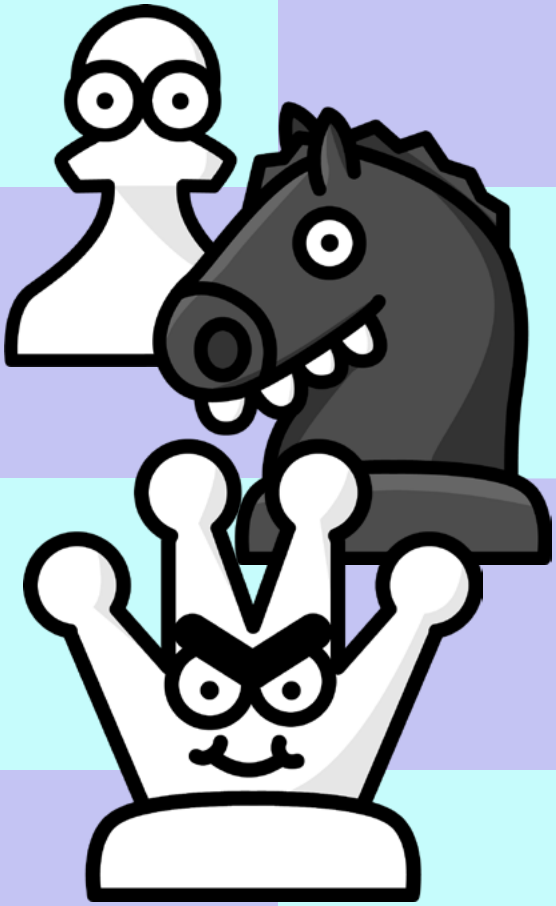
CHESS BATTLE!



Team 6 - Software 3

Contents

Introduction

Chess is a game of infinite possibilities, where each move can determine the course of the entire game! Similarly, Python offers a dynamic environment to translate strategic tactics into factual, functional code. In this document, we will journey through our process of combining these two worlds. The project's aim was to develop a chess game using Python with a focus on creating a visually engaging user experience with the PyGame library. The objectives include implementing the rules of chess, designing customisable visuals, and showcasing a proficient understanding of object-oriented principles.

This report outlines the project's journey from inception to completion. It covers the background and context of the chess game, specifications, design, and implementation details. With a focus on PyGame, testing, and evaluation, and concludes with a summary of our reflections, achievements and challenges.

Background

Our project involves creating a custom chess game with enhanced visuals using the PyGame library. Chess is a classic two-player strategy game played on an 8x8 grid, where each player controls distinct pieces with unique movement patterns. The goal is to checkmate the opponent's king, demonstrating a deep understanding of tactics and strategy. Players take turns moving their pieces according to their individual rules. The game continues until one player's king is checkmated, meaning it's in a position where it can be captured and cannot escape.

Specifications & Design

An Overview of Chess

Figure 1: Chess Battle! chess board set-up

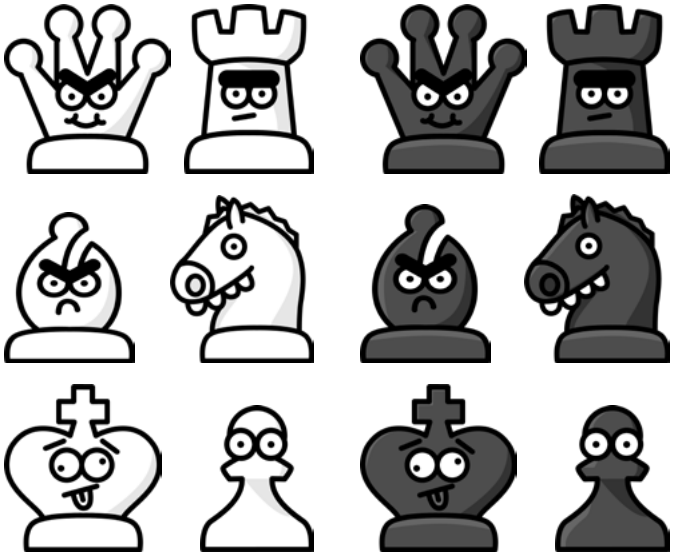


Figure 2: Custom chess pieces

Movement of Pieces:

