**Creating 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

„**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](#_Toc155380079)

[Table of Contents III](#_Toc155380080)

[1 Introduction 1](#_Toc155380081)

[1.1 Objectives and Guiding Questions 1](#_Toc155380082)

[1.2 Procedure and Method 1](#_Toc155380083)

[1.3 Structure of the Paper 3](#_Toc155380084)

[2 Standard Chess 4](#_Toc155380085)

[2.1 Concepts 4](#_Toc155380086)

[2.1.1 Figure (or Piece) 4](#_Toc155380087)

[2.1.2 Chessboard 6](#_Toc155380088)

[2.1.3 Position 6](#_Toc155380089)

[2.1.4 Turn 7](#_Toc155380090)

[2.1.5 Capture 7](#_Toc155380091)

[2.1.6 Check 7](#_Toc155380092)

[2.1.7 Checkmate 7](#_Toc155380093)

[2.2 Process Description 8](#_Toc155380094)

[2.3 Control Flow 10](#_Toc155380095)

[2.3.1 Error Handling 10](#_Toc155380096)

[2.4 GUI 11](#_Toc155380097)

[2.4.1 Overview 11](#_Toc155380098)

[2.4.2 Visualization of pieces 11](#_Toc155380099)

[2.4.3 Visualization of the chessboard 12](#_Toc155380100)

[2.4.4 Update screen before the move 12](#_Toc155380101)

[2.4.5 Update screen after the move 13](#_Toc155380102)

[2.5 Functions Catalog 13](#_Toc155380103)

[2.5.1 Functions 13](#_Toc155380104)

[2.5.2 Object Methods 14](#_Toc155380105)

[3 Color Chess 15](#_Toc155380106)

[3.1 Rules of Color Chess 15](#_Toc155380107)

[3.2 Implementation of Color Chess 16](#_Toc155380108)

[4 Chaotic chess 18](#_Toc155380109)

[4.1 Rules of chaotic chess 18](#_Toc155380110)

[4.1.1 Rules for Barrier 19](#_Toc155380111)

[4.1.2 Rules for Shield 19](#_Toc155380112)

[4.1.3 Rules for Coin 20](#_Toc155380113)

[4.1.4 Rules for Bomb 20](#_Toc155380114)

[4.2 Implementation of Chaotic Chess 21](#_Toc155380115)

[4.2.1 Overview 21](#_Toc155380116)

[4.2.2 Modules 21](#_Toc155380117)

[4.2.3 Common code for items 22](#_Toc155380118)

[4.2.4 Barrier 23](#_Toc155380119)

[4.2.5 Shield 24](#_Toc155380120)

[4.2.6 Coin 24](#_Toc155380121)

[4.2.7 Bomb 25](#_Toc155380122)

[4.2.8 Integrating the Items into the Original Chess Program 25](#_Toc155380123)

[5 Summary 27](#_Toc155380124)

[Bibliography IV](#_Toc155380125)

[Table of Figures V](#_Toc155380126)

[Appendix VI](#_Toc155380127)

[A. Appendix 1 VI](#_Toc155380128)

[B. Declaration of Authenticity VII](#_Toc155380129)

# Introduction

## Objectives and Guiding Questions

With my project I aim to develop two games that are variations of the original chess game. The variations I came up with are called “Color Chess” and “Chaotic Chess”. The rules of these games are described later in this paper.

I will write a program for each of these game modes, utilizing an existing standard chess code as a foundation. The programs will verify that the players’ moves are valid i.e. do not break the rules of the game. This in turn requires computation of the valid moves for each chess piece.

The gameplay shall be visually displayed using a GUI.

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 (Data, Default, 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 (j00nas, 2020).

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

Figure 3 Debugging in VS Code

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](#_Initial_Chess_Code) 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](#_Color_Chess) 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](#_Chaotic_Chess) covers the second variation, called "Chaotic Chess". Similarly, to the previous chapter, the first part covers the rules of the game, while the second part focuses on the implementation.

