hand, if the colours were darkened when the mouse pointed hovered over them, dark colours would be affected by the same issue. The solution was to either brighten or darken the colour which is being hovered over, depending on the colour's brightness. To find out a colour's brightness, the "HSV" colour space is used, where V is the brightness value. Unity provides a built-in method for conversion to HSV in the `Color` class. If the brightness is greater than a certain threshold, it is divided by a constant and if it is lower than the threshold it is multiplied. An edge case is when the brightness is 0, i.e. the colour is completely dark: in this case, the brightness is incremented instead. The altered HSV colour is then converted back to RGB.

```
public void OnPointerEnter(PointerEventData eventData)
{
    var tmp = gameObject.GetComponent<Image>().color;
    float h, s, v;
    Color.RGBToHSV(tmp, out h, out s, out v);
    if (v > 0.85)
    {
        v /= 1.3f;
    }
    else
    {
        if (v == 0)
        {
            v += 0.2f;
        }
        v *= 1.3f;
    }
    gameObject.GetComponent<Image>().color = Color.HSVToRGB(h, s, v);
}
```

■ **Listing 3.4** The colour highlighting function

This change resulted in the following GUI for the Settings menu:



■ **Figure 3.2** The redesigned Settings menu

### 3.3 Generation of combinations

Each colour is stored into a data structure containing the colour itself and the colour's remaining repetitions:

```
private class ColorAndCounter
{
    public Color Color;
    public int Repeats;
}
```

■ **Listing 3.5** The data structure used in the combination generation algorithm

These data structures are then all stored in an array and a random index is selected. This random index determines which colour is selected for the position. Every time a colour is selected, it is appended to the current combination that is being generated and its repetition number is decreased. If the repetition number reaches zero, the colour cannot repeat, so the colour's respective data structure is removed from the array.

In order to facilitate testing, the generator is written in a separate class. Therefore, it is written with the intention of it behaving as a separate module, therefore not relying on receiving validated input from other parts of the project. Therefore, it has to account for the case when all colours are exhausted before the full length of the combination can be generated. If such a case is encountered, an exception is thrown. This results in the following algorithm:

```
public List<Color> Generate(int length)
{
    List<Color> ret = new();
    List<ColorAndCounter> list = new();
    foreach (var color in _colorSet)
    {
        var newItem = new ColorAndCounter();
        newItem.Color = color;
        newItem.Repeats = _repeats;
        list.Add(newItem);
    }

    for (var i = 0; i < length; i++)
    {
        if (list.Count < 1)
        {
            throw new Exception("Not enough Colours and " +
                "repetitions for given length");
        }
        int nextIndex = Random.Range(0, list.Count);
        ret.Add(list[nextIndex].Color);
        list[nextIndex].Repeats--;
        if (list[nextIndex].Repeats <= 0)
        {
            list.RemoveAt(nextIndex);
        }
    }

    return ret;
}
```

■ **Listing 3.6** The combination generation algorithm

### 3.4 Checking the Player's attempt

The algorithm will take as an argument a list of game objects, called the result list, whose colour will be modified to reflect the ideal matches, misplaced matches, and mismatches. The attempt checking algorithm will have two stages:

1. Check all the ideal matches between the attempt and the target combination.
2. Check all the misplaced matches between the attempt and the target combination.

In the first stage, it will go through each index and compare the colours in the attempt and in the target combination at that index. If the colours are identical, both of the indexes are marked as checked and a colour signifying an ideal match is added to the result list.

To mark a colour as checked, a separate data structure is used to store both the target and the attempt combination. It contains the colour itself and a boolean flag **Status** to determine if this colour has already been checked or not.

In the second stage, a quadratic algorithm is used. Every unchecked colour in the attempt is compared with every unchecked colour in the target combination. If a match is found, a colour signifying a misplaced match is added to the result list.

```
private class ColorStatusPair
{
    public Color Color;
    public bool Status;
}
```

■ **Listing 3.7** The data structure used in the checking algorithm

```
for (var i = 0; i < current.Count; i++)
{
    if (currentPair[i].Color == targetPair[i].Color)
    {
        _results[currentResultIndex].GetComponent<Image>().color =
            Color.black;
        currentResultIndex++;
        currentPair[i].Status = true;
        targetPair[i].Status = true;
        correctCount++;
    }
}
```

