**The Coding of Chess Variations  
Implementing Different Ways to Play Chess into a Standard Chess Program**



**Matura Paper, Kantonsschule Sargans**

Attila Vizhanyo, 4bNP

**Supervisor:**Herr Lukas Fehr

**Submitted on:**  
8 January 2024

# Preface

In the pursuit of finding a subject for my Matura project, my objective was to combine two of my passions: chess and computer programming.

I have been playing chess for an extended period. My grandfather first introduced me to chess when I was young. I was captivated by the strategic and tactical elements of chess.

My second interest is programming, also called coding. Despite not having much programming experience, I find it interesting for its logical way of thinking and clear structure. Subsequently, I was eager to enhance my coding skills with a suitable coding project.

My idea was to program different versions, or variations of chess by changing the rules of the original game. I was inspired by a website called chess.com that published such variations, and consequently I was excited to come up with my own ideas of ways to play chess. By changing the rules, my goal was to make chess more enjoyable for people who may not appreciate its strategic complexity and memorization.

A game of chess pieces

Description automatically generatedA game of chess pieces

Description automatically generatedI also took inspiration from a quote of the famous Chess Grandmaster Bobby Fisher, whose response was the following to an interview question:

Figure 1 Chess 960 Starting Position

Figure 1 Example of a Chess 960 Starting Position

“**Interviewer**: Why do you hate Chess? Being the be… probably, possibly, the best Chess player ever?

**Bobby**: Because I know what Chess all is about! It’s all about memorization. It’s all about pre-arrangement…” (algekalipso, 2022)

In this statement, Fisher emphasized that a significant part of playing chess revolves around memorizing the opening sequences. He heavily criticizes this aspect of the game because it does not involve creative thinking but rather focuses on recalling information. In response Fischer came up with an alternative way of playing chess, where the pieces on the first and last row are shuffled at random. Therefore, the player cannot prepare for the starting position.

This chess variant, known as Fischer Random Chess or Chess960, was introduced in 1996.

Fisher inspired me to mitigate the memorization aspect of the game and focus on creativity.

# Table of Contents

[Preface II](#_Toc155188274)

[Table of Contents III](#_Toc155188275)

[1 Introduction 1](#_Toc155188276)

[1.1 Objectives and Guiding Questions 1](#_Toc155188277)

[1.2 Procedure and Method 1](#_Toc155188278)

[1.3 Structure of the Paper 3](#_Toc155188279)

[2 Initial Chess Code 3](#_Toc155188280)

[2.1.1 Overview 3](#_Toc155188281)

[2.1.2 Logic 3](#_Toc155188282)

[2.1.3 Control Flow 7](#_Toc155188283)

[2.1.4 Error Handling 7](#_Toc155188284)

[2.2 GUI 7](#_Toc155188285)

[2.2.1 Overview 7](#_Toc155188286)

[2.2.2 Visualization Chess Pieces 7](#_Toc155188287)

[2.2.3 Visualization Chessboard 8](#_Toc155188288)

[2.2.4 Update Game after a move 8](#_Toc155188289)

[2.3 Function catalog 8](#_Toc155188290)

[3 Color Chess 10](#_Toc155188291)

[3.1 Rules of Color Chess 10](#_Toc155188292)

[3.2 Implementation of Color Chess 11](#_Toc155188293)

[3.3 Rules of Chaotic Chess 12](#_Toc155188294)

[3.3.1 Barrier 12](#_Toc155188295)

[3.3.2 Shiel 13](#_Toc155188296)

[3.3.3 Coin 14](#_Toc155188297)

[3.3.4 Bomb 15](#_Toc155188298)

[3.4 Implementation of Chaotic Chess 16](#_Toc155188299)

[3.4.1 Overview 16](#_Toc155188300)

[3.4.2 Common code for items 16](#_Toc155188301)

[3.4.3 Barrier 18](#_Toc155188302)

[3.4.4 Shield 19](#_Toc155188303)

[3.4.5 Coin 19](#_Toc155188304)

[3.4.6 Bomb 19](#_Toc155188305)

[4 Summary 20](#_Toc155188306)

[Bibliography IV](#_Toc155188307)

[Table of Figures V](#_Toc155188308)

[Appendix VI](#_Toc155188309)

[A. Appendix 1 VI](#_Toc155188310)

[B. Appendix 2 VII](#_Toc155188311)

[C. Declaration of Authenticity VIII](#_Toc155188312)

# Introduction

## Objectives and Guiding Questions

With my project I aim to develop two games that are variants of the original chess game. The variations I came up with are called “Color Chess” and “Chaotic Chess” and they are played by two players.