In [Chapter 5](#_Summary), I summarize the results of my project and provide my concluding thoughts on the work. I also mention what I learned during my project.

The paper concludes with the Bibliography, Table of Figures, and Appendix.

# Standard Chess

In this chapter, I discuss the standard chess program I used as a starting point for my project.

This includes the key concepts of a chess game and how they are modeled in the program (Section 2.1). I provided a process model for the gameplay In Section 2.2. Section 2.3 explains the control structure used to implement the process in the program. Section 2.4 describes the user interface, i.e. how the program displays the game on the screen and how the user can interact with the game.

## Concepts

In software engineering, various programming constructs are used to model real world objects, their behavior, and the relationships between them. Programs make use of abstractions to represent the inherent concepts (i.e. “things”) of the problem domain in which they operate. In the context of this paper, the problem domain is that of the chess game, thus the program works with those concepts inherent in chess.

A chess set consists of black and white chess *pieces* and a *chessboard* (hereafter simply referred to as the *board*). Keeping track of the progress of the game requires knowing the *position* of each piece and their *possible moves* in each *turn*, according to the rules of chess. This in turn requires recognizing if a piece has been *captured*, and whether the king has been attacked (*check*). Finally, it must be detected if the king cannot escape the check (*checkmate*).

### 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.” (Data, Python Classes, s.d.)

A white paper with black text

Description automatically generated

Figure 4 Class Definition of Figure

The following table describes the properties of Figure:

|  |  |  |
| --- | --- | --- |
| 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). | (6,0)  see Section 2.1.3 |
| possible\_moves | The positions to which the piece can move. | [(5,0), (4,0)] |

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 |

Tabel 1

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

#### A green and white checkered board with numbers and letters Description automatically generated

Figure 5 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. The board is divided into 64 fields, also called squares.

In the program the squares are represented by buttons. Using a grid, the buttons are aligned in the structure of the eight-by-eight board. This enables the program to accept user inputs by providing event handler code for mouse button click events.

### 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 6 Squares use one-based indexing

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

Figure 7 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.

## Process Description

This section provides a general model of the standard chess game process.

First, the board and the pieces are initialized in a preparation step. Then, the program waits for input from the user. The white player needs to make the first move by selecting a white piece on the board. The program checks if the user selected a white piece – if not, the user is asked to try again. Next, the user is asked to select a square to which they would like to move their piece. If the piece is allowed to move to the selected square according to the rules of chess, the move is granted, otherwise the user needs to start again and select a piece. Then, the program updates the position of the selected piece. If the selected square was occupied by an opponent piece, it will be removed from the board. The program then calculates the possible moves of every piece on the board and verifies the check and checkmate conditions. In case of checkmate, the game ends. Otherwise, the turn is switched to black, and the process starts again. This process continues by alternating the turn until one checkmate is detected.