■ **Listing 3.8** The main logic of the first stage of the checking algorithm

```
for (var i = 0; i < current.Count; i++)
{
    if (!currentPair[i].Status)
    {
        for (var j = 0; j < targetPair.Count; j++)
        {
            if (!targetPair[j].Status && currentPair[i].Color ==
                targetPair[j].Color)
            {
                _results[currentResultIndex].GetComponent<Image>().color =
                    Color.white;
                currentResultIndex++;
                currentPair[i].Status = true;
                targetPair[j].Status = true;
                break;
            }
        }
    }
}
```

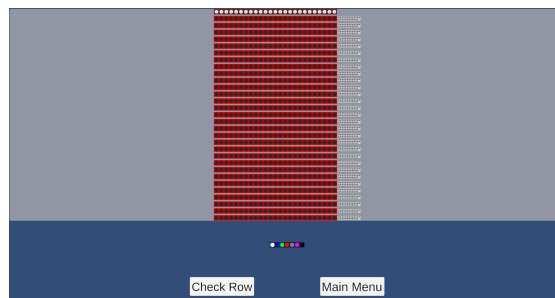
■ **Listing 3.9** The main logic of the second stage of the checking algorithm

Furthermore, it was decided later on to change the ideal match colour from red to black, because it is more common.

### 3.5 Zooming in and out in the Game scene

During a usability demo, it was pointed out that the game has a significant drawback because it does not show all the rows of holders simultaneously on screen. For a game that requires the Player to look at all his previous attempt to make their next move, it will be very uncomfortable to have to always scroll back and forth to view their previous answers. In order to make everything fit on the screen, it was decided to dynamically make the rows and the pegs smaller. For large input lengths, this resulted in a very small and difficult to see GUI, so the possibility to zoom in or out was added by holding left control and using the mouse scroll wheel.

The first version of scaling the GUI was implemented using Unity's canvas scale function. This changed the scale of all the items on the canvas, including the text and buttons. As a result, the "Check row" and "Main menu" buttons were difficult to differentiate between on the default scale. Therefore, the scale functionality was instead implemented using `Transform.localScale`[18] and the specific objects to be scaled can be selected, instead of having to scale everything on screen. For this purpose, the functions `AddObjectToScalables(GameObject)` and `RemoveObjectFromScalables(GameObject)` which will enable or disable an object from being scaled when the control and scroll wheel combination is inputted. This results in the following Game scene at the smallest zoom level:



■ **Figure 3.3** The Game scene with a large number of attempts at the minimum zoom level.

### 3.6 Build and Deployment

Because there is no standalone Unity compiler, Unity projects need to be built using the Unity editor, which makes it difficult to build Unity projects in a CI/CD pipeline. Unity offers an automated cloud building service, however, it requires a paid subscription which is not ideal. The solution is to use an open-source community-managed Docker image `unity-ci`[19]. It starts a container with the Unity editor already installed and it contains some wrapper commands for launching the editor build task from the CI pipeline. The setup process is documented at <https://game.ci/docs/gitlab/getting-started> and requires adding a `BuildCommand.cs` file with the build commands and configuring the editor in the pipeline with your own Unity account credentials. The successful setup allowed the pipeline on the faculty GitLab server to build the project, run tests, and publish it.

To publish the built application, GitLab's Package Registry and Release functionality was used. This allows the built application to be uploaded to the GitLab Package Registry and a new Release to be created in the Project Releases page. The Release will then contain download links to the Package Registry, providing visitors to the project with a convenient way to download the application. Gitlab Releases also provide a way to see a history of all releases in chronological order. The tasks to upload the built application to the Package Registry and create a Release are performed in a manually-triggered publish job, which runs when setting an environment variable



"VERSION". This variable will be read, checked, and then re-written into the variable "VER" used by the other jobs in the stage.

```

create_release:
  stage: publish
  image: registry.gitlab.com/gitlab-org/release-cli:latest
  variables:
    REGISTRY_URL: "${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/generic"
  dependencies:
    - publish-linux
    - publish-windows
    - publish
  needs:
    - publish-linux
    - publish-windows
    - publish
  script:
    - echo "Creating release ${VER}"
  release:
    tag_name: "${VER}"
    name: "Logic desktop release ${VER}"
    description: "Release generated by CI pipeline"
    assets:
      links:
        - name: "Linux"
          url: "${REGISTRY_URL}/linux/${VER}/logic-desktop.zip"
          link_type: package
        - name: "Windows"
          url: "${REGISTRY_URL}/windows/${VER}/logic-desktop.zip"
          link_type: package