I will write a program for each of these game modes, utilizing a pre-established standard chess code. I plan to integrate the rules for each game into the program. Meaning, the computer prevents the user from playing an illegal move. The program should be visually displayed on a screen, using a GUI

With my work I aim to develop two unique games that are variations of chess. My goal is to write two programs for these game modes, utilizing a pre-established standard chess program. I want to integrate the computation of legitimate moves for each of my variants. Therefore, the computer prevents the user from playing any illegal moves. Meanwhile I want my program to be visually displayed on a screen that the user can interact with. The objective of my work did not revolve around the design, that’s why I did not focus heavily on it. The game modes are not compatible on cross platform, rather two users most share the same device to play.

With my project I aim to produce two games that are variants of the original chess game. I will create alternative ways to play chess, by expanding and adapting the rules of chess. I called these games “Color Chess” and “Chaotic Chess” and they are played by two players, just as the original chess. The rules of these games are described later in this paper.

I will develop a program that will implement the rules and visualize the game. The program will check that the players’ moves are legit, i.e. do not break the rules of the game. This in turn requires computation of the valid moves for each chess piece, visual representation of the gameplay using a GUI, and validation of the users’ input to enforce the rules.

In general, chess programs come in two different forms: either they allow two players to play against each other, or one player to play against the computer. I chose the two-player model to avoid having to implement a chess engine capable of generating moves, which would be beyond my skills and experience.

## Procedure and Method

To realize the project, I took an existing chess program, and extended with additional functionality to implement the required rules of the chess variations.

First, I needed to choose a programming language for my project. Since I had little prior knowledge about programming, I opted for an easy to understand yet powerful programming language. Python is one of the most well-known programming languages, commonly used for many different applications, such as web development (server-side), desktop software development, mathematics, system scripting, etc. I started learning Python using a tutorial video I found on YouTube (Bro, 2021). The author explains the basic features of Python and reinforces the information with step-by-step projects. In addition, I also found it helpful to go through the Python Tutorial provided by W3 Schools (W3Schools, s.d.).

Next, I needed to find a chess program written in Python, which I could understand well and extend easily to create new chess variations. I found my way over to GitHub where thousands of programmers share their code with the world. After looking through dozens of chess programs, I finally found one which was easy to extend and covered my requirements [REF].

I decided to manage the source code of my project in GitHub. Using a version control tool, such as GitHub has many benefits: branching and merging for parallel development, detailed code history for tracking changes, and issue tracking for efficient project management. Program source files in GitHub are stored in so called repositories. I have created two new repositories for my project: one repository for each chess variation and imported the original chess code in each one. Now I was ready to make modifications to the original program code and store each developed version safely in the cloud during the entire course of the project.

A screenshot of a computer

Description automatically generated

Figure 2 GitHub Chaotic Chess History

Next, I needed to choose a development environment. Visual Studio Code (or simply Code) is a free integrated development environment, and I have made good experience with this tool previously in the school. It supports a wide range of programming languages with loads of extensions. Specifically, to use Code with Python, the Python extension must be used. I needed to install this Python extension, and the Python interpreter separately. Now I was ready to develop programs in Python, debug and execute the developed programs on my notebook, and also inspect my changes before committing and pushing them to the GitHub repository.

A screen shot of a computer

Description automatically generated

However, I needed to fully understand the original program code before making any changes. When I first inspected the code, it appeared very unclear and was difficult to understand. It contained a lot of variables and functions with a lot of code inside of them. To fully understand the program, I had to go line by line and understand what each variable and function is good for. This took me a long time since the code was written by a more experienced Python developer. Eventually I started understanding the code better and I got ideas how I could adapt it to my needs.

For example, I made use of modules to encapsulate all data and functionality that are logically related in separate source code files. I started with the implementation of one item, the Shield, by creating a module called *shield.py*, and implemented its functionality step-by-step. After several coding and testing iterations, I have managed to make the Shield work completely, and committed the changes to the GitHub repository. I repeated this process for the rest of the items in the same iterative manner. Using modules and developing in small program increments at a time allowed me to limit my attention to a small portion of the whole program, which greatly helped manage the increasing number of interactions in the program.

I made the figures illustrating the various chess positions with the help of chess.com.

## Structure of the Paper

In this paper I will discuss two chess variations that I invented. To implement these variations, I took an existing standard chess program and extended them.

Chapter 2 explains the logic and the key concepts used in the standard chess program. It also describes how the game is visualized on the screen.

Chapter 3 starts with an explanation of the first variation, called “Color Chess”. It continues with description of the implemented solution, i.e. how the rules of the games have been implemented into code.

Chapter 4 covers the second variation, called "Chaotic Chess". Similarly, to the previous chapter, the fist part covers the rules of the game, while the second part focuses on the implementation.

Chapter 5 …

Bibliography, List of figures, appendix.

# Initial Chess Code

## Overview

This section serves to provide a general understanding of how the standard chess code works.