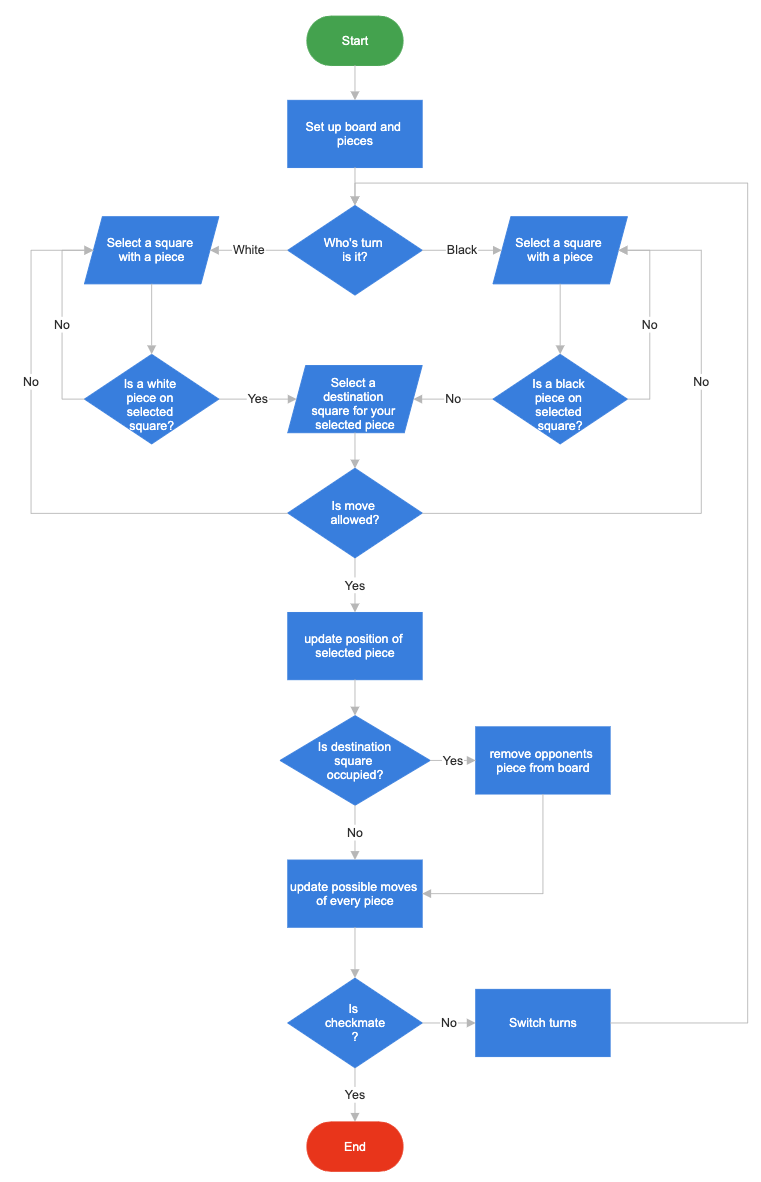


Figure 8 Standard chess code process

## Control Flow

The standard chess program is a standalone desktop GUI application that runs on the user’s computer. The program waits for user inputs (mouse click events) and reacts by updating the user interface and its internal state. This mode of operation largely determines the flow of control within the program. This means, that the program must be able react to user events any time by handling those events. Handling the event means executing a set of program statements, which ultimately leads to the expected program output (e.g. move a piece to a different position, detect checkmates, etc).

So, in the program, mouse click events trigger the execution of event handling code, which in turn generates the desired output. For each button representing a square on the board, two functions have been defined as event handlers: btnClick() and btnId(). These functions call other functions internally which together implement the event handling logic. The flow of control between these functions is shown in Figure 8. A description for each function can be found in Section 2.5.

A blue rectangular boxes with white text

Description automatically generated

Figure 9 Control Flow Diagram

Another interesting detail is that regarded from a logical point of view, mouse click events must be processed in pairs:

1. The first click is used to select a piece on the board.
2. The second click is used to select a target square, where the piece should be moved to.

The event handling code takes this into account and contains different logic for the first and the second click events. This will be also important to consider later when implementing the processing logic for the items.

### 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

The program uses a Graphical User Interface (GUI) to visualize the game. The GUI displays the chessboard, the pieces, and various messages to the user. The GUI also enables the user to interact with the game in a natural way: using a mouse, the user can select a piece on the board, and make a move by selecting a target square. If the move was valid, the display is updated to reflect the new state of the game.

There are several different GUI libraries that can be used in Python, but Tkinter is the most common among them and the only one that is part of the standard library:

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

In the tutorials that I used to learn Python this library was explained quite well, so it was straightforward for me to understand how to use it in my code.

A screenshot of a game

Description automatically generated

Figure 10 Game visualization with the Tkinter GUI library

### Visualization of pieces

The symbols of the pieces are represented 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.” (Data, Charsets, s.d.)

Chess symbols are part of Unicode. In the program, 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.

A black background with a black square

Description automatically generated with medium confidence

Figure 11 Unicode character '|u265C' 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 the button on the board in each location, it must be attached to a grid by specifying its row and column.

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

In chess the squares of the board are colored in a gray and white tile pattern, as shown in Figure 12. To represent this pattern, the program sets the background color of the buttons alternately either to white or gray.

