CS 247 Final Report

Roger Chen, Norman Chen, Peter Wang

Overview

The project has 3 primary classes:

- **Chess:** This is the main class component of the project, responsible for managing the overarching interactions of the system. This is what gets instantiated in the main function.
- **Game:** This manages a single game state and encapsulates the core game logic and interactions.
- **Board:** This represents a single board state and encapsulates the logic for things like piece placement and movement.

The main relationships between these are that **Chess Owns Games**. Chess can own as many games as it wants because you can play multiple games in a session. The other main relationship here is **Game Owns a Board**. This makes sense as each game is played on one board, so each game object only owns a single board object.

Each time the *game* command is ran to play a new game, the chess object will instantiate a new game object for the game, and the game object will instantiate a new board object.

The **Chess** class also has a **owns a** relationship with 2 other classes:

- **DisplayBoard:** An abstract class with **TextDisplay** and **GraphicsDisplay** as subclasses. Responsible for displaying a text-based board and graphical board (via X11) respectively to the user. An observer design pattern is deployed here.
- **Player:** Another abstract class with **Human** and ***Computer*** as subclasses (Computer is abstract with concrete subclasses for each level). Responsible for facilitating moves from either a computer or human player.

Chess will typically own 2 **DisplayBoard** objects, one for the console version and one for the graphical version. When the chess object gets instantiated, the DisplayBoard objects are also instantiated. Both displays are instantiated by default, but the graphics display can be omitted via a command line argument.

Chess can also own as many player objects as it since multiple games with different players can be played in a single session. Whenever a new game is created, the chess object will

instantiate 2 more player objects. We store all the games and players for possible future additions.

The **Board** class also **owns** ***Pieces***. ***Pieces*** is an abstract class with a concrete subclass for each of the 6 types of pieces. A board object can own as many piece objects as a board can allow. When a board object gets instantiated, the board will instantiate the required number of piece objects for the desired board.

When a *move* command gets ran, the command is first parsed by the chess object. Then the chess object calls the corresponding *make_move* function for either the human or computer player. This function calls the game object to validate and execute the move with the board object.

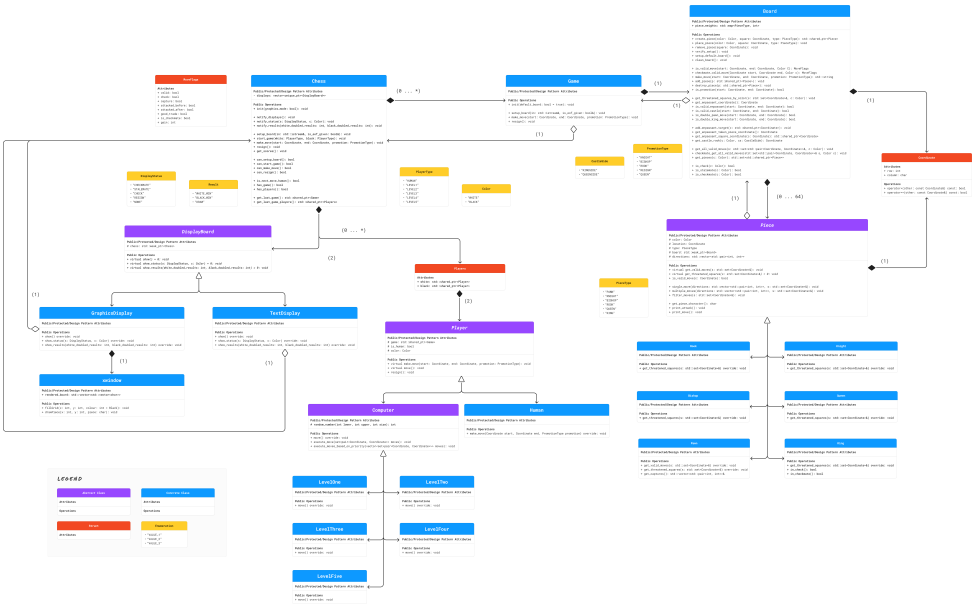
At the same time, the chess object notifies the displays to display the board. The boards get the state from the chess object then displays the new state.

