Project #6

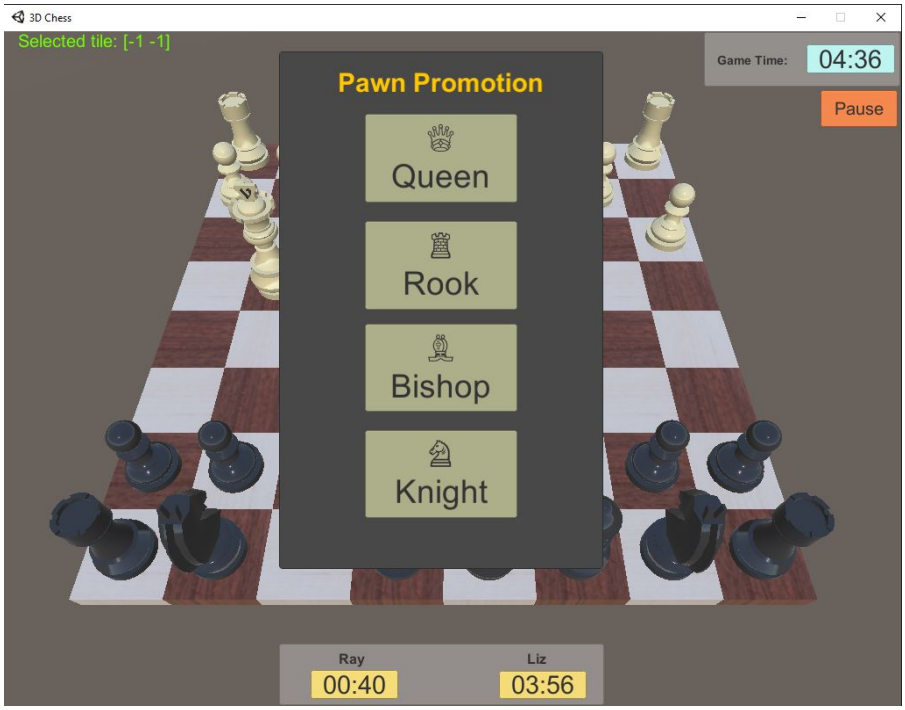
Semester Project - Final Submission

12/9/2019
Project: 3D Chess Desktop Application

Team Members
Ray Shash
Elizabeth Robinson
Joshua Khoo

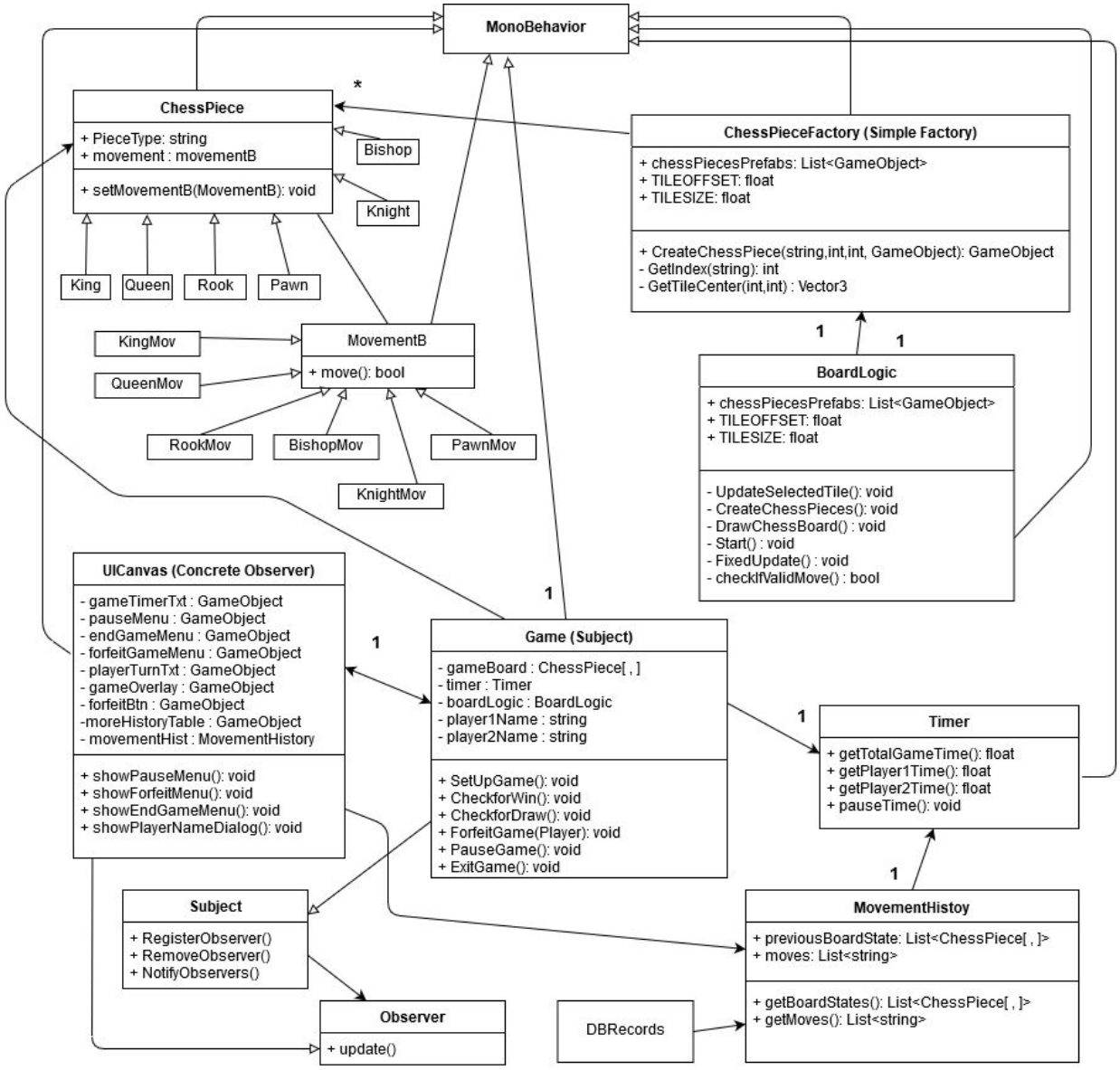
Final State of System
Due to the lack of time and difficulties encountered while implementing aspects like pseudo legal movement, we were not able to implement aspects of this project that were originally planned. This included setting up an external database where past game would be stored, saving the game state and resuming gameplay, displaying the current plays in a table adjacent to the chess board in algebraic notation. We do however have a menu system and the **user can play a fully fledged chess game with another opponent (local two player)**. We also added support and recognition of castling, en passant , which was a challenge to implement. We have a "Pre-Release" in the release section in the GitHub repository.

All legal movements and capture moves show up on the board!