Button(tk, bg='white') or Button(tk, bg='grey')

A screenshot of a computer screen

Description automatically generated

Figure 12 Chess board set up by buttons

### Update screen before the move

The program displays the possible moves of the selected piece so that the player can see which moves are available for that piece. The available moves are visualized by placing green dots on each valid target square. This is implemented by setting the text of the buttons representing the valid target squares with the Unicode character of the dot.

A screenshot of a computer game

Description automatically generated

Figure 13 Visualization of available moves

### Update screen after the move

After the move has been validated, the selected piece is moved to the target square, and the program displays the updated board. Displaying the move is implemented in two steps:

1. Clear the text of the button representing the source square (where the piece is moving away from)
2. Set the text of the button with the figure symbol representing the target square (where the piece is moving to)

It is necessary to save the text of the source button when the user selects a piece, so that the program can reference this text in the second step when setting the text of the target button. The program saves this text in a dedicated program variable, called “start\_text”.

## Functions Catalog

### Functions

|  |  |
| --- | --- |
| Function Name | Description |
| btnClick | Event handler, checks if move was valid on second mouse click. |
| btnID | Event handler, displays possible moves after first mouse click by displaying green dots on the valid squares. Highlights the square of the selected figure. Displays the piece in the new position after the move. Shows the user the king in check or checkmate. |
| undo\_coloring | Restores the squares after a move by removing the green dots |
| main | Coordinates the flow of control in the program. |
| checkinput | Moves the figure to the target square. Updates possible moves of all figures. Verifies the rules as a consequence of the move (check, checkmate). Switch turns. |
| check\_chosen\_move | Verifies if the player’s move is valid. |
| print\_board | Prints the current state of the board to the terminal. |
| get\_positions | Stores the position of each piece on the board in variables for further processing. |

### Object Methods

#### Figure

|  |  |
| --- | --- |
| Function Name | Description |
| update\_position | Moves the figure to the target position |
| return\_position | Returns figure to the start position |
| update\_possible\_moves | Calculates every position that the figure can move to, considering the current position of all other figures in the game |
| check\_if\_move\_legit | Determines if the requested move is valid, i.e. whether the figure can move to its chosen target position |

# Color Chess

This chapter describes the first chess variation, Color Chess. In the following figure an examples gameplay is shown.

A screenshot of a game

Description automatically generated

Figure 14 Color chess

## Rules of Color Chess

In Color Chess, players are assigned an individual color. When their pieces move to a square, that square is highlighted with the player's color. The goal of the game is to color in more squares with your own color than your opponent does with theirs. Only occupied squares are colored. If a piece moves away from a square, the square remains colored. 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 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 15 Starting position in Color Chess

A screenshot of a game

Description automatically generatedA screenshot of a game

Description automatically generated

Figure 16 Capturing pieces in Color Chess

## Implementation of Color Chess

At the start of the game, 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 (Figure 15). Chess always begins with the same setup: white pieces on rows one and two, and black pieces on rows seven and eight. The coloring of the squares on the board at the start is implemented the following way:

* If a button is in row one or two, set its background to white’s color.
* If a button is in row seven or eight, set its background to black’s color.
* Otherwise, leave the button background unchanged.

Next, I needed to make sure that each time a piece is moved to a square, the color of that square is set with the player’s color. To determine which player is making the move, I created a variable that counts the turns. I set the turns count variable to zero at the beginning of the game and increase it by one after every legitimate move. So, if the turns count is odd, the black player’s color is used. If it is even, the white player’s color is used.

The turns count variable is needed to end the game after 40 moves. It is also convenient to use it to check which player made the move, although the turn variable (See Turn) could have been also used for this purpose.

To display the score on the screen the Label class in the Tkinter library is used. Two variables are required to show the score: The score of the white and the score of the black.

