Chess

Designed by

Pravesh Mansharamani

Vihaan Jagiasi

Purav Madhan

Introduction

This is our final project for the CS246 Fall 2023 semester In this culmination of our academic journey, our team of three individuals delved into the intricate world of chess, leveraging various Object-Oriented Programming (OOP) principles to bring this classic game to life in a digital space.

The objective of our project was to develop a comprehensive and engaging chess application from scratch. Embracing the principles of OOP, we aimed to create a modular, scalable, and well-architected solution that reflects our understanding of object-oriented design.

Working collaboratively as a team of three, we navigated the challenges and triumphs of software development. Our collaborative efforts spanned from the initial conceptualization to the meticulous implementation phase, with each team member contributing their unique skills and perspectives.

Throughout the project, we conscientiously applied various OOP principles such as encapsulation, inheritance, polymorphism, and abstraction. These principles served as our guiding compass, shaping the architecture of our codebase to enhance modularity, reusability, and maintainability.

Overview

Object-Oriented Principles:

Description: Embraced encapsulation, inheritance, and polymorphism to construct a modular and extensible codebase. Each chess piece is represented as a distinct class with dynamic behavior through polymorphism.

Memory Management: Ensured efficient memory usage by implementing smart pointers for automatic memory management, minimizing the risk of memory leaks.

Undo Feature: Implemented a comprehensive undo feature using a stack of move objects, allowing players to seamlessly backtrack their moves and revert the entire game state.

Design Patterns: Applied design patterns, including the observer pattern for flexible game state management and the singleton pattern for crucial components, enhancing the program's architecture and maintainability.

User Interface (UI) Redesign: Improved user experience through a modular UI design, facilitating easy updates and expansions. Graphics libraries were utilized to enhance the visual representation of the chess board and pieces.

Adaptive Input Handling: Ensured seamless integration of different player types by designing a consistent interface for move generation, anticipating pote**ntial future player types** for easy adaptability.

Scalable Scoring System:Implemented a versatile scoring system encapsulated within the "ScoreBoard" class. Leveraged the Strategy Pattern to define multiple scoring strategies, allowing for easy modifications or additions.

Future Adaptability: Designed the program with flexibility in mind, incorporating conditional structures and modular components. Anticipated potential variations, such as four-handed chess, to ensure the program's adaptability**.**

Design

Encapsulation of Chess Pieces:

- Application: Each chess piece is encapsulated within its class, containing both data attributes and methods tailored to that piece.
- Why: Encapsulation shields the internal workings of each chess piece, fostering information hiding and reducing dependencies between different pieces. Changes to one piece do not impact the behavior of others.

Inheritance in Chess Pieces:

- Application: An inheritance hierarchy is employed, with a foundational class "Piece" and derived classes for each type of chess piece (King, Queen, Bishop, etc.).
- Why: Inheritance facilitates code reuse, allowing common attributes and methods to be inherited from the base class. This promotes a more organized class structure and reduces redundancy across different chess piece implementations.

Polymorphic Behavior of Chess Pieces:

- Application: Polymorphism is exhibited through overridden methods like canMove, canCapture, and `move" in each chess piece class.
- Why: Polymorphism enables a uniform interface (method name) for different types of chess pieces. It allows each piece to execute methods tailored to its unique behavior, enhancing adaptability and flexibility in the overall design.

Abstraction of Chess Players:

- Application: Chess players are abstracted using the "Player" class, with concrete implementations for human and computer players.
- Why: Abstraction simplifies the representation of complex entities. The "Player" abstraction accommodates various player types without revealing their internal workings, fostering a more modular and adaptable design.

Composition in Chess Board:

- Application: The chess board is composed of individual chess pieces, each treated as an object. The "Board" class contains and manages these piece objects.
- Why: Composition allows the construction of complex objects by assembling simpler ones. The "Board" class doesn't inherit from individual piece classes but contains them, promoting a modular and flexible design.

Observer Pattern with Chess Pieces:

- Application: The observer pattern is implemented with each chess piece acting as an observer and the game board as the subject.
- Why: The observer pattern fosters loose coupling between the game board and individual pieces. When the game state changes, each piece is notified, enabling them to react

independently. This enhances adaptability and scalability, ensuring changes in one part don't affect others.

Polymorphic Input Handling with Chess Pieces:

- Application: Input handling is designed to be polymorphic, accommodating both human and computer players using the same interface in the context of chess pieces.
- Why: Polymorphic input handling ensures a consistent interface, irrespective of the type of chess piece. This simplifies the overall structure, promoting adaptability and clarity.