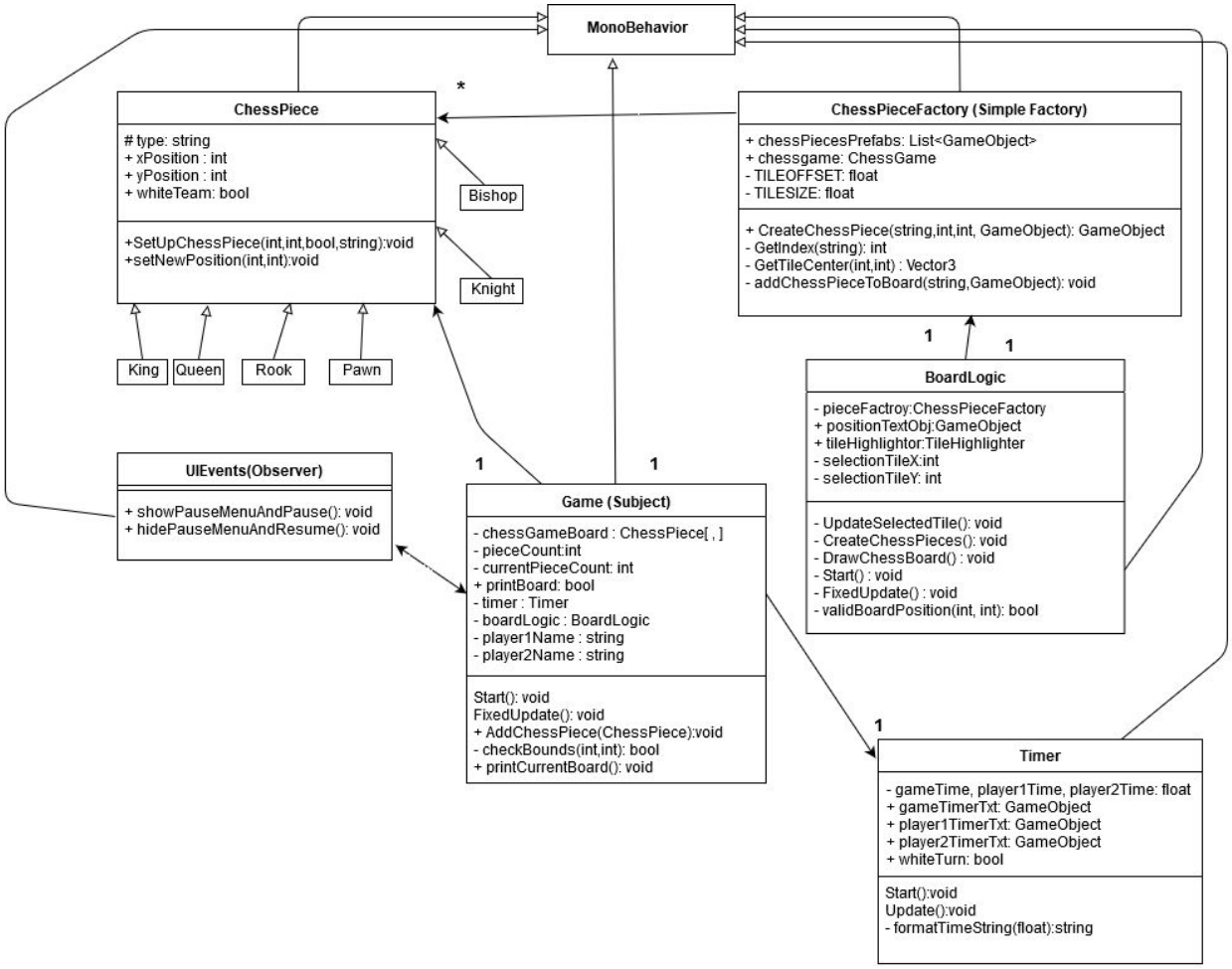
Class Diagrams and Comparison statements

This is the UML diagram from out project proposal (**Project #4**).