1. King: The king can move, and capture, on one square in any direction (horizontally, vertically, or diagonally).
2. Queen: The queen can move, and capture, within any number of squares in any direction (horizontally, vertically, or diagonally).
3. Rook: The rook can move, and capture, within any number of squares horizontally or vertically.
4. Bishop: The bishop can move, and captrue, on any number of squares diagonally.
5. Knight: The knight moves in an L-shape: two squares in one direction (horizontally or vertically) and then one square perpendicular to the first movement, or vice versa. Knights are the only pieces that can "jump" over any piece. The knight can also capture in any of these moves.
6. Pawn: Pawns move forward one square, but capture diagonally. On their first move, pawns have the option to move forward two squares. Pawns also have a unique diagonal capture called "en passant." When a pawn reaches the opposite end of the board, it can be promoted to any other piece (except a king).

Special Moves Included in our game:

Castling: This involves moving the king two squares towards a rook and then moving the rook to the square the king crossed. Castling is only possible if neither the king nor the rook has moved previously, there are no pieces between them, and the king is not in check.

Pawn Promotion: This occurs when a pawn piece travels to the edge of the opponent's board and is replaced by a queen (in our game).

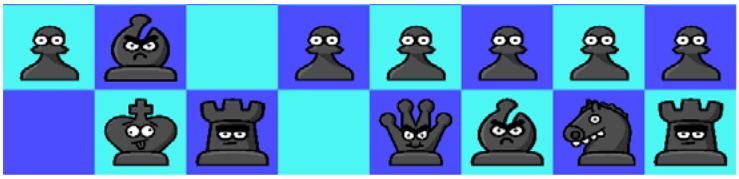


Figure 3: King and rook after castling

Figure 4: King and rook after castling

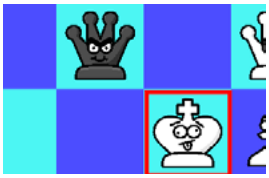


Figure 5: After pawn promotion (pawn promoted to queen & king in check

Functional Requirements

The functional requirements of building a chess game define the overall functionality of the operations and behaviours that will enable our software to effectively execute our game logic. We created an outline of the main structure of functional requirements necessary for a chess game to transition from board to software. It was a complex procedure of encapsulating the complex rules of chess that had to be understood and tested in our game.

Our foremost goal was to implement the intricacies of each unique piece's movements, factoring in their unique abilities and limitations. Equally crucial, our implementation accurately reflects the specific rules of capturing opponents' pieces while accounting for the abilities of each piece.

We also identified that detecting the "king in check" condition is pivotal, requiring continuous assessment of the king's state. By utilising these functional requirements, our coded version of the game achieves an accurate depiction of the intricate dynamics of this timeless game of strategy. Additionally, once satisfied with our minimal viable product, we integrated special moves of "castling" and "pawn promotion". These moves demanded careful consideration of the conditions that allow these manoeuvres on selected pieces.

Now that we have our essential piece functionality requirements met, our attention focused onto additional functional components of the game. We honour the traditional gentleman's game, where we give the option to resign from the game and reset all moves.

Non-Functional Requirements

There are an overwhelming number of Python chess games online. However, upon inspection two main areas of concern quickly became apparent. In smaller chess coding projects, it was very common to see almost all the chess code in 1 or two files. This is something we wanted to avoid in our code and employ an object-oriented approach to our overall structure. The next thing we noticed is that a lot of chess games looked very similar. Different colours, styles, but most lean towards a 'standard' design of a chess board and pieces.

We prioritised making our chess game fun, user-friendly and well-designed. We did this by creating our own branding, chessboard style, user interface, and custom chess pieces (please see figure 2). However, it is important to note, that even with this goofy fun exterior, players can still expect to play a seriously fun game of chess.

We also included a visual representation of our capture logic. This is displayed to the right of the board, updating with the corresponding image as each piece is captured. This element enhances user experience as it is a live depiction of each player's captured pieces that track their position in the game.