The following cases 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 |

Table 2

To determine what color square the piece steps on, I used a variable called “endbutton\_color". This variable already existed in the initial chess code. It holds the color of the square the user wants to move his piece to. If the endbutton\_color is white or gray, it should increase the score by one. If it is the opponents color, it should increase its own score by one and decrease the opponents score by one. Since the scores don’t change if the piece moves to his own-colored square, we don’t have to include it in the if statements:

Before coloring and scoring, the program checks if the move was valid:

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 rounds, it must be first converted to text, because a label only accepts string parameters.

# Chaotic chess

In this chapter, I will discuss my second chess variation, called Chaotic Chess. The following figure shows an example gameplay.

A screenshot of a game

Description automatically generated

Figure 17 Chaotic chess

## Rules of chaotic chess

Chaotic Chess introduces four different types of items that appear randomly on the board and the pieces can pick up, granting the pieces 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 the square occupied by the item.

The four items are called the bomb, the shield, the coin and the barrier. At the beginning of the game, there are no items on the board. After the first move, two shields appear on the board – one shield on the white side, and one shield on the black side. After the second move, two bombs appear. After the third move, two coins appear. After the fourth move, two barriers appear. Afterwards the items change their positions every four rounds, successively until the end of the game. The game ends if either king is in checkmate.

A screenshot of a game

Description automatically generated

Figure 18 Items During Play in Chaotic Chess

The following sections describe the behavior of each item and the rules governing them.

### Rules for Barrier

The square that the barrier is placed on, prevents pieces to step onto that square. Only the square that the barrier is placed on is affected, pieces can jump over the barrier. If a player tries to step onto a barrier, the move is prohibited, and the player must choose a different move.

A screenshot of a game

Description automatically generatedA screenshot of a game

Description automatically generated

Figure 19 Use case Barrier

In this case, white plays a tactical move that traps black’s queen. Since pieces cannot move onto the barrier, black has no way of defending their queen. Therefore, black plays pawn to a6, and is forced to give up their queen.

### Rules for Shield

The shield makes the piece that steps on it “protected”, i.e. the piece on the shield cannot be captured. When the shield changes its position, the piece can be captured again. Anchored to its square, the shield does not move with the piece that picked it up. If the piece moves away, the piece will be no longer protected. If the opponent tries to capture a piece on the shield, the move is prohibited, and the player must choose a different move.

A screenshot of a game

Description automatically generatedA screenshot of a game

Description automatically generated

Figure 20 Use case Shield

This case shows how the white queen picks up the shield in order to be protected from black knight. This enables the white queen to attack without getting captured for a few rounds.

### Rules for Coin

The coin can resemble three different pieces: knight, bishop and rook. The piece that steps onto the coin, is swapped with the piece which the coin resembles.

A screenshot of a game

Description automatically generatedA screenshot of a game

Description automatically generated

Figure 21 Use case Coin

This case demonstrates white picking up the coin with one of his pawns, thus swapping out the weaker piece for a bishop. This way, the player can acquire stronger pieces.

### Rules for Bomb

Picking up the bomb triggers an explosion in a three-by-three area. Pieces within that area are destroyed and removed (captured) 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 gets limited to the edge of board. The bomb explodes only after the 3rd move, after a piece picked it up. This gives both players an opportunity to move one or two pieces outside the explosion area, thus “rescue” them.

A screenshot of a game

Description automatically generatedA screenshot of a game

Description automatically generated

Figure 22 Use case Bomb

In this case white decides to pick up the bomb in the center of the board. After the explosion, all the pieces in the explosion area are destroyed: a black pawn, a black bishop and two white pawns. Consequently, white gains the advantage over the position.

## Implementation of Chaotic Chess

### Overview

Implementing Chaotic Chess turned out to be more difficult, than Color Chess, for two reasons:

1.     Chaotic Chess has more functionality (four items, each with distinct rules).

2.     The logic that implements the item rules is more complex and requires more interactions between the various parts of the program.

To make the program easier to maintain and easier to read, I have organized the code of each item into a separate module (Section 4.2.2). During development I have recognized that the four item implementations share some commonalities, which are described in Section 4.2.3. The sections 4.2.4 through 4.2.7 discuss the implementation of each individual item. Finally, Section 4.2.8 describes the changes I made to integrate the items into the original chess program.