Scoring System as a Strategy for Chess Pieces:

- Application: The scoring system for evaluating player performance, encapsulated within the "ScoreBoard" class, adheres to the Strategy Pattern with individual pieces as subjects.
- Why: The Strategy Pattern encapsulates algorithms (scoring mechanisms), making them interchangeable. This enhances maintainability and flexibility, enabling easy modification or extension of scoring strategies for each chess piece.

Single Responsibility of Chess Pieces:

- Application: Each chess piece class adheres to the Single Responsibility Principle (SRP), handling a specific aspect of behavior related to that piece.
- Why: Following SRP enhances code readability, maintainability, and testability. Each chess piece encapsulates a unique behavior, making it easier to comprehend and modify without affecting other pieces.

Resilience To Change:

1. **Modular Class Hierarchy:**

   The use of a modular class hierarchy, with each chess piece encapsulated within its class, ensures that changes to one type of piece do not necessitate modifications to others.

   Impact: Modifications or additions to the behavior of specific chess pieces can be achieved independently, reducing the risk of unintended side effects across the codebase.

2. **Flexible Inheritance Structure:**

   The inheritance structure allows for flexibility in extending and modifying the behavior of chess pieces. New pieces can be added or existing ones modified without disrupting the overall architecture.

   Impact: The program can easily accommodate changes or additions to the types of chess pieces, supporting the introduction of new rules or variations.

3. **Dynamic Polymorphism:**

   The implementation of dynamic polymorphism through overridden methods enables the seamless extension of functionality for each chess piece.

   Impact: Introducing new rules, move patterns, or behaviors specific to certain pieces can be achieved by extending existing classes, fostering adaptability to changes in game dynamics.

4. **Smart Pointers for Memory Management:**

   The use of smart pointers for memory management ensures proper deallocation of resources, reducing the likelihood of memory leaks.

   Impact: This design choice facilitates the addition or modification of chess pieces without concerns about manual memory management, enhancing the program's resilience to changes in memory requirements

5. **Adaptive Input Handling:**

   The polymorphic input handling design allows for seamless integration of different player types (human and computer) through a consistent interface.

   Impact: Modifications to input handling, such as accommodating new player types or input formats, can be achieved without significant alterations to existing code, supporting adaptability to changes in user interaction.

Answers To Questions:

1. **Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. See for example https://www.chess.com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.**

   To instill a collection of conventional chess openings into our program, we would opt for a tree structure to arrange move sequences. Let's take a concrete example, the London System. In this methodology, each node within the tree denotes a distinct move in the opening sequence, while the branches of the tree illustrate potential responses, forming a visual representation of the unfolding strategic choices in the opening.

   Consider the outset, where the moves may begin with 1. d4 (White's move). Branches sprouting from this node could depict diverse reactions from Black, encompassing possibilities like d5, Nf6, or even the less common c5.
   Subsequently, if Black counters with 1... d5, additional branches could reveal White's potential moves, including options like exd5, Nf3, or c4. This pattern endures, constructing an exhaustive tree structure that elucidates the plausible moves and countermoves throughout the opening sequence.
   In the context of the London System, one might encounter a branch like 2. d5 (Black's response to 1. d4). This node would then unfurl branches for White's potential moves, ranging from Bf4, Bd3 to even Qe2.
   By navigating this tree in accordance with the ongoing board state during gameplay, we can systematically adhere to the predetermined opening sequence. The inherent simplicity and transparency of this tree structure facilitate seamless modifications or expansions with additional openings, presenting an exceptionally effective approach for incorporating a compendium of standard openings into our chess program.

2. **How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?**

   Implementing an "undo" feature in our chess program aligns with the principles of object-oriented design and encapsulation. Each move is represented as an object, and the state of the game is managed through a stack. To allow a player to undo their last move, we would pop the top move from the stack, reverting the game state accordingly. This approach ensures that the move object encapsulates all necessary information for a complete reversal. For an unlimited number of undos, we would continue popping moves from the stack until reaching the desired point, considering the stack as a sequential record of moves. This design leverages the principles of encapsulation, as the move objects encapsulate the entire state change, and the stack acts as a chronological history that facilitates undo operations efficiently.
   This implementation ensures that the undo functionality seamlessly integrates with the existing object-oriented structure of the chess program, offering a flexible and intuitive solution for players to backtrack their moves.

