Team: Tyler Tafoya, Mitch Zinser, Elanor Hoak, Anna Yudina

Title: Chess Xpress

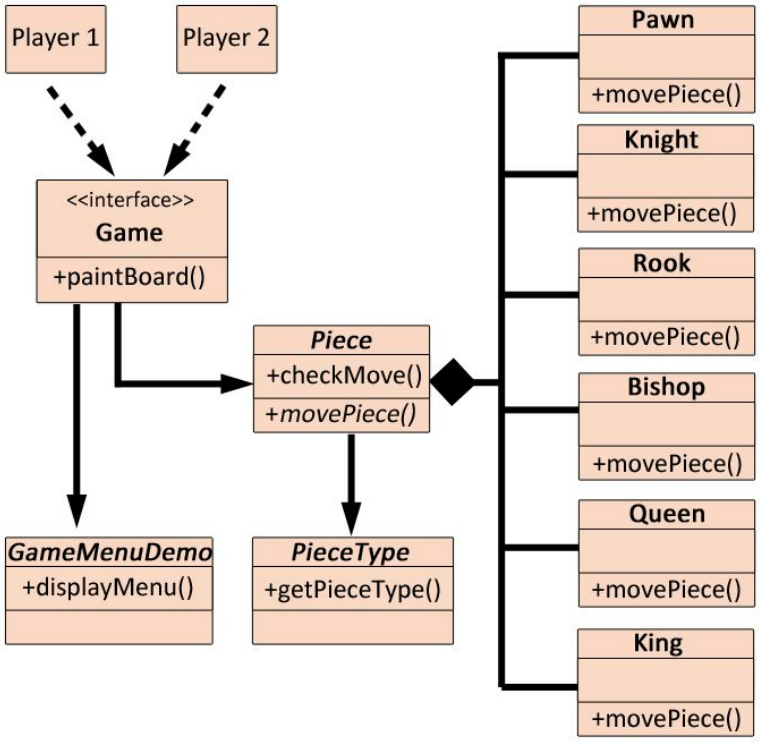
Project Report

Final Features

- The system displays a 2D representation of the chess board
- The system displays a 2D representation of the chess pieces
- Each player will begin the game with 16 pieces (8 pawns, a king, a queen, two rooks, two bishops, two knights)
- Pieces respect the capture logic
- Game ends in either checkmate or stalemate
- Two players can play this game of chess
- Pieces can only move based on predefined rules

Final Class Diagram

Class Diagram at the end of the project:



Benefits Of Design Before Coding

The main benefit of design, before the coding process, is that it allows one to thoroughly think through the whole entire project. One aspect of design is to define the requirements of the project, which can be completed by running through multiple use cases for the different kinds of requirements. This helps to determine what classes and methods should be implemented in the program. The design allows one to foresee any errors that can occur, prior to the coding process. The programmer(s) can alter the design to avoid the errors in the first place or come up with a solution that solves the errors. If there is a client, then they can approve of the design or they can alter the design by adding or removing certain features of the system. Our team benefited from this design process by explicitly thinking of how methods and classes would interact in our system. We were also able to pinpoint future potential challenges, and devise plans to mitigate those risks. Even though our final implementation of the project shifted from the initial design, this was expected. Not all challenges are easily anticipated, and sometimes new issues arise during the development phase. The real benefit in the design was to get our group thinking at the system level, in order to plan and delegate roles and responsibilities.

Design Patterns

In the final implementation of the Chess Xpress prototype, we did not explicitly include any design patterns. Given more time, there are a couple design patterns we might have considered to improve the object-oriented design approach, and optimize the efficiency of the system.

Strategy

The Strategy design pattern allows the system to choose algorithms dynamically at runtime, which would be very effective for Chess Xpress. Each subclass of *Piece* (i.e. Pawn, Bishop, Queen, etc.) unique movement logic, and an optimal design would utilize loose coupling and allow the system to determine a *Move*() at runtime, rather than coding each piece's movement logic within the class itself. An implementation of this would to create some abstract class which all the pieces can extend and set their own movement behavior. This would also allow for code reusability, as movements are limited to vertical, horizontal, and diagonal movements.

