Project 5 Documentation

Scott Rizzo

12/08/18

GitHub: smrizzo

1 Design Patterns

1.1 Singleton Pattern

When creating the pieces I used the Singleton pattern to create those pieces. I have a abstract class I named Piece which doesn't have much currently but was designed in a way to prepare for the future. It currently only has a character variable for each individual piece. The Piece class has many subclasses that extend it and all those classes are Singletons. So I currently have a blackPawn, blackKnight, blackRook, blackBishop, blackQueen, blackKing, whitePawn, whiteKnight, whiteRook, whiteBishop, whiteQueen, whiteKing, Player1Marble, Player2Marble, Player3Marble, Player4Marble, Player5Marble, Player6Marble, and EmptyBoardSlot classes that are all singletons. This was designed to only have one object at any given time of each piece as well as make it easy to add more pieces if ever wanted to change the game of chess which actually just happened when adding PlayerMarble pieces for Chinese Checkers.

These PlayerMarble classes I added with p4 are to represent PlayerPieces on the Chinese Checkers Board. I was able use these classes for representing player pieces on the board so when a move happened I was updating which marble slot on the board was referencing the player marble single class. I also used these Player Marble classes to identify which state we were in which I will talk about in the state pattern section.

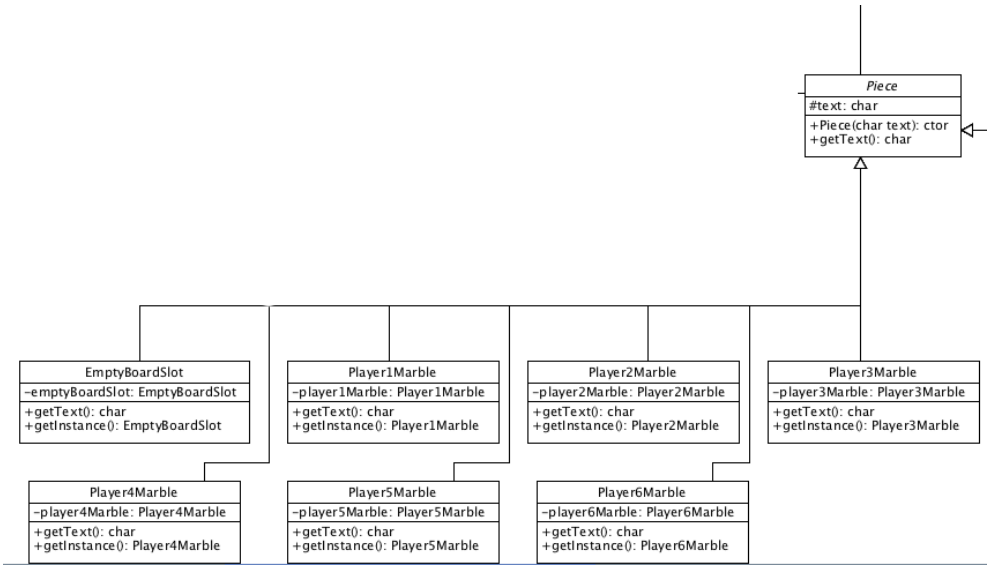


Figure 1: Singleton classes

1.2 Flyweight Pattern

When creating my 2d array of type Piece for Chess and Marbles I had in mind the Flyweight pattern. It appears on the board visually I have many pieces when in fact I really only have one pawn piece objects but using the Flyweight pattern my 2d array is referencing the same object over and over again. For Chinese Checkers was a little different I instead have a 2d array of Marble objects and each Marble object has a reference to the Player Marble classes but still the same Idea. There is not much to say for this pattern besides that the Flyweight and Singleton kind of work well together and it allows me to use less memory.

```
private Marble[][] marbles = new Marble[17][25];
private Piece player;
```

Figure 2: flyweight

1.3 Observer Pattern

For chess I designed it using the observer pattern so its very flexible and can add new Views very easily preparing for the future. I used this pattern so that I could decouple my View and Controller from my Model. I made each View and controller object an observer for the model which was the observable. The view never changed or updated the board when pertaining to pieces moving or getting promoted until the model called the update method inside using the observer pattern.

For Chinese Checkers was the same idea but was little different because we didn't use spritely. So instead I had CCViewObserver was for the StarUI View, and CCController. I also had a CompObserver which was for the Component that StarUI View created. The reason for having to different observers was because in the StarModel I wanted to send the updated model to the Component class before the StarUI because the StarUI class was where I invoked a repaint method but before that happened I needed to make sure the Component had the updated model first. So when a action happened in Chinese Checkers I first notified all the Component Classes that have been created which there will be a Component object per StarUI Object created, that something has changed, then I update all the StarUI and CCController objects with the updated model.

A action happens in the view, the Controller decides that the action is something I can care about, then manipulates the model changing the board, then the model notifies all the observers that the board has changed. The views then re-render to display the new board.