3. **Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.**

   To modify our chess program for a four-player chess variant, we would introduce two extra players, each with their unique set of chess pieces. The grid's size would be expanded to accommodate the heightened complexity of the four-player game.
   During the setup phase, players would be given the option to form teams or play individually. Specific adjustments would be made to align with the distinctive rules of four-person chess, designating coordinates at the corners of the expanded grid as out of bounds to mirror the true structure of the four-person chess grid.
   For game termination handling, customizations in the code would be essential. In team play, the game would conclude when both kings of a team are in check, while in single-player mode, the game would end when all kings but one are in check. These conditions introduce an extra layer of complexity to the win-checking logic, necessitating a thorough reassessment and modification of functions responsible for checking the status of kings on the board.
   With the increased number of kings on the board, many existing functions related to check and checkmate would require recoding. This involves a meticulous consideration of interactions between the kings of different players, as any king being in check significantly impacts the overall state of the game.

Extra Credit Features:

1. Threefold Repetition:

Implemented a check for whether the last three moves were the same for a particular piece.

Challenging Part: Tracking and comparing the last three moves required careful management of game states and history.

How we solved it: Maintained a history of moves and implemented logic to check if the last three moves for a piece matched, ensuring compliance with the threefold repetition rule.

2. **Fifty-Move Rule:**

   Checked if neither a pawn was moved nor a piece was captured in the last 100 moves.

   Challenging Part: Keeping track of all moves and ensuring accurate counts while considering special cases could be complex.

   How we solved it: Maintained a move counter, incrementing it for every move. Checked the conditions for the Fifty-Move Rule by iterating through the move history, ensuring that neither a pawn was moved nor a piece was captured in the last 100 moves.

3. **King & Piece Configurations Making Checkmate Impossible:**

   Identified specific configurations where checkmate is impossible, such as KingBishop vs. King, KingKnight vs. King, and King vs. King.

   Challenging Part: Recognizing these specific configurations and ensuring the correct implementation required a deep understanding of chess rules.

   How we solved it: Implemented logic to identify scenarios where checkmate is impossible based on the configurations specified. This involved checking the piece combinations on the board and determining the impossibility of checkmate in these specific cases.

4. **Undo Feature:**

   Implemented an undo feature allowing players to revert the last move.

   Challenging Part: Managing the game state, including the board and move history, to facilitate seamless undo functionality could be intricate.

   How we solved it: Maintained a history of moves and board states, enabling the system to revert to the previous state when the undo feature is invoked. Carefully managed the data structures to ensure accurate representation and restoration of the game state.

5. **Smart Pointers and Memory Management:**

   Adopted smart pointers and STL containers for all memory management, eliminating the need for delete statements and minimizing raw pointers.

   Challenging Part: Ensuring robust memory management and eliminating the risk of memory leaks required a disciplined approach to coding.

   How we solved it: Smart pointers, such as std::shared_ptr and std::unique_ptr, were used to manage dynamic memory. STL containers facilitated efficient storage and retrieval of game states, moves, and other relevant data structures, resulting in a clean and memory-safe implementation.

Final Questions :

1. **What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?**

- **Strategic Collaboration and Communication:**In the crucible of collaborative development, effective communication emerged as the cornerstone of success. We realized that regular, transparent communication channels were not just facilitators but integral to a harmonious and efficient workflow.The refinement of our communication processes drastically reduced misunderstandings, ensuring that everyone shared a unified vision. This, in turn, elevated the overall productivity and cohesion of the team.
- **Project Planning and Management:** The meticulous planning and systematic management of the project were pivotal. From defining granular milestones to assigning tasks judiciously, we recognized that structured planning was not merely a formality but a linchpin of success. Breaking down the project into manageable components with well-defined timelines led to a streamlined development process. The team's ability to adhere to these plans resulted in the project progressing smoothly, meeting deadlines with precision.
- **Strategic UML Design and Architecture:** The project underscored the critical importance of robust design before implementation. The use of UML diagrams was not a perfunctory step but a strategic investment in a cohesive, scalable architecture. The UML diagrams served as a visual compass, guiding us through the complexities of the project. This upfront design thinking laid the foundation for a modular, adaptable codebase that stood the test of subsequent development phases.
- **Efficient Code Integration and Version Control:** Harnessing the power of version control, particularly Git, emerged as a linchpin in ensuring code integrity and seamless collaboration. Code integration was not just a technical necessity but a strategic imperative. The team's proficiency in version control empowered parallel development, allowing for a harmonious convergence of diverse contributions. Learning to navigate and resolve conflicts enhanced our collective coding acumen.
- **Rigorous Testing and Debugging Practices:** Testing ceased to be a periphery activity; it became the vanguard of code reliability. Rigorous testing, encompassing unit tests and meticulous debugging, transformed from a routine to a strategic imperative. The comprehensive testing strategy became a shield against unforeseen bugs and glitches. Early detection and resolution of issues not only fortified the codebase but instilled a culture of code quality within the team.
- **Adaptability and Flexibility in Development:** The dynamic nature of software development demanded a paradigm shift towards adaptability. Unforeseen challenges were not stumbling blocks but opportunities to showcase resilience and flexibility. Embracing adaptability equipped the team to navigate unforeseen challenges seamlessly. The experience heightened our collective agility, enabling us to respond to shifting project requirements with creativity and resilience.