We also have command line flags. These command line flags include

- –**no-graphics** or **-ng**: This flag disables the graphical interface, for development in environments that do not support X11.
- –**no-strict** or **-ns**: This flag disables strict mode for input. Strict mode throws errors when invalid inputs / arguments are given. Non-strict mode catches these errors and allows for the user to re-enter inputs.

Updated UML Diagram

The updated UML diagram is highlighted below:



Throughout the development process, we had to make adjustments to our initial UML diagram in order to make everything work together.

Display Ownership

The main difference between the updated and initial UML diagram was that we moved the ownership to the displays (both graphical and text based) from the *Game* object to the *Chess* object. This allowed us to have one display for each chess instance, instead of heaving to maintain a vector of displays for each Game.

Additional Helper Methods & Fields

There were also many helper methods and additional fields that we had to include in our program beyond our initial considerations. For example, boolean fields had to be maintained to track whether en-passant or castling was optional for each player.

Global Structures

We discovered that maintaining global structures, such as *PieceTypes* and *Color*, throughout the program was significantly easier. This approach contrasted with our initial method of embedding these structures within classes, which required additional code and complexity.

Design

Smart Pointers

We helped solve the problem of memory management through vectors and smart pointers. For example, the ownership of our display classes were represented via unique pointers in our chess class. The games and players were also represented with a vector of smart pointers in the chess class. Moreover, "has-a" relationships between objects were represented via weak pointers.

Because of this, we were able to not explicitly managing our own memory and have no delete statements, making development easier to and simply without worrying as much with memory.

Dynamic Casting

We use dynamic casting to help with en-passant and castling. In castling, we need to check if the king is in check in order to be able to castle. Dynamic casting helps us achieve this since we can dynamic cast the piece in the board class method to king in order to call its *in_check* function.

Likewise, dynamic casting has helped us with en passant. It helps us call the *get_valid_moves* method in Pawn in order to help us determine whether we can en passant. Dynamic casting has helped simplify and improve our code by improving cohesion, since each concrete piece class tries to handle its own logic for specific chess rules.

Observer Pattern

For the graphics part, we employed the observer design pattern. The chess class is the concrete subject and the DisplayBoard class is the abstract observer. The concrete graphics and text display classes are the concrete observers.

This helps improve cohesion as the display observers only job is to display the chess board it has. It also helps reduce coupling as the display logic is not as tightly coupled to the chess class. Thus the project is more maintainable and easier to work on.

Strategy Pattern

The Player class employs the strategy design pattern, particularly the computer and each of its levels. By defining the player class as abstract and implementing each type of player as a concrete subclass, it is easy to add in new strategies with new subclasses (like new computer levels).

This also follows the open/close principle. For example, if you want a new level for computer, you simply add a new subclass to Computer instead of having to modify an entire Computer class or something in that nature if the strategy pattern wasn't applied.

Template Pattern

Our Piece class follows a template pattern. The piece abstract class has the template method *get_threatened_squares*. Each concrete piece class implements this method in its own way to help provide their specific movement and capture logic. The piece class just defines the overall structure of the piece and provides common functionality.

Coupling and Cohesion

During our design and implementation, we aimed to minimize coupling and maximize cohesion. We achieved these things by employing a variety of different design patterns and OOP principles.

For example, our template pattern helped with coupling by separating the piece logic from the board itself, it also helped with cohesion since each piece is just responsible for the spaces it can threaten.

We also minimized coupling by having no friend classes or global variables for game logic (we had global variables for debugging flags, but none for game logic).

Cohesion was maximized by having many classes for each part of the game. Splitting the core game into the 3 classes (chess, game, and board), and having each piece, player, and display as its own class really helped maximize cohesion as each class has its own single clear objective.