Decorator

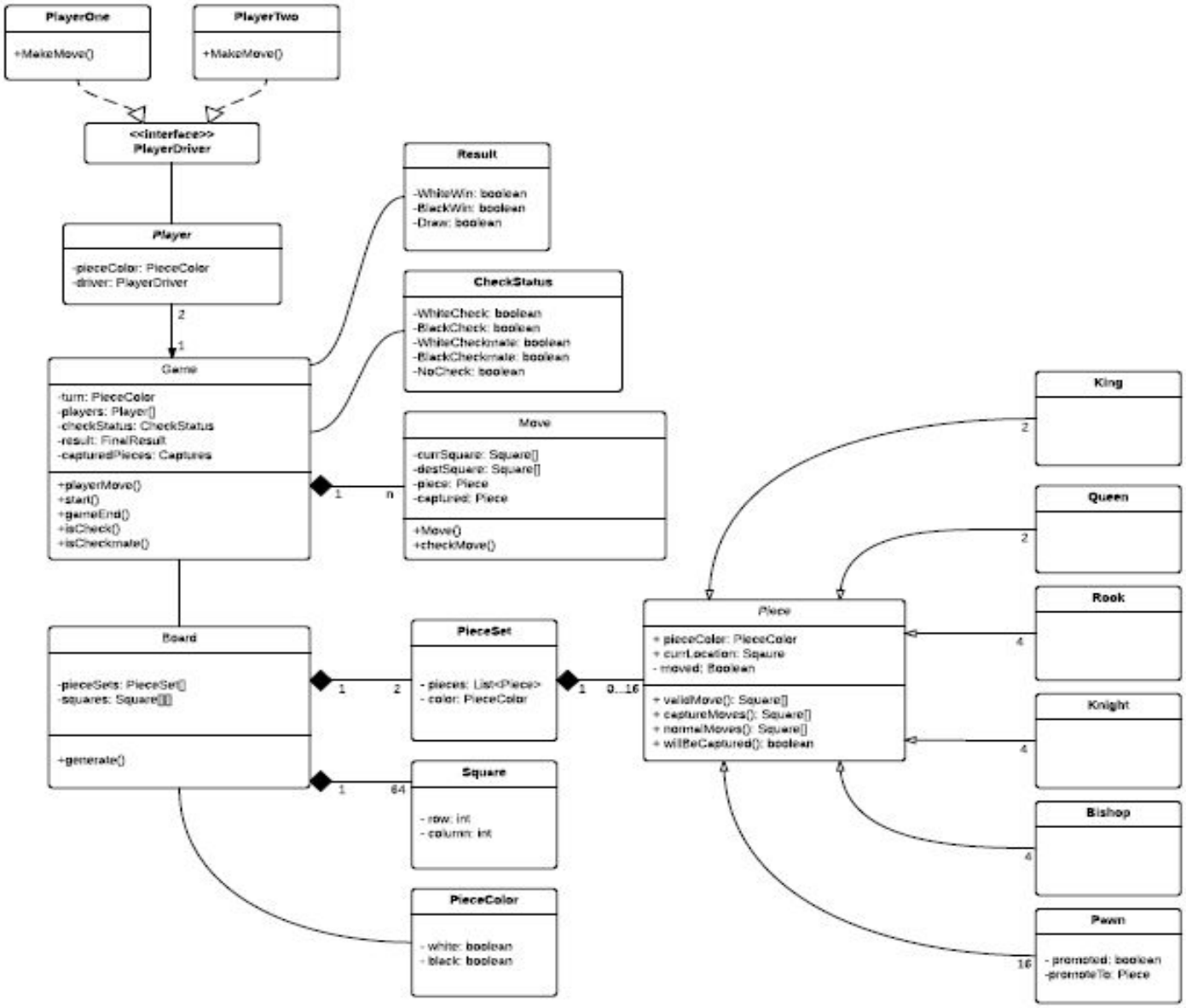
The Decorator pattern is designed to dynamically grant objects additional responsibilities and extended functionality. This design pattern would be suitable to include in our system in order to give the *Pawn* subclass the ability to promote. Each subclass of *Piece* (i.e. Pawn, Bishop, Queen, etc.) implements validity checks and movement logic corresponding to the respective chess piece. The *Pawn* subclass needs the added functionality of "promoting" if it reaches the back row of the opposing player. Essentially, a *pawnDecorator* class would be created with promotion methods, and allow *Pawn* to maintain the current *Piece* substructure.

Abstract factory
The Abstract Factory design pattern creates “families of related objects without specifying concrete classes”, and could have been useful in our implementation of Chess Xpress. Since Chess involves two players, White and Black, these could be extended into abstract factories of a main *ChessFactory* class. This would allow for dynamic assignment of the *Piece* class with respect to the player color.

Memento
The Memento design pattern stores the internal state of a system and allows a user to revert to that state later without violating encapsulation properties. This design pattern could be useful for Chess Xpress in order to allow a player to revert X number of moves before reaching *checkmate*. When a player loses a game, it would be an interesting feature to allow that player to “go-back” say, 10 moves, in an attempt to correct any fatal mistakes. The use of this design pattern would also couple with the *Iterator* design pattern in order to “iterate” through each system state prior to checkmate.

Class Diagrams

Initial class diagram from Part 2:



Our final class diagram is far more simple than our original plan. The most significant change was between the Board and Game classes. A major difficulty we had was drawing the images on the board corresponding to their object pieces. CheckStatus was also implemented within the Game class to check for the “check” condition, but due to time constraints there is not a “check checkMate” function. Checking for black vs. white “check” states are considered the same and are always applied when the turn switches to the next player. The design would be better if it included relationships between the graphics and the game such as onClick to highlight a piece or square. PieceSet and PieceColor were also removed from the initial class diagram, and were implemented in the Piece and Game classes respectively.

Reflection

In the initial phase of the project, it's important to define the requirements and the different use cases, that a user will go through. This defines the flow of the project and helps the programmers to determine the main purpose or goal that the project will accomplish. These aspects will also define the design and architectural patterns that will be used for the satisfying the requirements and use cases. Once the design and architectural patterns have been chosen, the team of programmers should create a class diagram that will display the classes and their methods that the program will utilize. The design and architectural patterns should also be integrated into the class diagram.This will help the team to get an overall picture of how their program should be structured. Sequence diagrams, which make use of the class diagram's classes and methods, can be used to see how the program, theoretically, should behave with each use case scenario. If this is all accomplished, then the actual process of coding isn't difficult.