In the beginning, the program generates the necessary components to play chess. To represent the chess board, the program creates an eight-by-eight field of button using the tkinter library. It then proceeds to construct the chess pieces using a class called Figure. The possible moves for the starting position are declared in the Figure objects. Because in chess white starts the game, the turn variable is set to white.

Then, the program waits for an input by the user. The user is asked to select a square from where he wants to move his piece from. Followingly the program checks if a piece of the user is located on the chosen square, if not the user is asked to select a square again. Once the user choses a right square, they are then asked to select a second square where they want to move their piece to. If the piece can move to the position based on the calculated moves or the setup, the move is granted. Then the program updates the position of the pieces and calculates the possible moves from the current board. At the end the program switches the turn variable to be black, indicating that white has finished playing their move.

A screenshot of a computer

Description automatically generated

Figure 3 Inital Chess Code Process

## Concepts

### Figure (or Piece)

The first key concept that needs to be modeled in any chess program is the concept of the chess pieces, also called the figures. There are sixteen pieces of each color, black and white: one king, one queen, two rooks, two bishops, two knights, and eight pawns. Each piece has a type and a position. Each piece has a defined starting position at the beginning of the game. During the game, the players move the white and black pieces in alternate turns, thus the position of the pieces need to be updated. The pieces can move to a square that is either unoccupied or occupied by the opponent’s piece. Each figure needs to know which squares it can move to, based on its type.

In the program, the pieces are modeled with the Figure class. A Class is like an object constructor, or a "blueprint" for creating objects. (W3Schools, s.d.)

A white paper with black text

Description automatically generated

|  |  |  |
| --- | --- | --- |
| Member name | Description | Example |
| name | Represents the type of the figure. | WB |
| object\_name | Unique identifier of the figure. | B1 |
| color | Black or White |  |
| position | The current position of the piece on the board (zero-indexed). | e.g. (0,4) – see Section 2.1.1.3 |
| possible\_moves | The positions to which the piece can move. |  |

The following table describes the valid names (i.e. types), counts (i.e. how many objects of that type exists in the program) and the associated object names:

|  |  |  |  |
| --- | --- | --- | --- |
| Figure Name | Description | Count | Object Names |
| WB | White Pawn  (German: Weisser Bauer) | 8 | B1,B2,B3,B4,B5,B6,B7,B8 |
| WT | White Rook (German: Weisser Turm) | 2 | T1,T2 |
| WS | White Knight (German: Weisser Springer) | 2 | S1,S2 |
| WL | White Bishop (German: Weisser Läufer) | 2 | L1,L2 |
| WD | White Queen (German: Weisse Dame) | 1 | D1 |
| WK | White King (German: Weisser König) | 1 | K1 |
| BB | Black Pawn  (German: Schwarzer Bauer) | 8 | B9-B16 |
| BT | Black Rook (German: Schwarzer Turm) | 2 | T3,T4 |
| BS | Black Knight (German: Schwarzer Springer) | 2 | S3,S4 |
| BL | Black Bishop (German: Schwarzer Läufer) | 2 | L3,L4 |
| BD | Black Queen (German: Schwarze Dame) | 1 | D2 |
| BK | Black King (German: Schwarzer König) | 1 | K2 |

The pieces are laid out on the chess board the following way:

The name of the pieces

In summary, while using the *name* enables the developer to compose logic that depends on the type of the figure, the *object\_name* enables him to reference individual figures unambiguously by a unique name.

#### Chessboard

Chess is played on a square board of eight rows and eight columns. In the program the squares are represented by buttons. Using a grid, the buttons are put in position of an eight-by-eight.

#### Position

The squares of the board and the chess pieces are both arranged into an eight-by-eight grid. However, while the square positions use one-based indexing for the rows and columns (1..8), the pieces use zero-based indexing (0..7). So, when comparing the two grids, one must be subtracted from the row and column position of the squares to correspond to the position of the pieces.

A screenshot of a game

Description automatically generated

Figure 4 Squares use one-based indexing

#### A game of chess with a checkerboard and chess pieces Description automatically generated

Figure 5 Pieces use zero-based indexing

Finally, the buttons representing the squares also have names. These button names are consistent with the names of the squares, as defined in chess, e.g. “a8”.

Throughout the program, conversion routines are used to convert between the various position representations:

1. From square position to piece position, and vice versa. E.g. (6,4) 🡨🡪 (5,3)
2. From square position to button name, and vice versa. E.g. (6,4) 🡨🡪 “d3”

#### Turn

In the game of chess, each player makes a move one after each other in alternate turns.

Turn is a variable that the program uses to determine which player is making the next move. This variable is important to check whether the move played is legitimate. The turn variable can be set to “W” (for white) or “B” (for black). If a legit move has been played, the Turn must change to the other value. If the player does not make a valid move, the program must not change the value of this variable.