### 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.” (Data, Python Modules, s.d.)

The individual items work independently of each other, consequently their implementation should also have no dependency on each other. This means, that changing the functionality of one item should not affect the behavior of any other item. This is of great benefit, because with no dependency between the different item implementations, the risk of side-effects is much lower, simplifying testing and troubleshooting procedures.

Therefore, I separated the implementation of each item into distinct modules. Since no item module imports anything (function, data variable, etc.) from another item module, they are truly independent, thus easier to read and maintain.

### Common code for items

The different items have the same underlying mechanism for their operation, which is discussed next.

#### Placing an item

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

To generate the items equally on both sides of the board, I have split the board in half with the help of two buttons lists, 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

Figure 23 White and black side buttons

Using these two lists, I was able to write a function that randomly selects a square on each side of the board. However, there is a problem if a square gets selected which is already occupied by a piece. This would lead to overwriting the button and destroying the piece. To prevent this, I needed to check if the selected square is occupied and tell the program to select a different button in such cases. To determine whether a square 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. I created a loop which runs until an unoccupied square is found.

Once a suitable square is found, the item is placed by setting the text property of the corresponding button to the item symbol ([see Section 4.2.2.2](#_Visualization_of_the)). Finally, the button is stored in a dedicated variable since we will need to reference it later.

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 the items

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. The following table summarizes the visualization properties of the items:

|  |  |  |  |
| --- | --- | --- | --- |
| Item | Color | Unicode | Symbol |
| 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 | \* | \* |

Tabel 3

*\*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 piece to the square where the item is located.

Each time a player makes a move, the program needs to check if the piece is moved to a square occupied by an item and identify which type of item has been picked up, so that it can execute the pick-up logic of that specific item. To do this, the program simply compares the button found at the target position of the move with the dedicated button variables created earlier, when placing the items (Section 4.2.3.1). If there is a match, the respective pick-up function is called for the matched item. Finally, the move is made, i.e. the position of the piece is updated, and all possible moves of all pieces are re-calculated.

#### Changing position of an item

To change the position of an item, it must be first removed from the board, and then recreated in another position. To achieve this, I had to delete the item first and then find a new position for it. An important detail to consider was that if the item has been already picked up it should not be removed, since the piece already removed the item by picking it up. However, this logic does not apply to the barrier since it is impossible to pick up a barrier. To delete the original item from the board, the text of the underlying button is set to empty. This is another place where the dedicated button variables created earlier are used.

### Barrier

The only two requirements for the barrier are the following:

1. Placing the barrier: each four rounds, the program must remove the existing barriers and place two new barriers on the board.
2. Picking up the barrier: The program must stop any piece from stepping onto the barrier.

The check\_chosen\_move function validates every move. In addition, this function should also check if a piece stepped on the barrier. This is accomplished by checking if the endbutton is equal to the barrier button. If yes, the error variable is set to “1” (the provided move was invalid, see Section 2.3.1) and returns “False”. The user is asked to make a different move.

def isBarrier1Set(endbuttonPosition):

return is\_barrier\_1\_set and endbuttonPosition == getButtonPosition(

barrier\_field\_button\_1)

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

global error

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

error = 1

return False

### Shield

The functionality of the shield is similar to the barrier: pieces cannot move to the square occupied by protected piece. However, picking up the shield has different logic. The requirements of the shield are:

1. Placing the shield: same as the barrier.
2. Picking up the shield: The piece must be marked as protected.
3. Step away from shield: Remove shield from piece.
4. Capture validation: The opponent cannot capture the protected piece.

When a piece picks up the shield, a variable called is\_Shield\_1\_Set (or is\_Shield\_2\_Set) is set to true. If another piece tries to move to the shield protected square while this variable is true, the move is prohibited. This check is performed from within the check\_chosen\_move function, identical to the barrier. Removing the shield from the piece is simply implemented by setting the is\_Shield\_1\_Set (or is\_Shield\_2\_Set) variable to false.