In essence, the chess project served as a crucible for refining not just technical skills but also the softer aspects of teamwork, strategic planning, and adaptability. The impact of these lessons extends beyond the confines of this project, forming a robust foundation for future endeavors in the ever-evolving landscape of software development.

What would you have done differently if you had the chance to start over?

1. **Strategic Testing Implementation:** Looking back, a more assertive approach to testing, aligned with object-oriented principles, would have significantly elevated the project's robustness. In a hypothetical restart, we would incorporate a stringent testing strategy, incorporating comprehensive unit tests and integration tests early on. This approach, in line with principles like encapsulation and modularity, would not only catch issues sooner but also fortify the project against unforeseen challenges, ensuring a more resilient and maintainable end product.

2. **User Feedback as a Cornerstone:** The realization dawned that user feedback is not just a phase but a continuous process crucial to realizing the full potential of object-oriented design. In a do-over scenario, we'd insist on relentless user feedback at every development juncture, ensuring our object-oriented design aligns seamlessly with user expectations. This iterative engagement, guided by principles like polymorphism and inheritance, would not only fine-tune the project but also establish a more adaptable, user-centric architecture.

3. **Exemplary Documentation Practices:** Documentation, though present, could have been a more potent tool when aligned with object-oriented principles like abstraction and encapsulation. If given a fresh start, I'd advocate for a documentation overhaul, focusing on clarity and abstraction. Comprehensive code comments, detailed README files, and user manuals, guided by principles of encapsulation, would serve as essential documentation artifacts. This wouldn't just expedite our processes but also enhance the project's longevity and maintainability, reflecting the principles of good object-oriented design.
4. **Agile Adaptability for Optimal Performance:** The hindsight revelation is that our project could have better embodied the principles of object-oriented design with a more dynamic and adaptive development approach. In a do-over scenario, we'd strongly advocate for the infusion of agile methodologies, synergizing with the principles of polymorphism and encapsulation. Regular sprint planning, shorter development cycles, and frequent retrospectives would not only enhance our adaptability to changing project dynamics but also ensure a consistent and high-quality output, mirroring the principles of object-oriented development.
5. **Strategic Resource Allocation:** While task allocation was efficient, a more strategic alignment with object-oriented principles, such as composition and inheritance, could have optimized our team's potential further. Starting afresh, I'd champion a flexible task allocation system guided by principles like composition. This approach would dynamically adjust to each team member's evolving strengths and workload, not just maximizing individual contributions but also creating an environment where our team collectively operates at its peak efficiency.
6. **Git Governance for Seamless Collaboration:** Utilizing Git was a wise move, but ensuring adherence to best practices from the onset could have better reflected the principles of object-oriented design. If given the chance to start over, we'd insist on early and comprehensive training on Git best practices, emphasizing the principles of encapsulation and modularity. This preemptive strike against conflicts would elevate our team's collaborative coding proficiency, leading to a more harmonious and productive development process in line with object-oriented principles.

   In conclusion, these strategic adjustments, interwoven with object-oriented principles, aren't just improvements; they're potential game-changers that could transform our chess project into a more resilient, user-centric, and efficiently designed software endeavor. The strength of these changes lies not only in rectifying past oversights but in proactively steering the project towards unparalleled success through a solid foundation in object-oriented design.

Conclusion

In wrapping up this chess project, we found ourself both reflective and proud of the journey we've undertaken. From the inception of UML diagrams to the nitty-gritty of coding each chess piece, this endeavor has been an immersive exploration of object-oriented programming and collaborative teamwork.
Our commitment to object-oriented principles has not only resulted in a modular and adaptable codebase but has also given the program a robust framework for future enhancements. The implementation of features like the undo functionality, smart pointers, and a nuanced scoring system underscores our dedication to crafting a sophisticated and user-friendly chess experience.

The integration of design patterns, like the observer pattern, showcases our commitment to creating a scalable and resilient architecture. The meticulous attention to detail, from UI redesign to strategic AI improvements, reflects our pursuit of excellence in delivering an engaging gaming experience.