Project 6

Team members: Haley Hartin, Blythe Waltman, Haley Drexel
Title: Un-Chess-Ted

Final State of System Statement

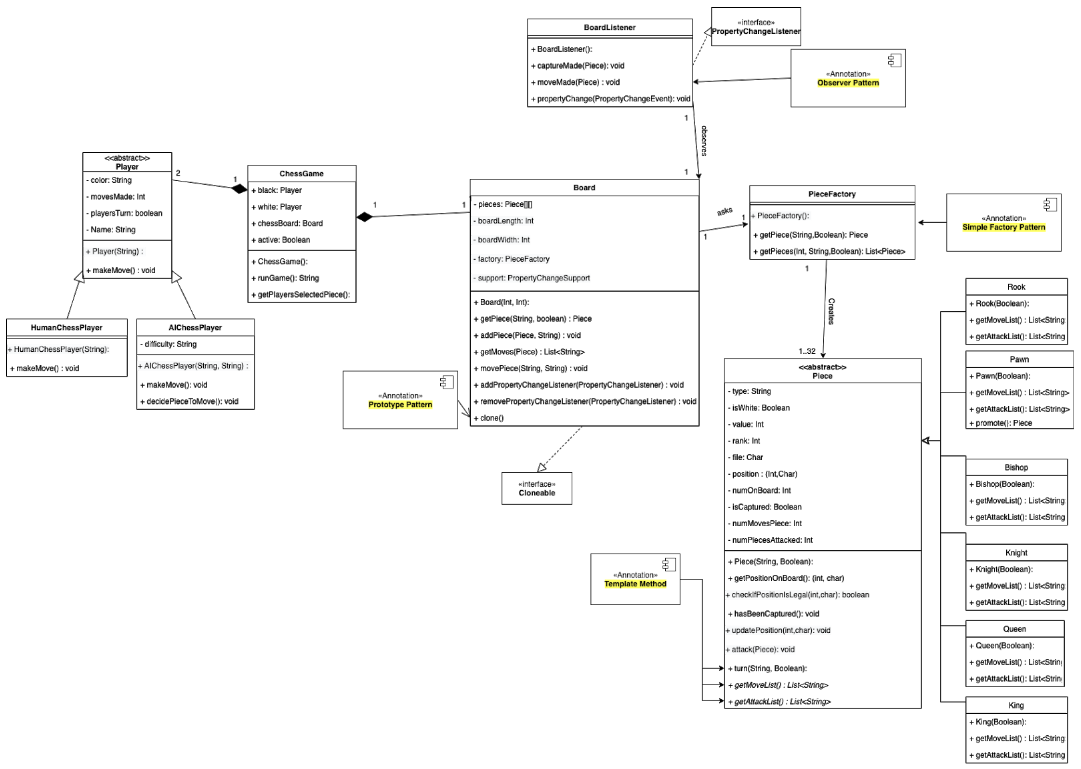
The final state of our system is a functional chess game for two human players or a human player against an AI player. Two human players can play locally and share the same screen on a standard game 8x8 checkerboard. When a piece to move is selected, the legal moves are highlighted, and only a move to a highlighted space is allowed. The players take turns making moves. The game ends if there are no legal moves to make, a player's king is captured, or a player's king is in checkmate with no legal moves to make. The system stores the pieces and moves each player made throughout the game in a table the players can access.

We did not have time to implement a database to store login information and players' statistics. We wanted to focus on the board functionality and game logic before we implemented this, and we ran out of time. We also did not have time to implement different difficulty levels for the AI player. Instead, we have a default difficulty level of "easy" where the AI player randomly chooses a piece and legal spot to move to. We also didn't have time to implement special moves (like castling) in order to refine the core moves of the chess pieces. An exception is pawn promotion, which was implemented due to it enabling the pawn to be more valuable.