#### Capture

When a figure has been captured in chess it is removed from the broad.

In the program, captured pieces are given the position (-1,-1). The user can no longer interact with pieces that are in this position, and they are not considered in further interactions during the game.

#### Check

A check is given when a piece directly attacks the king. If the program finds a legit move that can capture the king with the next move, it detects a check. When a player is in check, they are asked to play a move that escapes the check.

#### Checkmate

Once a king is in check and the program cannot find a legitimate move to escape the check, it is checkmate, and the game is over.

### Control Flow

### Error Handling

A well-written code must deal with its errors. If an error occurs, the program freezes and the user can no longer interact with it. This leaves the user with no idea what caused the problem. To prevent this, the original chess code introduces a variable called “error”. This variable is used to tell the user what problem occurred while interacting with the program. The program continues to run and since the user is informed about the problem, he can avoid it. The error variable can have one of the following four values:

1. the provided move was invalid.
2. the game is over.
3. the king is in check.
4. there is a checkmate

## GUI

### Overview

“A graphical user interface (GUI) is a digital interface in which a user interacts with graphical components such as icons, buttons, and menus. In a GUI, the visuals displayed in the [user interface](https://blog.hubspot.com/website/ui-design?hubs_content=blog.hubspot.com/website/what-is-gui&hubs_content-cta=user%20interface" \t "_blank) convey information relevant to the user, as well as actions that they can take.” (Juviler, 2023)

Why is the GUI needed?

The program uses Tkinter for visualization of the board and the pieces.

“Tkinter is an open source, portable graphical user interface (GUI) library designed for use in Python scripts.” (Dufour, s.d.)

A library is code written by other people that you can import and use in your program. One can create an graphical element by simply using a class in the tkinter library. For instance, to create a button, use the Button class and define its properties.

In the guide that I used to learn Python, I have already met this library, so it was relatively straightforward to use it in my code.

Insert screenshot

### Visualization of pieces

The chess pieces are displayed with Unicode Characters.

“Unicode is a universal character set that defines all the characters needed for writing the majority of living languages in use on computers.” (W3Schools, s.d.)

The Unicode characters are stored within string variables. Every identical chess piece of the same color has the same Unicode. Therefore a total of twelve Unicode characters were needed to visualize all black and white pieces: pawn, knight, bishop, rook, queen and king, for both colors.

To put a chess piece on a square, the underlying button text must be configured with the Unicode character of the piece. See Section 2.2.3

A black background with a black square

Description automatically generated with medium confidence

Figure 7 '|u265C' Unicode character for a black rook

BR = '\u265C'

a8 = Button(tk, text=BR, font='Times 20 bold', bg='white', height=2, width=5, command=lambda: [btnClick(a8), btnID('a8')])

### Visualization of the chessboard

The chessboard is made up of an eight-by-eight field of squares. A square on the board is represented by a button in the program. “A button is a tk widget which is designed for the user to interact with, i.e. if the button is pressed by mouse click, some action might be started. They can also contain text and images like labels.” (Klein, 2022). A grid is used to arrange the buttons horizontally and vertically into a table format, where each button has its own coordinates. Each button is stored in a variable. To create a button, a class named “Button” from the tkinter library is used. To display our button on the board, we must attach it to the grid by specifying its row and column.

a8 = Button(tk, text=BR, font='Times 20 bold', bg='white', height=2, width=5, command=lambda: [btnClick(a8), btnID('a8')])

a8.grid(row=1, column=1)

Background of buttons – chess pattern

### Update screen before the move

### Update screen after the move

To make an ongoing chess game, the program must update the position of each piece. It is also necessary to calculate every possible move that can be played from the current position. And finally, the program needs to display the updated board.

When the user clicks on the piece he wants to move with, the program saves the text of its button. Once the user selects a legitimate button to move the piece to, the program can transfer the saved text to display on the target button.

## Function catalog

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result. (W3Schools, s.d.)

|  |  |
| --- | --- |
| Function Name | Description |
| btnClick |  |
| undo\_coloring | Inverts the highlighting of squares that can be moved to when a figure is selected. |
| btnClick |  |
| btnID |  |
| main | Regulates the entire process of the chess program, besides setting up variables, buttons and the pieces. |
| update\_position |  |
| convert\_to\_figure\_position | Subtracts one from the row and column of the coordinate |
| check\_if\_move\_legit |  |
| update\_all\_possible\_moves | Use the current board to see which moves can be played according to the rules of chess. |
| Try\_helping\_white\_king\_in\_check |  |
| checkinput | Checks whether the move played puts the player who made the move in check. |
| check\_chosen\_move |  |
| print\_board | Prints the current state of the board to the terminal. |
| get\_position | Stores position of each piece on the board. |

# Color Chess

## Rules of Color Chess

In Color Chess, players are assigned an individual color. When their chess pieces move to a square, that square is highlighted with the player's distinctive color. The goal of the game is to color in more squares with your own color than your opponent does with theirs. Only squares on which the pieces stand on or have been standing on are colored in. If a piece captures the opponent’s piece, it will overtake its color. Since each game of chess starts with the same position, the first two rows on each side will be colored in at the beginning. The game ends after 30 moves. The player who has colored in more squares during this period wins. Alternatively, victory can be attained by checkmating the opponent within these 30 moves.