Figure 6: Chess Battle! logo

Figure 7: Chess Battle! dock icon

Figure 8: Chess Bot Charlie character

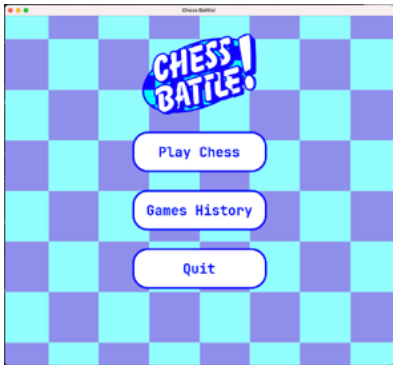


Figure 9: Chess Battle! main menu

Figure 10: Chess Game History screen

Figure 11: Chess Battle! winning screen

Design & Architecture of Code

Fundamentally, our priority in this project was to employ the core object-oriented program principles. To achieve this criteria we have separated code into classes reusable code into functions for readability and testability.

We fulfil the key principle of employing classes in our program by having each chess piece represented as a class with their individual functions, each inheriting from a common piece superclass. To ensure clarity, variable and function naming is understandable, reflecting the purpose they serve in the game. Additionally, our code facilitates future modifications or additions, demonstrating flexibility in our code architecture foundations. For instance, we pass our colours in as variables, allowing customisation in theme and aesthetics to be made.

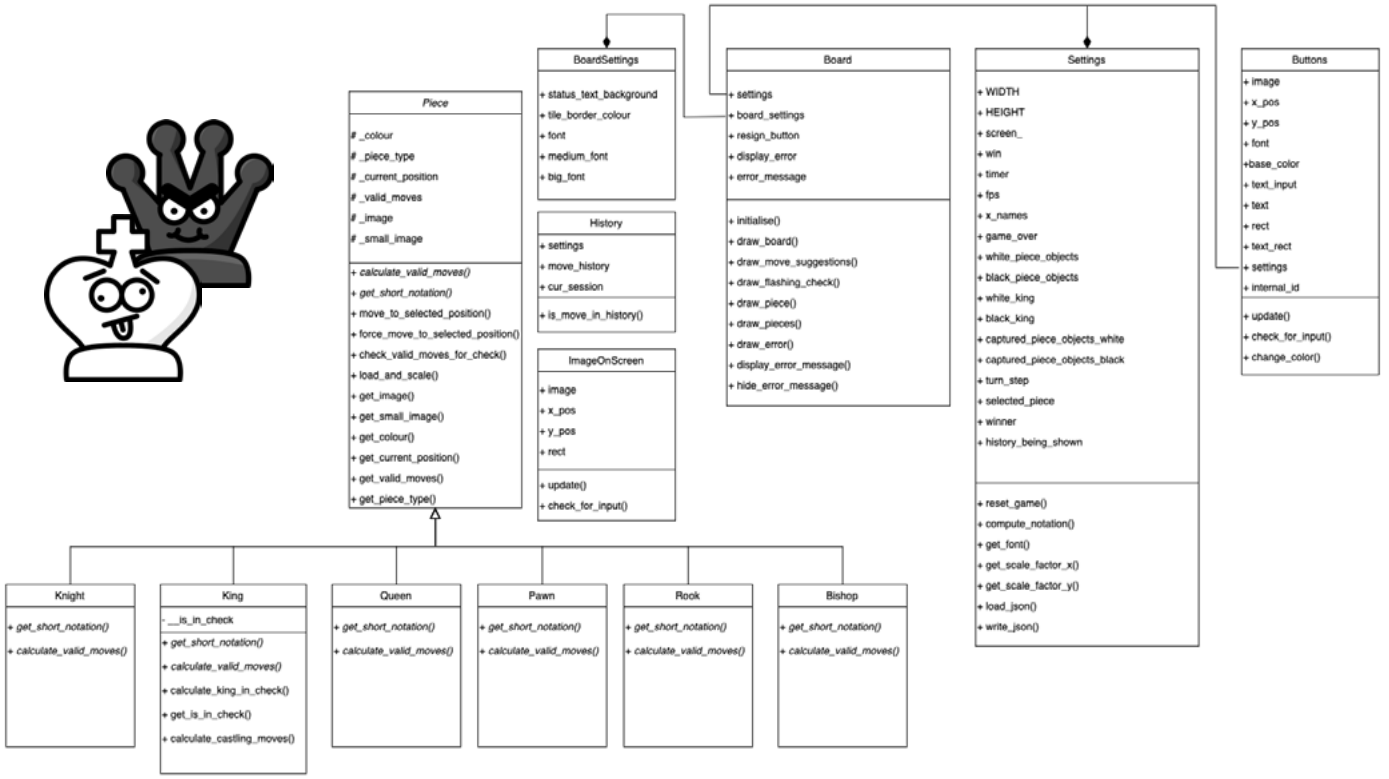


Figure 12: Chess Battle! UML Diagram (please click here)