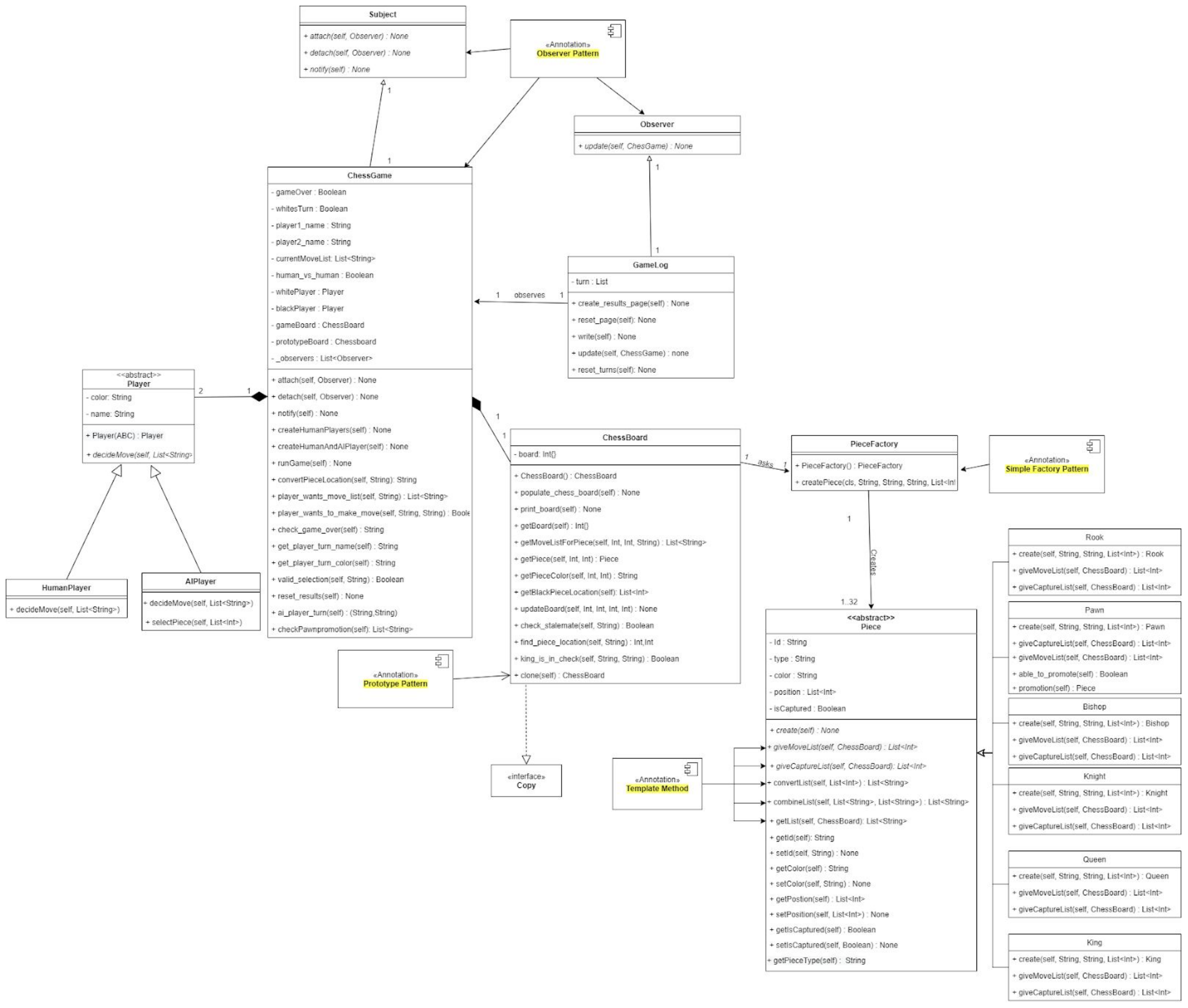
Additionally, the system can run on only keyboard input making it more accessible to those that would need a switch interface. The keyboard's arrows can navigate the chessboard, and the enter key selects. The game also keeps track of all the moves made throughout the game. The *results* button brings the user to a new html page which shows a table of all the moves made throughout the game.

Final Class Diagram and Comparison Statement

Class Diagram from Project 4:


«interface» PropertyChangeListener
BoardListener
+ BoardListener():
+ captureMade(Piece): void
+ moveMade(Piece) : void
+ propertyChange(PropertyChangeEvent): void
«Annotation» Observer Pattern
1
observes
1
<<abstract>> Player
- color: String
- movesMade: Int
- playersTurn: boolean
- Name: String
+ Player(String) :
+ makeMove() : void
2
1
ChessGame
+ black: Player
+ white: Player
+ chessBoard: Board
+ active: Boolean
+ ChessGame():
+ runGame(): String
+ getPlayersSelectedPiece():
1
1
Board
- pieces: Piece[][]
- boardLength: Int
- boardWidth: Int
- factory: PieceFactory
- support: PropertyChangeSupport
+ Board(Int, Int):
+ getPiece(String, boolean) : Piece
+ addPiece(Piece, String) : void
+ getMoves(Piece) : List<String>
+ movePiece(String, String) : void
+ addPropertyChangeListener(PropertyChangeListener) : void
+ removePropertyChangeListener(PropertyChangeListener) : void
+ clone()
asks
1
1
PieceFactory
+ PieceFactory():
+ getPiece(String,Boolean): Piece
+ getPieces(Int, String,Boolean): List<Piece>
«Annotation» Simple Factory Pattern
1
Creates
1..32
HumanChessPlayer
+ HumanChessPlayer(String):
+ makeMove() : void
AIChessPlayer
- difficulty: String
+ AIChessPlayer(String, String) :
+ makeMove(): void
+ decidePieceToMove(): void
«Annotation» Prototype Pattern
«interface» Cloneable
<<abstract>> Piece
- type: String
- isWhite: Boolean
- value: Int
- rank: Int
- file: Char
- position : (Int,Char)
- numOnBoard: Int
- isCaptured: Boolean
- numMovesPiece: Int
- numPiecesAttacked: Int
+ Piece(String, Boolean):
+ getPositionOnBoard(): (int, char)
+ checkIfPositionIsLegal(int,char): boolean
+ hasBeenCaptured(): void
+ updatePosition(int,char): void
+ attack(Piece): void
+ turn(String, Boolean):
+ getMoveList() : List<String>
+ getAttackList() : List<String>
«Annotation» Template Method
Rook
+ Rook(Boolean):
+ getMoveList() : List<String
+ getAttackList(): List<String
Pawn
+ Pawn(Boolean):
+ getMoveList() : List<String>
+ getAttackList(): List<String>
+ promote(): Piece
Bishop
+ Bishop(Boolean):
+ getMoveList() : List<String
+ getAttackList(): List<String
Knight
+ Knight(Boolean):
+ getMoveList() : List<String
+ getAttackList(): List<String
Queen
+ Queen(Boolean):
+ getMoveList() : List<Strin
+ getAttackList(): List<Strin
King
+ King(Boolean):
+ getMoveList() : List<String
+ getAttackList(): List<String

Final Class Diagram:


Subject
+ attach(self, Observer) : None
+ detach(self, Observer) : None
+ notify(self) : None
«Annotation»
Observer Pattern
Observer
+ update(self, ChesGame) : None
ChessGame
- gameOver : Boolean
- whitesTurn : Boolean
- player1_name : String
- player2_name : String
- currentMoveList: List<String>
- human_vs_human : Boolean
- whitePlayer : Player
- blackPlayer : Player
- gameBoard : ChessBoard
- prototypeBoard : Chessboard
- _observers : List<Observer>
+ attach(self, Observer) : None
+ detach(self, Observer) : None
+ notify(self) : None
+ createHumanPlayers(self) : None
+ createHumanAndAIPlayer(self) : None
+ runGame(self) : None
+ convertPieceLocation(self, String): String
+ player_wants_move_list(self, String) : List<String>
+ player_wants_to_make_move(self, String, String) : Boole
+ check_game_over(self) : String
+ get_player_turn_name(self) : String
+ get_player_turn_color(self) : String
+ valid_selection(self, String) : Boolean
+ reset_results(self) : None
+ ai_player_turn(self) : (String,String)
+ checkPawnpromotion(self): List<String>
GameLog
- turn : List
+ create_results_page(self) : None
+ reset_page(self): None
+ write(self) : None
+ update(self, ChessGame) : none
+ reset_turns(self): None
observes
1
1
<<abstract>>
Player
- color: String
- name: String
+ Player(ABC) : Player
+ decideMove(self, List<String>
2
1
HumanPlayer
+ decideMove(self, List<String>)
AIPlayer
+ decideMove(self, List<String>)
+ selectPiece(self, List<Int>)
ChessBoard
- board: Int{}
+ ChessBoard() : ChessBoard
+ populate_chess_board(self) : None
+ print_board(self) : None
+ getBoard(self) : Int{}
+ getMoveListForPiece(self, Int, Int, String) : List<String>
+ getPiece(self, Int, Int) : Piece
+ getPieceColor(self, Int, Int) : String
+ getBlackPieceLocation(self): List<Int>
+ updateBoard(self, Int, Int, Int, Int) : None
+ check_stalemate(self, String) : Boolean
+ find_piece_location(self, String) : Int,Int
+ king_is_in_check(self, String, String) : Boolean
+ clone(self) : ChessBoard
«Annotation»
Prototype Pattern
«interface»
Copy
PieceFactory
+ PieceFactory() : PieceFactory
+ createPiece(cls, String, String, String, List<Int
asks
«Annotation»
Simple Factory Pattern
Creates
1..32
<<abstract>>
Piece
- Id : String
- type : String
- color : String
- position : List<Int>
- isCaptured : Boolean
+ create(self) : None
+ giveMoveList(self, ChessBoard) : List<Int>
+ giveCaptureList(self, ChessBoard): List<Int>
+ convertList(self, List<Int>) : List<String>
+ combineList(self, List<String>, List<String>) : List<String>
+ getList(self, ChessBoard): List<String>
+ getId(self): String
+ setId(self, String) : None
+ getColor(self) : String
+ setColor(self, String) : None
+ getPostion(self) : List<Int>
+ setPosition(self, List<Int>) : None
+ getIsCaptured(self) : Boolean
+ setIsCaptured(self, Boolean) : None
+ getPieceType(self) : String
«Annotation»
Template Method
Rook
+ create(self, String, String, List<Int>) : Rook
+ giveMoveList(self, ChessBoard) : List<Int>
+ giveCaptureList(self, ChessBoard) : List<Int>
Pawn
+ create(self, String, String, List<Int>) : Pawn
+ giveCaptureList(self, ChessBoard) : List<Int>
+ giveMoveList(self, ChessBoard) : List<Int>
+ able_to_promote(self) : Boolean
+ promotion(self) : Piece
Bishop
+ create(self, String, String, List<Int>) : Bishop
+ giveMoveList(self, ChessBoard) : List<Int>
+ giveCaptureList(self, ChessBoard) : List<Int>
Knight
+ create(self, String, String, List<Int>) : Knight
+ giveMoveList(self, ChessBoard) : List<Int>
+ giveCaptureList(self, ChessBoard) : List<Int>
Queen
+ create(self, String, String, List<Int>) : Queen
+ giveMoveList(self, ChessBoard) : List<Int>
+ giveCaptureList(self, ChessBoard) : List<Int>
King
+ create(self, String, String, List<Int>) : King
+ giveMoveList(self, ChessBoard) : List<Int>
+ giveCaptureList(self, ChessBoard) : List<Int>