A screenshot of a game

Description automatically generated

Figure 8 Color Chess Starting Position

A screenshot of a game

Description automatically generatedA screenshot of a game

Description automatically generated

Figure 9 Color Chess Piece Capture Example Prior

Figure 10 Color Chess Piece Capture Example After

## Implementation of Color Chess

At the start of the program, the user is asked to choose two different colors from a color panel. These two colors are then stored within variables. The first color selected is used to color the squares where the white pieces move to, and the second color is used for the black player.

Afterwards, the starting position of color chess is set up [see figure 7]. A button resembles the square on the chess board. To change the properties of a button “button.config” function is used within the tkinter library. Chess always begins with the same setup: white pieces on rows one and two, and black pieces on rows seven and eight. To color the correct square on the board at the start, I instructed the program the following: if a button has the row one or two, it should be colored with white’s color. If a button has the row seven or eight, it should be colored with black’s color.

Next, I needed to make sure that the squares on which the pieces move are colored with the right color. Using “button.config” once again, I can color the selected square. To determine which player is making the move, I created a variable that counts the turns. The turn variable I set to zero at the beginning and increases by one after every legitimate move. So, if the number of the count of turns is odd the blacks color is used. If it is even, whites color is used to color in the squares.

To display the score on the screen the Label class in the tkinter library is used. This Label requires two variables to show the score: The score of white and the score of black.

The following instances can occur if a piece moves to a square:

|  |  |  |
| --- | --- | --- |
| Condition | Own Score | Opponents Score |
| The piece moves to an uncolored square | +1 | +0 |
| The piece moves to his own-colored square | +0 | +0 |
| The piece moves to opponents colored square | +1 | -1 |

To determine what color square the piece steps on, we can use “endbutton\_color". The endbutton\_color is the color of the square the user wants to move his piece to.

Since I only want the coloring and scoring the happen if the move was legit, I put everything within the condition of check\_chosen\_move.

Here is the code from white’s side:

if check\_chosen\_move(startbutton\_position, endbutton\_position):

      if count\_turn % 2 == 0:

        if endbutton\_color == "white" or endbutton\_color == "grey":

          WHITE\_SCORE += 1

        elif endbutton\_color == BLACK\_COLOR:

          BLACK\_SCORE -=1

          WHITE\_SCORE += 1

        endbutton\_color = WHITE\_COLOR

      count\_turn +=1

To display the round that have been played so far, I created a Label that displays the count\_turn variable as a string.

## Rules of Chaotic Chess

Chaotic Chess introduces four items that the chess pieces can pick up, granting them special abilities. To make the game balanced, the items are equally distributed on empty fields on the white side (rows one to four) and on the black side (rows five to eight). A piece can pick up an item by stepping on its according square. The items change their positions every four moves successively. The game comes to an end if either king is in checkmate.

The four items consist of a bomb, shield, coin and barrier.

A screenshot of a game

Description automatically generated

Figure 11 Chaotic Chess Random Position Example

### Barrier

The square that the barrier is placed on, prevents pieces to step onto that field. Only the field that the barrier is placed on is affected, pieces can jump over the barrier.

A screenshot of a game

Description automatically generated

Figure 12 Chaotic Chess Barrier Usage Example Prior

A screenshot of a game

Description automatically generated

Figure 13 Chaotic Chess Barrier Usage Example After

### Shiel

The shield makes the piece that steps on it invincible until the shield changes its position. Anchored to its square, the shield does not move with the piece that picked it up.

A screenshot of a game

Description automatically generated

Figure 14 Chaotic Chess Shield Usage Example Prior

A screenshot of a game

Description automatically generated

Figure 15 Chaotic Chess Shield Usage Example After

### Coin

The shield makes the piece that steps on it invincible until the shield changes its position. Anchored to its square, the shield does not move with the piece that picked it up.

A screenshot of a game

Description automatically generated

Figure 16 Chaotic Chess Coin Usage Example Prior

A screenshot of a game

Description automatically generated

Figure 17 Chaotic Chess Coin Usage Example After

### Bomb

Picking up the bomb triggers an explosion in a three-by-three area. Pieces within that area are destroyed and removed from the board. The piece that initially stepped on the bomb is eliminated as well. If a bomb is near the edges or corners of the board, the explosion radius only takes up as much space as it is granted.