Implementation & Execution

Agile Development

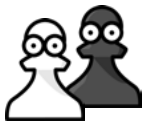
We embraced an Agile methodology, dividing our team into developers, designers, and testers. Developers focused on game logic, designers worked on visual assets, and testers validated the game's functionality. Regular meetings facilitated collaboration and decision-making.
We are focusing on taking an agile approach to the project workload and communication as a team to recreate a real, working scenario as closely as possible.
To help us achieve this we have scheduled regular meetings, collaborative shared files and, industry-standard tools and software to help us keep organised and communicate effectively. Ultimately, produced a well-structured, professional and engaging end product as a result. As our SCRUM master ensured all documents were up-to-date and encouraged collaboration.

Implementation Process

Our first action was to get our logic of the game and rules cemented in our program initially to begin developing. The intricate nature of chess rules posed challenges in ensuring accurate interactions between every element of the game. We researched different tutorials to familiarise ourselves with how our logic should perform.
Following this, we researched further and decided collectively to use PyGame to create a visual representation of our software and bring it to life! PyGame is a cross-platform set of Python modules that is used to create video games. It also has documentation and tutorials that we explored to give us a base knowledge of its mechanics.
We refactored our basic Python chess code to OOP, one of our first milestones, and began extending its capabilities. Meanwhile, we condensed and cleaned up our game logic and further refined repetitive code.
Overall, we achieved our key milestones including implementing piece movements, turn management, resetting the game, and special moves all with PyGame visualisation.

Implementation Challenges

The challenge of refactoring our basic, initial large file with all logic in one place was a big challenge to implement the skills recently covered in this course. This was certainly one of the main challenges of the project because of the sheer size of the initial file containing all of our code and took time to comprehend and implement. We had to be considerate of the connections between files and accommodate this with our imports, functions, and methods.
Additionally, once the refactored, clean file was ready to implement, we met various obstacles while merging into existing code that was developing alongside due to constant extension updates. Working with PyGame also had specific requirements that demanded some investigating to work with our Python game. Overcoming these conflicts with precise, manual merging and persevering, we achieve a clean, efficient standard of the program.



Testing & Evaluation

Testing Strategy

Our comprehensive testing strategy encompassed unit tests, manual tests, and user tests. Unit tests validated individual components, manual tests ensured interactions were accurate, and user tests evaluated the overall game experience.

User Testing

User testing was a critical process that helped us understand real players will interact with our software, identify problems or bugs, and gather valuable feedback for improvement.

Our user testing gathered feedback on our unique PyGame visuals, functional controls, and overall enjoyment with testers of varying skill levels playing our game.

Figure 13: Chess board screen showing 'invalid move'

Feedback gathered from Justina and Hayden:

- Both enjoyed the user interface and graphic illustrations and noted it was more appealing, fun and less intimidating, for someone who isn't particularly familiar with chess to use.
- Justina liked that the 'invalid move' message was shown on the screen and steered her on the right path when playing.
- Some improvements that were suggested were adding an undo button – they both made a few mistakes when playing, moving somewhere they didn't mean to, and so requested an undo button to correct the mistake before continuing to play.
- Users wanted a way to get to the main menu from the chessboard screen. This would be a good enhancement. When the player clicks the "chess battle!" logo it 'quits the game' and brings them back to the main menu.
- Found it easy to forget that they were playing a chess game that was made for a group project. Which we think is a big compliment and highlights the professional standard we created!

Unit Tests & Manual Testing

For our project, we meticulously created a vast body of unit tests as a quality assurance technique and debugging method. These tests focus on evaluating the correctness of individual components, such as our classes and methods, in isolation from the rest of the codebase. By doing so, it ensured that each piece's click logic, movement, capture mechanisms, and special moves were all functioning as intended. This approach allowed us to assess the correctness of specific code when it was working properly and also vitally when it was not.