Resilience to Change

Given our high cohesion, any specific changes to certain parts of the program can be easily made by simply going to the class and making the change there.

Furthermore, due to our low coupling, any changes in our classes are unlikely to break anything else. For example:

Simultaneous Games/Game History Support

We store all the games and players that play in a vector. When a new game starts we simply append to the game and players vector in the chess class. Because of this, if we wanted to support simultaneous games we could add support for "pausing" games and allow the user to go back in the vector to play a past game. It also supports game history a since you can go back in the vector to look at past game states and informatiom.

Undo Operation

Our board has FEN support (a concise way to represent a chess board as a string). We have a FEN serializer and deserializer and our board has a constructor that takes in a FEN value to create the board.

Because of this, we simply store the FEN after each move in a string vector. To undo a certain number of moves we just use the find the corresponding FEN value and recreate the board with it with our serializer.

New Pieces/Piece Rule Changes

Because we employ the template design pattern, adding in new pieces is quite easy as all you need to do is add a new subclass to Piece and write the *get_threatened_squares* function for the pieces behaviour.

If you want to change the behaviour of an existing piece it is also quite simple, just go to the pieces class and modify just the template method. Template design makes this feature simple as typically only you only need to focus on the one function.

New Computer Levels/New Players

Again, because we implement the strategy design pattern, adding in new computer levels or any new player types is easy. All you would need to do is add a new subclass for the player/level.

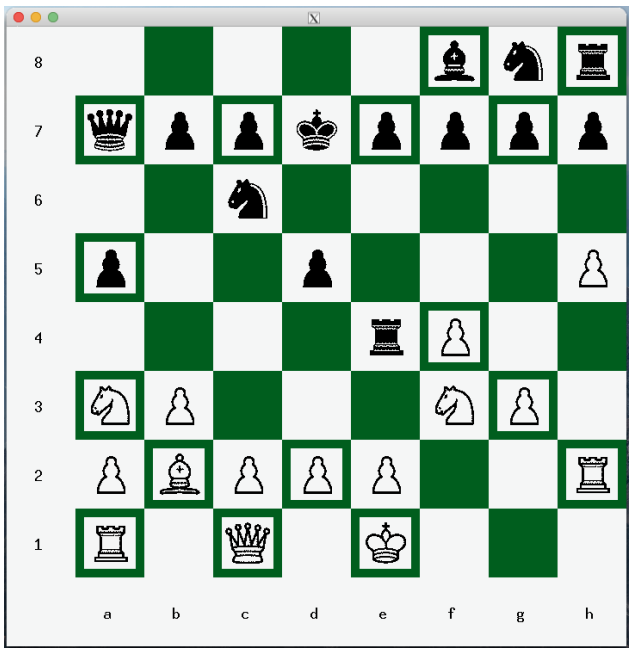
Fun Game Modes

We could implement game modes like, no Queen or no pawns. Or even game modes where one player is down certain material. We could implement these via our setup function for creating any board state and wrapping it around the new command for the game mode.

Extra Credit Features

Sophisticated Graphics Display

Having visually appealing graphics was a challenge. The provided Xwindow class had very little functionality builtin, so we had to learn a good amount of X11 and heavily modify the Xwindow class to get clean graphics. We were able to import images of the actual chess pieces with color coding into our graphical display, as seen in the screenshot below.



For example, in order to use images, we had to get bitmap of each image. This involved converting the image of each individual chess piece to a bitmap and importing it in. Then we had to use the Pixmap functions of X11 to actually render it, this made the graphical display much more visually appealling to work with.

Comprehensive Test Suite

A comprehensive test suite was generated by taking public machine learning game data from Hugging Face's chess dataset. These games are the result of Stockfish 16.1 playing itself. We took over 1000 games and parsed them into our own input and output format as test cases.