```

■ **Listing 3.10** The create\_release job, a part of the publish stage

## 3.7 Testing

### 3.7.1 Manual testing of the UI

Manual testing of the UI has revealed some errors in the implementation, namely:

- If pressing both Left and Right mouse buttons at once, the drag input would be duplicated. This causes a problem when one of the buttons releases over a holder, but the other one keeps dragging the peg. Releasing one button called the `OnEndDrag` function which put the holder into a state of holding the token. However, the other button would continue dragging the peg and release it over an invalid location, destroying the token. Such a chain of events caused the holder to store a reference to a non-existent peg which could not be dragged out, and the row could not be checked because the check function failed due to a null reference.
- It was possible to place pegs into holders which were not visible. If the row of holders was scrolled up or down in such a way that the current row went outside of the scroll window (view port), the hitboxes for the holders in the row were still active, meaning that the collision

between the peg and the holder was still detected and it would be considered a valid place to drop a peg.

To fix the first issue, a check at the beginning of each drag function was added, to cancel the drag action if the registered mouse input did not correspond to the left mouse button.

```
if (eventData.button != PointerEventData.InputButton.Left)
{
    eventData.pointerDrag = null;
    return;
}
```

■ **Listing 3.11** Checking that the mouse button used to drag an object is the left mouse button

To fix the problem with placing pegs outside of the viewport, it was decided to add a hitbox for the entire viewport itself. The peg would only be placed if it is inside the viewport hitbox, in addition to the already existing requirement of having to collide with a holder.

### 3.7.2 Integration and System tests

Because of the nature of the application, it would not make sense to write Integration and System tests. Such tests would require having to simulate user input at exact coordinates. If the layout of the UI is changed, all these tests will have to be carefully reviewed and updated. Furthermore, the application is designed to scale appropriately if the window is resized. In such a case, the coordinates will also change and the testing will become irrelevant. Therefore, such testing does not bring much value, is very brittle and quite complex to implement.

### 3.7.3 Unit tests

It was decided that the most critical part of the application which needs to be tested thoroughly is the business logic, such as the combination generation, checking of the combination, and validating the settings. Therefore all these functionalities must be covered by unit tests that cover all cases. However, it was discovered that the business logic was not implemented in a testable way. It was implemented directly in the game objects, meaning that in order to test the logic it was necessary to instantiate all the game objects and their dependencies. To fix this, all business logic was moved to separate classes and specific interfaces were created, in case different variations of the business logic need to be implemented later on. The interfaces created as a result of this refactoring are:

```
public interface ICombinationChecker
{
    public bool CheckCombination(List<Color> target, List<Color> current);
}
```

■ **Listing 3.12** Interface for the attempt checker

And they each have one implementation: `BasicCombinationGenerator` and `UnorderedCombinationChecker`. The tests for the `BasicCombinationGenerator` are designed to test them as separate entities, which do not rely on input validation from other parts of the application. Therefore, these tests will cover the following cases:

```
public interface ICombinationGenerator
{
    public List<Color> Generate(int length);
}
```

■ **Listing 3.13** Interface for the combination generator

1. A simple test case where a combination of length 3 is generated from 3 colours, with no repetitions.
2. A combination of a length greater than the number of colours is generated thanks to colour repetitions.
3. A combination of a length shorter than the number of colours is generated.
4. Invalid input where there are not enough colours and repetitions to generate the combination of desired length.

The tests for the `UnorderedCombinationChecker` cover the following cases:

1. All the colours in the attempt are correct.
2. All the colours in the attempt are wrong.
3. Some colours correct, some partially correct, and others wrong.
4. The result is unordered and the same position is not checked twice.
5. Invalid input where the result list is shorter than the combinations being compared.
6. Invalid input where the result list is longer than the combinations being compared.

### 3.8 Used third-party libraries and assets

Unity's built-in `GridLayout` has an issue where only the first row is centered and the subsequent rows are aligned to the left. To fix this, a third-party library called "Beardy's Layout Group" [20] was used. It is distributed under the Unity companion license [21] which allows to use the library freely as long as it is used with the Unity Engine.

The high-resolution pictures of the circles were taken from <https://www.iconsdb.com> which allows using the icon for anywhere, except other icon websites, and for any purpose. It is also not required to request permission to use the icons or give credit to the website.