A screenshot of a game

Description automatically generated

Figure 18 Chaotic Chess Bomb Usage Example Prior

A screenshot of a game

Description automatically generated

Figure 19 Chaotic Chess Bomb Usage Example After

## Implementation of Chaotic Chess

### Overview

#### Modules

“Consider a module to be the same as a code library. A file containing a set of functions you want to include in your application.” (W3Schools, s.d.)

To make my program easier to maintain and easier to read, I have put the code of each item into a separate module.

Importing the four modules (barrier.py, bomb.py, coin.py, shield.py) into the initial chess code allows you to use the functions contained in the modules. However, it is not allowed to use functions from the initial code in the modules.

import bomb

import coin

import shield

import barrier

#### Differences In Chess Code

#### Usage Of Initial Chess Code

The item differs in the code that they execute when they are picked up. Otherwise they share similar code.

### Common code for items

#### Placing an item

To find a location where the item can be placed on, we need to randomly select a square on the board.

In the result of developing Chaotic Chess in a balanced way, I had to generate the items equally on both sides of the board. To accomplish this, I split the board in half and declared two list of buttons, representing the squares of the board on each side. Buttons on the white side (row: one to four) and buttons on the black side (row: five to eight).

A chess board with chess pieces

Description automatically generated

With these two lists, I can write a program that randomly selects a button from each list. Leaving us with two buttons that our item can be placed on. However, there is a problem if the code selects a button which is already occupied by a chess piece. This would lead to overwriting the button and deleting the piece. To prevent this, I had to tell the program to select a different button when it is occupied. To determine whether a button is occupied, I used a list that was already included in the original chess code. “Player\_pos\_list” contains the positions of all chess pieces on the board. Using this list, I created a loop in which the program randomly selects a button until it finds an unoccupied one. Once it finds a suitable button, it should then store its variable, since we will need to come back to it.

def placeShield(button\_list, players\_pos\_list, endbutton\_pos, shieldText, shield\_button):

removeShieldIfExist(shield\_button)

# find an empty field to place the shield

while True:

shield\_button = random.choice(button\_list)

shield\_pos = getButtonPosition(shield\_button)

if shield\_pos not in players\_pos\_list and shield\_pos != endbutton\_pos:

shield\_button.config(text=shieldText, fg = "red")

break

return shield\_button

#### Visualization of an Item

To let the user know where the item has been placed, it needs to be displayed on the screen.

For the visualization of the items I chose to use Unicode characters. Displaying the item onto a square, works the same way as with the pieces, i.e. by configuring the text of the relevant button. Another advantage of using Unicode is that it can be colorized. I colored the items the following way:

|  |  |  |  |
| --- | --- | --- | --- |
| Item | Color | Unicode | Unicode |
| Barrier | Purple | \U0001F5D9 | A black background with a black square  Description automatically generated with medium confidence |
| Shield | Red | \U0001F6E1 | A black background with a black square  Description automatically generated with medium confidence |
| Bomb | Green | \U0001F4A3 | A black background with a black square  Description automatically generated with medium confidence |
| Coin | Yellow | \* | \* |

\*The coin uses the Unicode character of the relevant chess piece

#### Picking up an item

Picking up an item means, that the player moves a chess piece to the square where the item is located.

The code needs to identify which item has been picked up, to execute the code written for that specific item. To do this…

#### Change position of an item

By changing the position of an item, I want to make the game more random and chaotic. To make this happened I had to delete the original items place if it already existed and choose another position for it. An important feature to implement is that if the item has already been picked up it should not delete its former place since the player already removed the item from the board by picking it up. However, this code does not apply to the barrier, since it’s impossible to pick up the barrier and we always have to delete it’s former position. To delete the former item, we can use once again “button.config” function. By setting the text

### Barrier

The check\_chosen\_move function checks if the chosen move is valid.

If the endbutton and the barrier button are the same button, that means the user tries to move to the barrier. To prevent a piece to move on the barrier, I used the “elif” statement inside of the check\_chosen\_move function.

“The elif keyword is pythons way of saying ‘if the previous conditions were not true, then try this condition’.” (W3Schools, s.d.). With this logic, we can tell our program if the user steps on the barrier it should set error to “1” and return check\_chosen\_move to be “False”. Subsequently, the check\_chosen\_move function will no longer be executed and the error variable is not overwritten by the rest of the function.

def check\_chosen\_move(ps, pe):

global error

if barrier.isBarrier1Set(pe) or barrier.isBarrier2Set(pe):

error = 1

return False

#rest of check\_chosen\_move code

### Shield

If a Shield is picked up, it should function similarly than the Barrier: Pieces cannot capture the piece protected by the Shield. Therefore, we can use the same logic that is used in the previous function, with the exception that pieces can move to the Shield if it has not been picked up.