The test cases are in the *.in* and *.expect* format, which can be found under the 'tests/in' and 'tests/expect' directories respectively.

Continuous Integration & Continuous Development (CI/CD) Pipeline

To ensure that additional features added to our chess program did not break basic functionalities, we implemented a modern CI/CD pipeline using Github Actions with over 1000+

test cases as mentioned above. Correctness testing was conducted on every push and pull request, whereas the longer and more sophisticated memory tests with Valgrind was conducted every hour.

Level-5 Computer

In addition to the already advanced level-4 computer, we implemented a level-5 computer program that consistently beat every other level (including level-4 with over 90% checkmate rate) in less than 200 moves.

Additional features of the level-5 computer include:

- Checkmate Detection: Detecting and executing immediate checkmates (mate in one) if such a move exists, as this has the highest priority.
- Favorable Trades: Engaging in trades only if they are favorable, meaning the piece taken is more valuable than the piece that could be lost on the next turn.
- Simultaneous Check/Capture/Escape: Prioritizing moves that achieve multiple objectives, such as delivering a check, capturing a piece, and escaping from an attack, over moves that accomplish only one of these tasks.
- Capture Value: Out of all possible captures, select the one with the highest value.
- Escape Value: Among pieces under attack, move the most valuable piece out of danger.
- Endgame Strategy: Prioritize checking rather than protecting pieces in the end game, as this would make it easier for checkmates to occur

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?

Building and developing this chess program provided many valuable lessons and takeaways for each team member.

- **Division of Labor**: One of the most important aspects contributing to the success of our project was the equitable division of work among team members. By tracking each members progress and redistributing tasks as needed, we ensured that everyone contributed effectively and no one was overwhelmed by their workload.
- **Modular Design & Documentation**: The modular design of our program facilitated smooth division of labor. Each member could work on different parts of the program independently, without waiting for others to finish their tasks. Detailed documentation and comments enabled team members to quickly understand each others work, ensuring seamless transitions between different stages of the project.

- **Importance of Version Control**: Our prior co-op experiences underscored the importance of version control systems like Git in team environments. We utilized branches to maintain a working version of our software, preventing conflicts during development. Additionally, Git allowed us to conduct effective code reviews, incorporating multiple opinions before merging changes into the main branch.
- **Test Driven Development (TDD)**: Focusing on TDD throughout our development process gave us confidence in our code changes. Writing tests before implementing features ensured that new additions did not break existing functionality, maintaining the integrity of the chess game.
- **Continuous Integration and Continuous Development (CI/CD)**: Early in the project, we set up a CI/CD pipeline using GitHub Actions. This automated the running of our test suite on every code push, pull request, and merge, ensuring that our changes did not break the production environment without manual intervention.

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

If we had the opportunity to start from scratch, we would make some key changes:

- **Improved Initial UML Diagram**: Spending more time on the initial UML diagram to ensure all cases are considered would have minimized design changes during coding. A comprehensive UML diagram would provide a holistic view of the project, preventing surprises during development.
- **Reduce Code Redundancy**: We would focus on reducing code redundancy. Many components shared similar functionalities but had separate functions. Consolidating these into single, more versatile functions would optimize the codebase and improve maintainability.
- **Integrated Test Suites**: We would integrate tests directly within our program using libraries such as GTest, instead of relying on external tests via shell or Python scripts. This integration would enable more sophisticated and detailed checks on valid moves, checkmates, and overall game logic for both human and computer players.
- **Performance Optimizations**: Focusing on performance optimizations, especially for the computer logic, would allow for faster computations and deeper searches of possible outcomes. This would enhance the efficiency and effectiveness of the chess engine, providing a more challenging and responsive experience.

Conclusion

In conclusion, this project provided us with real world examples of C++ concepts and software engineering design principles. This experience also helped us develop crucial teamwork skills, which will be extremely crucial for our future careers. Overall, it was a fun project that we all enjoyed and had a fun time building.