# Conclusion

The resulting application fulfills all requirements set out by the assignment, as well as accomplishing extra functionality such as saving the configuration to a file and allowing the user to dynamically change the size and scale of the game. The application is stable on both Windows and Linux and contains an automatic deployment pipeline. Some possible further modifications to the application would be purely cosmetic: adding textures to the pegs and the background, custom fonts and logos, etc. A possible technical modification could be allowing the Player to continue a game after returning to the menu, because currently the game is reset when leaving the Game scene.

However, if the application were to be developed again, a different framework would have been chosen. This is because of a flaw in the way potential technologies were researched: only game engines were considered because Mastermind is a game itself. However, it does not require any of the features that most game engines provide, such as 3D-rendering, physics, etc. Instead, the application could have been written using a regular UI framework, for example .NET. In the present day, most UI frameworks are multiplatform which would not have interfered with the non-functional requirements.

The application is available to download at  
<https://gitlab.fit.cvut.cz/rybakki1/mastermind/-/releases>.

# Bibliography

1. *Introduction to components*. Available also from: <https://docs.unity3d.com/Manual/Components.html>.
2. *Horizontal layout group*. Available also from: <https://docs.unity3d.com/Packages/com.unity.ugui@3.0/manual/script-HorizontalLayoutGroup.html>.
3. *Unity - Manual: Scenes*. Available also from: <https://docs.unity3d.com/Manual/CreatingScenes.html>.
4. *Unity - Manual: Prefabs*. Available also from: <https://docs.unity3d.com/Manual/Prefabs.html>.
5. *C# Language documentation*. Available also from: <https://learn.microsoft.com/en-us/dotnet/csharp/>.
6. *Creative commons share-alike license*. Available also from: <https://creativecommons.org/licenses/by-sa/2.0/>.
7. *Creative commons universal license*. Available also from: <https://creativecommons.org/publicdomain/zero/1.0/deed.en>.
8. *Unity - Manual: MonoBehaviour*. Available also from: <https://docs.unity3d.com/Manual/class-MonoBehaviour.html>.
9. *Unity - Scripting API: GameObject.Find*. Available also from: <https://docs.unity3d.com/ScriptReference/GameObject.Find.html>.
10. *Unity - Scripting API: Object.Instantiate*. Available also from: <https://docs.unity3d.com/ScriptReference/Object.Instantiate.html>.
11. *Unity - Scripting API: IPointerDownHandler*. Available also from: <https://docs.unity3d.com/2018.3/Documentation/ScriptReference/EventSystems.IPointerDownHandler.html>.
12. *Unity - Scripting API: IDragHandler*. Available also from: <https://docs.unity3d.com/2018.3/Documentation/ScriptReference/EventSystems.IDragHandler.html>.
13. *Unity - Scripting API: Slider*. Available also from: <https://docs.unity3d.com/2018.2/Documentation/ScriptReference/UI.Slider.html>.
14. *Class TMP\_Text*. Available also from: [https://docs.unity3d.com/Packages/com.unity.textmeshpro@1.0/api/TMPro.TMP\\_Text.html](https://docs.unity3d.com/Packages/com.unity.textmeshpro@1.0/api/TMPro.TMP_Text.html).
15. *Unity - Scripting API: MonoBehaviour.Start*. Available also from: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html>.

16. *Unity - Scripting API: IPointerEnterHandler*. Available also from: <https://docs.unity3d.com/2019.1/Documentation/ScriptReference/EventSystems.IPointerEnterHandler.html>.
17. *Unity - Scripting API: IPointerExitHandler*. Available also from: <https://docs.unity3d.com/2018.3/Documentation/ScriptReference/EventSystems.IPointerExitHandler.html>.
18. *Unity - Scripting API: Transform.localScale*. Available also from: <https://docs.unity3d.com/ScriptReference/Transform-localScale.html>.
19. *GameCI: getting started*. Available also from: <https://game.ci/docs/gitlab/getting-started>.
20. *Beardy's layout group*. Available also from: <https://github.com/mrbeardy/BeardyGridLayout>.
21. *Unity companion license*. Available also from: <https://unity.com/legal/licenses/unity-companion-license>.

# Contents of the attached .zip file

```
mastermind ..... Root directory of the .zip file
├── target ..... Directory containing the compiled application
│   ├── windows ..... Compiled application for Windows
│   └── linux ..... Compiled application for Linux
├── src ..... Directory containing the source code
├── thesis ..... Directory containing the thesis PDF and images
│   ├── diagrams ..... Directory containing all images used in the thesis in full resolution
│   └── Thesis.pdf ..... The thesis in PDF format
```