### Coin

### Bomb

# Summary

# Bibliography

Metzger, P. (2010). *Abschlussarbeiten.* Aarau: Sauerländer.

Bonati, P., & Hadorn, R. (2009). *Matura- und andere selbständige Arbeiten betreuen* (2. Auflage ed.). Bern: Heb Verlag AG.

*Panorama*. (2015, December 15). Retrieved December 6, 2019, from Statistik Schweiz - Panorama: http://www.bfs.admin.ch/bfs/portal/de/index/themen/01/01/pan.html

algekalipso. (25. July 2022). *Qualia Computing*. Von Qualia Computing: https://qualiacomputing.com/2022/07/25/bobby-fischer-1972-world-chess-champion-on-why-chess-is-a-lousy-game-and-how-to-save-it/#:~:text=Bobby%20Fischer%3A%20No%2C%20no%2C,I%20don't%20need%20it. abgerufen

*W3Schools*. (kein Datum). Von https://www.w3schools.com/python/python\_classes.asp abgerufen

*W3Schools*. (kein Datum). Von https://www.w3schools.com/python/python\_functions.asp abgerufen

*W3Schools*. (kein Datum). Von https://www.w3schools.com/charsets/ref\_html\_utf8.asp abgerufen

Klein, B. (1. 2 2022). *python-course*. Von https://python-course.eu/tkinter/buttons-in-tkinter.php#:~:text=The%20Button%20widget%20is%20a,text%20and%20images%20like%20labels. abgerufen

*W3Schools*. (kein Datum). Von https://www.w3schools.com/python/python\_modules.asp abgerufen

*W3Schools*. (kein Datum). Von https://www.w3schools.com/python/default.asp abgerufen

Bro, C. (15. 2 2021). *Youtube.* Von https://www.youtube.com/watch?v=XKHEtdqhLK8&t=23528s&ab\_channel=BroCode abgerufen

Dufour, B. (kein Datum). Von https://www.cs.mcgill.ca/~hv/classes/MS/TkinterPres/ abgerufen

Dufour, B. (kein Datum). Von https://www.cs.mcgill.ca/~hv/classes/MS/TkinterPres/ abgerufen

*W3Schools*. (kein Datum). Von https://www.w3schools.com/python/gloss\_python\_elif.asp abgerufen

# Table of Figures

[Figure 1 Chess 960 Starting Position II](file:////Users/attila/Documents/GitHub/matura-arbeit/matura-paper-template.docx#_Toc155188164)

[Figure 2 GitHub Chaotic Chess History 2](#_Toc155188165)

[Figure 3 This is the grid used by the buttons. 5](#_Toc155188166)

[Figure 4 This is the grid used by the pieces. 6](#_Toc155188167)

[Figure 5 The pieces are named this way. 6](#_Toc155188168)

[Figure 6 '|u265C' Unicode character for a black rook 8](#_Toc155188169)

[Figure 7 Color Chess Starting Position 10](#_Toc155188170)

[Figure 8 Color Chess Piece Capture Example Prior 10](#_Toc155188171)

[Figure 9 Color Chess Piece Capture Example After 10](#_Toc155188172)

[Figure 10 Chaotic Chess Random Position Example 12](#_Toc155188173)

[Figure 11 Chaotic Chess Barrier Usage Example Prior 12](#_Toc155188174)

[Figure 12 Chaotic Chess Barrier Usage Example After 13](#_Toc155188175)

[Figure 13 Chaotic Chess Shield Usage Example Prior 13](#_Toc155188176)

[Figure 14 Chaotic Chess Shield Usage Example After 14](#_Toc155188177)

[Figure 15 Chaotic Chess Coin Usage Example Prior 14](#_Toc155188178)

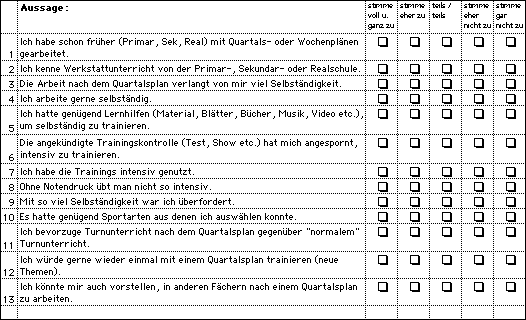
[Figure 16 Chaotic Chess Coin Usage Example After 15](#_Toc155188179)

[Figure 17 Chaotic Chess Bomb Usage Example Prior 15](#_Toc155188180)