The program also displays the rounds left until the shields are placed again.

### Coin

The implementation of this item required deeper understanding of the existing chess code, because changing the type of the pieces had side-effects in other parts of the program.

To swap a piece with a different one, the following properties must be changed: the symbol, the name (figure type) and the possible moves of the piece. However, it is important to make the changes at the right time to avoid side-effects. For the coin to work properly, the name of the figure must be updated before the program updates the possible moves. Otherwise, the possible moves would have been calculated for the originally moved piece, which is incorrect.

The coin requirements are the following:

1. Placing the coin: each four rounds, the coins are removed and two new coins are placed on the board.
2. Picking up the coin: after each valid move (round), the program checks if a player moved a piece onto a square occupied by a coin. If yes, the coin is picked up and the piece is transformed. Finally, the program calculates the possible moves of the new piece.

When a piece picks up the coin, it transforms into a knight, bishop, or rook. The symbols of these three pieces are put into a list. Using the random library, the program makes a random choice from this list. Once a piece picks up the coin, the program must determine its color. This is necessary because the program should transform the coin into the correct color. For instance, if a white pawn picks up a bishop coin, the pawn should change into a white bishop. The turn variable (Section 2.1.4) is used for this purpose.

.

### Bomb

The bomb is a special item that requires special treatment for two reasons:

* the bomb does not explode immediately after the player has picked it up
* picking up a bomb just sets the bomb off for explosion three rounds later; consequently, there can be more than two bombs on the board at the same time (although there can be only maximum two that are not picked up).

The bomb has the following requirements:

1. Placing the bomb: each four rounds, the not active bombs (which are not set off for explosion) are removed and two new bombs are placed on the board.
2. Picking up the bomb: after each valid move (round), the program checks if a player moved a piece onto a square occupied by a bomb. If yes, the bomb is set off for explosion.
3. Exploding the bomb: after each valid move (round), the counters of the active bombs are decreased by one. Then the program checks if there are any bombs that should be exploded. If, yes, then all pieces in a 3x3 vicinity of the bomb are removed from the board. Finally, the program calculates the possible moves all remaining pieces.

For each bomb a set of properties need to be maintained in the program:

* position: on which button the bomb has been placed randomly.
* status: has the player picked up the bomb (and set off for explosion)?
* rounds: how many rounds are left before the bomb explodes?
* label: the label GUI element that is used to display the remaining rounds to the user.

Therefore, I have created a class to represent the bomb items, and maintain a list of bombs in the program according to the following rules:

* if a bomb is placed on the board, a new Bomb object is created and added to the list.
* if a bomb is not picked up after four rounds, it will be deleted and removed from the list.
* if a bomb has been picked off, it will be deleted and removed from the list after it explodes.

### Integrating the Items into the Original Chess Program

To be able to use the developed functionality of the various items in the original chess code, the modules must be imported first:

import bomb

import coin

import shield

import barrier

Next, I needed to find out from which places to call which item routine. The following list describes which functionality is called from which place and in which context:

·       Placing the items: btnClick(), after selecting the target square, before making the move

·       Picking up the items: checkInput(), before making the move

·       Exploding the bomb: checkInput(), after making the move

·       Check if trying to moving to barrier: check\_chosen\_move(), before making the move

·       Check if trying to capture shield-protected piece: check\_chosen\_move(), before making the move

·       Restore item color: undo\_coloring(): after selecting the target square, before making the move

Since the item modules were designed to be independent, I was able to integrate each item implementation step-by-step.

After all integration work has been completed, the chaotic chess program is now functionally complete.

# Summary

I have learned

One my goals with this project was to improve my programming skills. I have learned a lot about Python With my progress in writing and understanding Python, I have achieved this goal. I think this skill will be beneficial for my future career.

My initial goal was to develop three different chess variations. Sajnos ez nem sikerült mert sok idöt igényelt a programok tesztelése és a talalt hibak kijavitasa. A color chess-ben sikerült minden hibat kijavitani, viszont a chaotic chess-ben maradtak meg hibak. Pl.. bomb own king check,

even the original had some issues: pawn promotion and casling are not supported.