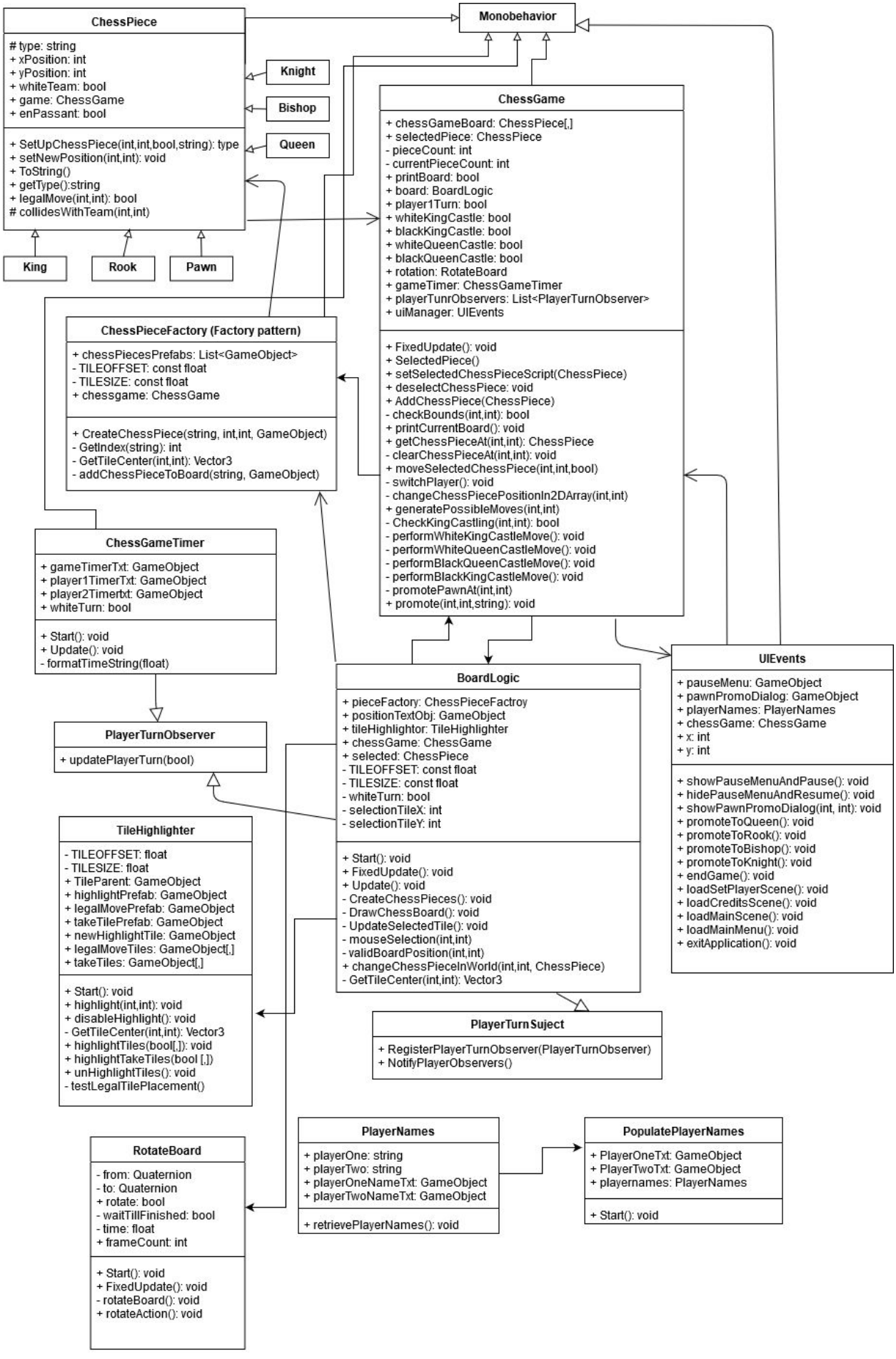
As can be seen from the UML diagram above, we were a little too ambitious with the inclusion of movement history, DBRecords. We did not end up using the strategy pattern to delegate legal movements and taking and relied on inheritance instead (We felt that this was not necessary here and we did not want to force a pattern) . It is not that different from our final design...

This is the UML diagram from our “first sprint” (Project #5).



After the first sprint, the initial grandwork, model and setup was completed. It was now up to implementing chess piece movements, legal movements, highlighting, taking etc. That overall structure remained the same. More classes and methods were added.

This is the UML diagram for the current status of the project (Project #6) follows:


ChessPiece
type: string
+ xPosition: int
+ yPosition: int
+ whiteTeam: bool
+ game: ChessGame
+ enPassant: bool
+ SetUpChessPiece(int,int,bool,string): type
+ setNewPosition(int,int): void
+ ToString()
+ getType():string
+ legalMove(int,int): bool
# collidesWithTeam(int,int)
Monobehavior
Knight
Bishop
Queen
King
Rook
Pawn
ChessGame
+ chessGameBoard: ChessPiece[,]
+ selectedPiece: ChessPiece
- pieceCount: int
- currentPieceCount: int
+ printBoard: bool
+ board: BoardLogic
+ player1Turn: bool
+ whiteKingCastle: bool
+ blackKingCastle: bool
+ whiteQueenCastle: bool
+ blackQueenCastle: bool
+ rotation: RotateBoard
+ gameTimer: ChessGameTimer
+ playerTunrObservers: List<PlayerTurnObserver>
+ uiManager: UIEvents
+ FixedUpdate(): void
+ SelectedPiece()
+ setSelectedChessPieceScript(ChessPiece)
+ deselectChessPiece: void
+ AddChessPiece(ChessPiece)
- checkBounds(int,int): bool
+ printCurrentBoard(): void
+ getChessPieceAt(int,int): ChessPiece
- clearChessPieceAt(int,int): void
+ moveSelectedChessPiece(int,int,bool)
- switchPlayer(): void
- changeChessPiecePositionIn2DArray(int,int)
+ generatePossibleMoves(int,int)
- CheckKingCastling(int,int): bool
- performWhiteKingCastleMove(): void
- performWhiteQueenCastleMove(): void
- performBlackQueenCastleMove(): void
- performBlackKingCastleMove(): void
- promotePawnAt(int,int)
+ promote(int,int,string): void
ChessPieceFactory (Factory pattern)
+ chessPiecesPrefabs: List<GameObject>
- TILEOFFSET: const float
- TILESIZE: const float
+ chessgame: ChessGame
+ CreateChessPiece(string, int,int, GameObject)
- GetIndex(string): int
- GetTileCenter(int,int): Vector3
- addChessPieceToBoard(string, GameObject)
ChessGameTimer
+ gameTimerTxt: GameObject
+ player1TimerTxt: GameObject
+ player2Timertxt: GameObject
+ whiteTurn: bool
+ Start(): void
+ Update(): void
- formatTimeString(float)
BoardLogic
+ pieceFactory: ChessPieceFactroy
+ positionTextObj: GameObject
+ tileHighlightor: TileHighlighter
+ chessGame: ChessGame
+ selected: ChessPiece
- TILEOFFSET: const float
- TILESIZE: const float
- whiteTurn: bool
- selectionTileX: int
- selectionTileY: int
+ Start(): void
+ FixedUpdate(): void
+ Update(): void
- CreateChessPieces(): void
- DrawChessBoard(): void
- UpdateSelectedTile(): void
- mouseSelection(int,int)
- validBoardPosition(int,int)
+ changeChessPieceInWorld(int,int, ChessPiece)
- GetTileCenter(int,int): Vector3
UIEvents
+ pauseMenu: GameObject
+ pawnPromoDialog: GameObject
+ playerNames: PlayerNames
+ chessGame: ChessGame
+ x: int
+ y: int
+ showPauseMenuAndPause(): void
+ hidePauseMenuAndResume(): void
+ showPawnPromoDialog(int, int): void
+ promoteToQueen(): void
+ promoteToRook(): void
+ promoteToBishop(): void
+ promoteToKnight(): void
+ endGame(): void
+ loadSetPlayerScene(): void
+ loadCreditsScene(): void
+ loadMainScene(): void
+ loadMainMenu(): void
+ exitApplication(): void
PlayerTurnObserver
+ updatePlayerTurn(bool)
TileHighlighter
- TILEOFFSET: float
- TILESIZE: float
+ TileParent: GameObject
+ highlightPrefab: GameObject
+ legalMovePrefab: GameObject
+ takeTilePrefab: GameObject
+ newHighlightTile: GameObject
+ legalMoveTiles: GameObject[,]
+ takeTiles: GameObject[,]
+ Start(): void
+ highlight(int,int): void
+ disableHighlight(): void
- GetTileCenter(int,int): Vector3
+ highlightTiles(bool[,]): void
+ highlightTakeTiles(bool [,])
+ unHighlightTiles(): void
- testLegalTilePlacement()
PlayerTurnSuject
+ RegisterPlayerTurnObserver(PlayerTurnObserver)
+ NotifyPlayerObservers()
RotateBoard
- from: Quaternion
- to: Quaternion
+ rotate: bool
- waitTillFinished: bool
- time: float
+ frameCount: int
+ Start(): void
+ FixedUpdate(): void
- rotateBoard(): void
+ rotateAction(): void
PlayerNames
+ playerOne: string
+ playerTwo: string
+ playerOneNameTxt: GameObject
+ playerTwoNameTxt: GameObject
+ retrievePlayerNames(): void
PopulatePlayerNames
+ PlayerOneTxt: GameObject
+ PlayerTwoTxt: GameObject
+ playernames: PlayerNames
+ Start(): void