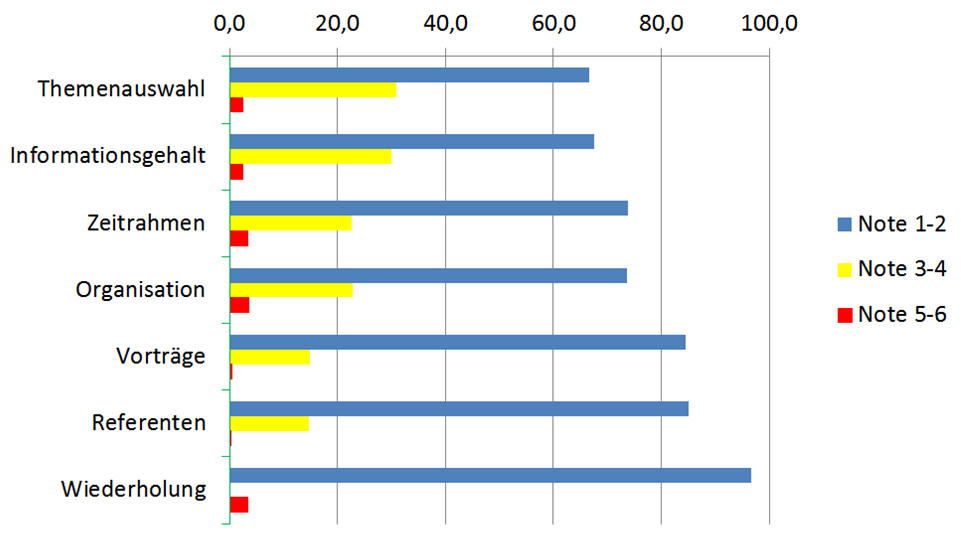
[Figure 18 Chaotic Chess Bomb Usage Example After 16](#_Toc155188181)

# Appendix

## Appendix 1

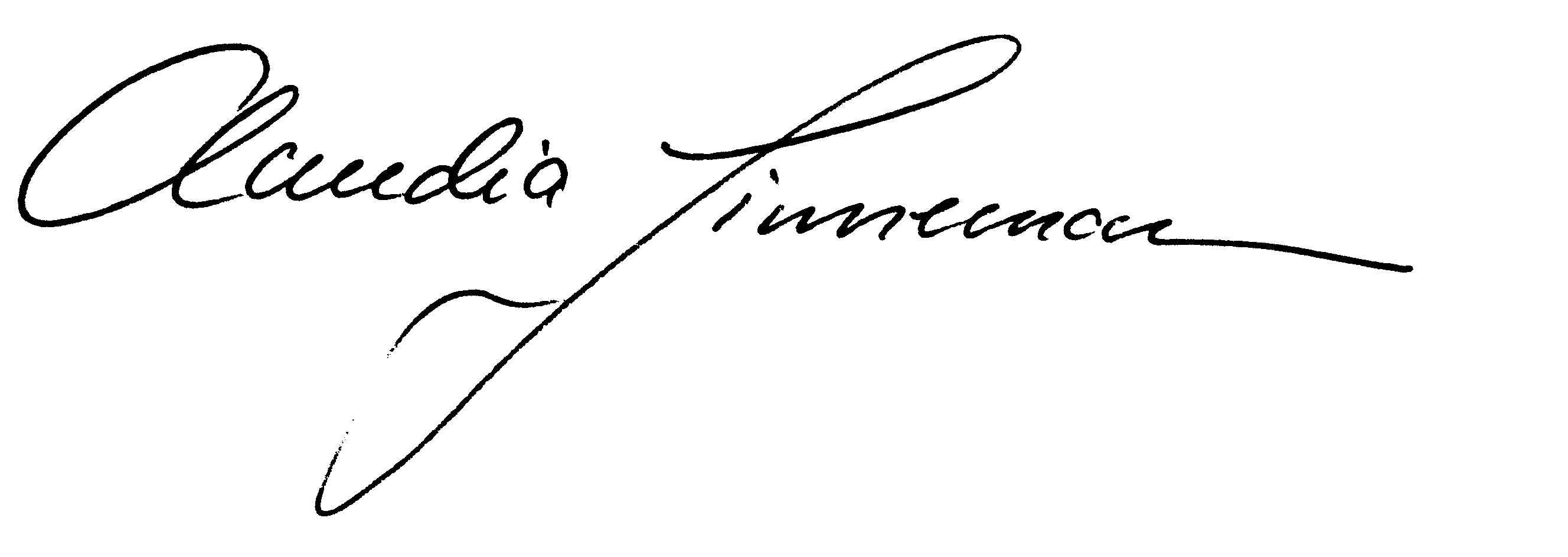


## Appendix 2



## Declaration of Authenticity

I hereby declare that the work submitted is my own and that all passages and ideas that are not mine have been fully and properly acknowledged.



Mels, 6.1.2021