In my finished program there are some remaining errors. To begin with, the initial code had some issues like not supporting pawn promotion or castling. When looking back I wish that I had started with a different chess code. While Color Chess does not have any known errors, it was an easier concept then Chaotic Chess. “Errors in Chaotic Chess”

~~Unfortunately Due to my efforts in fully understanding the initial chess code and underestimating the complexity of Chaotic Chess, I shifted my focus on developing two functioning games~~.

The project however did not only revolve around programming: time management, documentation and accumulating information were also important aspects of the project. I have learned how to evaluate and reference source information. Explaining the code to those unfamiliar with it was challenging. I needed to explain the program's functionality in a clear and concise manner. Having to reference every citation and figure, I have learned how to structure and write a technical document, like the Matura paper.

# Bibliography

Bro, C. (15. February 2021). *Python Full Course for free 🐍*. Von YouTube: https://www.youtube.com/watch?v=XKHEtdqhLK8&t=23528s&ab\_channel=BroCode abgerufen

Data, R. (kein Datum). *Python Functions*. Von W3Schools: https://www.w3schools.com/python/python\_functions.asp abgerufen

Data, R. (kein Datum). *Python Classes*. Von W3Schools: https://www.w3schools.com/python/python\_classes.asp abgerufen

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

Juviler, J. (30. August 2023). *What is GUI*. Von HubSpot: https://blog.hubspot.com/website/what-is-gui abgerufen

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

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

Data, R. (kein Datum). *Python Modules*. Von W3Schools: https://www.w3schools.com/python/python\_modules.asp abgerufen

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

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

algekalipso. (25. July 2022). *Bobby Fischer (1972 World Chess Champion) On Why Chess is a Lousy Game and How to Save It*. 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

j00nas. (2020). *chess GUI*. Von GitHub: https://github.com/j00nas/python-chess-game-GUI/blob/master/chess\_GUI.py abgerufen

# Table of Figures

[Figure 1 Chess 960 II](file:////Users/attila/Documents/GitHub/matura-arbeit/matura-paper-template.docx#_Toc155379732)

[Figure 2 GitHub Chaotic Chess History 2](#_Toc155379733)

[Figure 3 Debugging in VS Code 2](#_Toc155379734)

[Figure 4 Class Definition of Figure 4](#_Toc155379735)

[Figure 5 The name of the pieces 6](#_Toc155379736)

[Figure 6 Squares use one-based indexing 6](#_Toc155379737)

[Figure 7 Pieces use zero-based indexing 7](#_Toc155379738)

[Figure 8 Standard chess code process 9](#_Toc155379739)

[Figure 9 Control Flow Diagram 10](#_Toc155379740)

[Figure 10 Game visualization with the Tkinter GUI library 11](#_Toc155379741)

[Figure 11 Unicode character '|u265C' for a black rook 12](#_Toc155379742)

[Figure 12 Chess board set up by buttons 12](#_Toc155379743)

[Figure 13 Visualization of available moves 13](#_Toc155379744)

[Figure 14 Color chess 15](#_Toc155379745)

[Figure 15 Starting position in Color Chess 15](#_Toc155379746)

[Figure 16 Capturing pieces in Color Chess 16](#_Toc155379747)

[Figure 17 Chaotic chess 18](#_Toc155379748)

[Figure 18 Items During Play in Chaotic Chess 19](#_Toc155379749)

[Figure 19 Use case Barrier 19](#_Toc155379750)

[Figure 20 Use case Shield 20](#_Toc155379751)

[Figure 21 Use case Coin 20](#_Toc155379752)

[Figure 22 Use case Bomb 21](#_Toc155379753)

[Figure 23 White and black side buttons 22](#_Toc155379754)

# Appendix

## Appendix 1

Source code for Color Chess: https://github.com/LiroAV/color.chess

Source code for Chaotic Chess: https://github.com/LiroAV/chaotic.chess

## 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.

Vilters, 5.1.2024