It appears that we underestimated the complexity of creating this chess game. We ended up adding more methods and classes (aka scripts). We decided against implementing strategy pattern as mentioned above due to it not being necessary.

Pattern usage:

1. Factory Pattern: ChessPieceFactory is a factory that can create chessPieces from a set of prefabs. There are many convenient functions to easily create and place chess pieces on the board. This was helpful not only in setting up the chess board, but also for pawn promotion.
2. Observer Pattern: Observer pattern was implemented so many classes can observe the current player's turn. In this case the subject is ChessGame (the main class that handles the actual chess game) while ChessGameTimer & BoardLogic are the observers.

Third-Party Code & Sources

The app was developed using the Unity engine. Assets taken from:
https://www.raywenderlich.com/.
Assets included:
1 - Chess Pieces
2 - Chess Board

Information about how to do Raycasting with unity:
https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

Used inspiration from this video on setting up a "virtual debug board" and Raycasting (used by BoardLogic):
https://www.youtube.com/watch?v=CzImJk7ZesI

Constantly used the Unity Documentation for various tasks. There are always examples provided that simplified understanding.
https://docs.unity3d.com/ScriptReference/index.html

Format string to look like time (used in ChessGameTimer):
https://answers.unity.com/questions/45676/making-a-timer-0000-minutes-and-seconds.html

OOAD Design Process

1. It was a challenge to maintain high cohesion (classes should have specific roles) and loose coupling (minimal dependency). At times it was necessary to violate either of these rules in order to prevent major refactoring of code. We did try to enforce high cohesion even though it yielded more code. In our class structure, we have a BoardLogic and a ChessGame class. BoardLogic's sole responsibility is to handle the infrastructure and mechanisms necessary to deal with the 3D graphical portion of the chess game. This included mouse interaction, selection, the board coordinates etc. ChessGame's responsibility is to handle everything involved to play chess game including checking of legal movements, special movements like castling or en passant, players

taking turns, etc. At times it became obvious that separation of these two functions caused us to refer to both classes a lot and make them both very dependent.

2. We had many patterns in mind when working on this project. In the end we only used two concrete patterns (two were only necessary in the scope of our project anyway). We took good advice from a previous lecture where the usage of software patterns is not mandated or intended to be shoehorned into a project. Creating a factory for chess pieces and using observer pattern for the current players turn was an obvious design decision. The incorporation of more patterns like strategy would just add more complexity to this project it was not necessary. Strategy pattern was considered as a way to “encapsulate what varies” and to “favor composition over inheritance”. Since the chess pieces do not change their behavior and we used the ChessPieceFactory to do pawn promotion anyway, we decided to not incorporate strategy pattern.
3. Just with any software project or big plans, it sometimes does not end up the way you want to. We underestimated the time it would take us to implement basic chess piece movement, legal move and capture detection and special chess movements like castling and en passant. There was also a learning curve at first. One of us never used C# or Unity before while the reset had only basic prior experience. While trying to implement various aspects, we also were also learning. This prevented us from meeting with our original requirements.

Github Repository

https://github.com/OOAD-Team-Veritas/SemesterProject-3DChess

Release

Download and unzip the “3D.Chess.OOAD.zip file”. Open the directory and click on “3D Chess”.exe to run the program.

https://github.com/OOAD-Team-Veritas/SemesterProject-3DChess/releases