Key changes: Since our class diagram from project 4 many new variables and methods were added to the class ChessGame in order to support the setup of the game and the actual gameplay. For example, methods such as createHumanPlayers and runGame were needed for constructing the players and chessboard. Other methods have been added as well to ChessGame for the AI player to make moves. Another major change since projects 4 and 5 is that ChessBoard is no longer implementing the Observer Pattern. This pattern has been taken over by a new Class called GameLog which has become the new observer and ChessGame has become the subject. The purpose of GameLog is to be notified whenever a move has been made in the game and who it has been made by.

GameLog then adds this information to the results.html file that can be viewed when the “results” button on the gameplay screen is pressed. Because the classes that are implementing the Observer Pattern have changed, methods such as attach (which attaches an observer) have been moved into and are implemented by the ChessGame class. Two new classes Subject and Observer have been added as well to better utilize the structure of the Observer Pattern. For the template method, convertList and combineList functions have been added to the skeleton of the operation getList(). The functions giveMoveList() and giveCaptureList() have been renamed from their original names in project 4 but are still part of the template method as abstract methods to be handled by the piece subclasses. Since we switched over to Python, the alternative copy was used instead of cloneable.

Third-Party code vs. Original code Statement

- In setting up Flask, https://flask.palletsprojects.com/en/1.1.x/tutorial/, provided examples and references
- https://www.programcreek.com/python/example/9977/jsonpickle.encode provided examples of encoding and decoding JSON objects
- https://devcenter.heroku.com/articles/getting-started-with-python had tutorials on deploying with flask
- https://www.w3schools.com/js/DEFAULT.asp gave many examples using html, css, and JavaScript
- https://refactoring.guru/design-patterns/observer/python/example provided some code segments for the structure of the Observer design pattern in Python (creating a list of observers, attaching, and detaching them, and how to notify them)
- https://aaravtech.medium.com/design-patterns-in-python-factory-c728b88603eb provided structure for the Factory Pattern design in Python
- https://www.onenaught.com/posts/382/firefox-4-change-input-type-image-only-submits-x-and-y-not-name#toc-workarounds Helped fix huge bug in the beginning where some code in HTML wouldn't work in Chrome
- https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html use for helping with the AI logic
- All of the logic for the different pieces are original code, although we got the rules for the different chess pieces from https://www.chess.com/terms/chess-pieces and https://en.wikipedia.org/wiki/Rules_of_chess

Statement on the OOAD process for your overall Semester Project

1. We originally wanted to deploy our website using Amazon Web Services since none of us had used that before, and we hoped it would be a good learning experience. However, AWS has a very complicated system for hosting a website through them. It became extremely difficult and confusing, so we instead used Heroku to host the website.
2. We weren't sure what web framework we were going to use for the website. Neither of us were too comfortable with building web applications, so it was a challenge to decide which framework would work best for our requirements. We ended up using Flask since it supported python on the backend. Using Flask had quite the learning curve at the beginning stages of our development process. However, we soon picked it up, and were able to communicate easily between the backend and frontend.
3. Since we had a clear understanding of the requirements and the structure of our system before we started coding, it was fairly easy to implement all our classes and get them communicating properly. We got the chess game, chess board and chess pieces updating properly each turn without any major obstacles. This may have also been in part to the good communication efforts our team had during this project. We did a good job updating each other as to what we were working on, and the state each part of the system was in. This made the design process fairly easy.