We found unit testing beneficial for these purposes. However, there was an instance where we discovered a bug while manually testing the system. This was unexpected as the unit tests for this specific logic were all running successfully. We were able to use both of these methods of testing simultaneously to narrow in on our issue and efficiently highlight the bug to target and correct. This bug was hard to spot and, with our project being a game and frequent updates merged we were

required to manually test out our changes to prove visually that it was operating as expected.
This, as a result, led us to increase our testing on areas and throughout our game. In essence, combining these tests provided reassurance and confirmation of the chess game's core functionalities operating as anticipated thus contributing to a robust and error-free final product.

System Limitations

While we achieved core chess functionality, a visually and professionally appealing GUI, and OOP code criteria, we do have elements that could be enhanced. Given the time constraints of the project and our reduced group members, our brainstorming of this project envisioned great enhancements. When reviewing our code for submission we took time to tidy up as much as we could. During this review, we noticed that the game.py file, more specifically the main game loop in the game.py file, had quite a lot of duplicated code. This is something that, for future development, would be a main focus as it is a complex task and involves many lines of code and chess logic.
Our system limitations include the lack of AI opponents that we were eager to take our game to the next level. However, by meticulously organising our tasks and time, we deemed this to be a 'nice to have' feature of future development possibilities.
The system could have potential performance constraints on older hardware due to PyGame's graphical demands.
We also aimed to integrate SQL databases to store and display our game log history, however, we have taken a different approach to fulfil this feature of the game using JSON.
We presume our players know how to play chess. Other than supplying the possible moves of a piece once selected, our game doesn't teach players how to play chess. Our initial ideas on how we could expand our game included implementing pop-up messages and chess facts to aid and interact with the user.

Future Development: Accessibility Features

While the primary focus of the project was to create an enjoyable and functional chess game, there are several accessibility features we identified that could enhance its usability for a wider audience. Due to time constraints and project priorities, these features were not implemented but should be considered for future development. We would include an accessibility guide within the game's documentation to provide information on available accessibility features, keyboard shortcuts, and how to enable screen reader compatibility.
Implementing keyboard controls would make the game accessible to individuals who have difficulty using a mouse or touchpad. Integrating voice commands using a speech recognition library would enable players to control the game verbally. This feature would be beneficial for users with mobility or dexterity impairments. Players could say commands like "Move Queen to E4" or "Select Rook."
Also, providing descriptive alt text for in-game images and buttons would enhance the experience for players who rely on screen readers and would ensure they receive context about the game's elements. Additionally implementing audio announcements or text-to-speech (TTS) feedback for in-game events, such as check, checkmate, and promotions, would assist visually impaired players in understanding the game state.

For our screen settings, adding a high-contrast colour theme or mode could benefit players with visual impairments and make it easier to distinguish between chess pieces and squares. Likewise, allowing users to adjust the font size of in-game text and interface elements would accommodate players with vision challenges. Also, enabling users to customise the colour scheme of the game can cater to individuals with colour vision deficiencies or preferences for specific colour combinations. A colour picker or predefined themes could be included.

Reflections

We have identified some of the lessons from this experience that are beneficial to reflect on and improve upon for our careers. We found that it was challenging to foresee the scope of our chosen project that as beginners in Python, was capable of maxing out our capabilities in connection with our group downsizing. Moreover, combining the complexities of chess game logic, Python, PyGame, and team members also not knowing chess logic was a tough situation to strive through. We also underestimated the complexity of conquering our project in the time frame as our group had an abundance of quality ideas we could extend our project with. We appreciate the quality we have produced, our communication throughout, and the experience this task has provided us with.

Conclusion

In summary, our project successfully delivered a custom chess game with captivating PyGame visuals. Embracing an Agile approach, we overcame challenges in our game logic, GUI design, initial experience with pygame and overall user experience. However, our rigorous testing strategies ensured accuracy and functionality were delivered to our final product.

Although certain limitations exist we achieved our goal of creating an interactive, visually rich experience, complete with all the required traditions of the established game. This experience enhanced our ability to practise and further understand the transition from basic python code to object-oriented and the challenges this encompasses.

This project not only showcased our technical prowess but also vitally, our ability to collaborate seamlessly within a professional, dynamic development environment. We successfully combined many attributes and skills to simulate an authentic, online chess game for our final product that we are proud of.

See you on the chessboard!