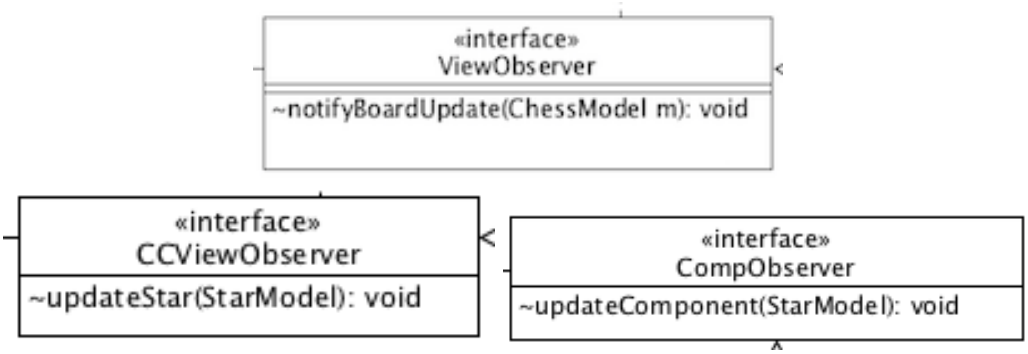


Figure 3: Observer Interface

1.4 Command Pattern

I used the Command pattern for making a move in Chinese Checkers. I wasn't able to refactor p3 with the Command pattern since it took all the time I had just to get p4 where it is. It made sense to use it for moving a piece because so could encapsulate the move and undoMove invocation in the same class which was nice. Some actions I didn't really think were necessary for command pattern class for instance just setting the, FromXY and ToXY destinations. When I have some time though I would like to add a command pattern for initialClick that way when a initial click happened I could just invoke execute on that class which then would then invoke methods to setFromXY, GetAllMoves, and perhaps others. I had some issues with trying to do that so will have to figure that out.

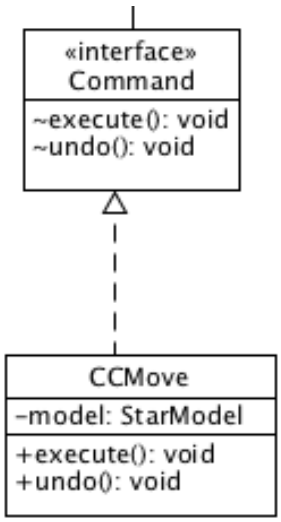


Figure 4: Command Pattern

1.5 State Pattern

State pattern was used for Chinese Checkers for when it was a specific players turn. When it was player 1's turn there was restrictions on what pieces that player was able to click initially. This was also useful because the view would display things different based on what state it was, so if it was player1 state then it would display specific colors, text, etc.. It was also useful for when I needed to store a move which was the moveSnapshot class. I was able to store which state the board was in which was very useful for undoLastMove because when I popped the move off that stack I was able to just grab the state of the board at that time.

```
final static Piece PLAYER_1 = Player1Marble.getInstance();
final static Piece PLAYER_2 = Player2Marble.getInstance();
final static Piece PLAYER_3 = Player3Marble.getInstance();
final static Piece PLAYER_4 = Player4Marble.getInstance();
final static Piece PLAYER_5 = Player5Marble.getInstance();
final static Piece PLAYER_6 = Player6Marble.getInstance();
private Piece state = null;
```

Figure 5: State Pattern

1.6 Strategy Pattern

I used the Strategy pattern to really try and encapsulate each different task within its own class or objects. I refactored to better encapsulate specific task and through composition allowing other classes to gain access to other variables or methods that are not there own. I do feel I could have done better here but was struggling to do it with everything going on. I currently have a single Controller for both chess local, chess network, and Checkers local. It might be better to separate those three different controllers into a ChessNetworkController, ChessLocalController, ChineseCheckersLocalController which are subclasses of a main Controller abstract class. This would allow easily being able to add new controllers if ever needed to for the future. Would also be better If did something similar for my model classes. I will be working on improving this for p5.

1.7 MVC Pattern

The way I designed my project with MVC is by creating a ChessModel class, and Chess Controller Class, and a View class. Then for Chinese Checkers a StarModel class, CCController class, StarUI(view class) and StarComponent(view class) . The view class is where all my animations are taking place and essentially everything that has to do with the GUI. This is where all the setClickHandlers reside and I when a click or key press happens it lets the Controller decide what to do with that action. If at some point there is an action that happens that is meaningful to the game of chess my Controller manipulates the ChessModel by invoking methods in the ChessModel class.

1.7.1 The Model

The model in my program is defined by the ChessModel. The chess model class contains a 2d array that has references to Piece objects using the Flyweight and Singleton patter explained above. All the board logic as far as moving a piece from location to the next all happens in the model. It has methods like regularMove, promoteWhitePawn, promoteBlackPawn, promotingPawn etc.... It also the appropriate setter and getting methods so that the controller can set things up in the model when certain actions happen. This allows the model to not really know what's going in the View or the Controller it just provides methods to be invoked by whoever which in this case is the Controller.

The model for Chinese Checkers has a lot more going on. It has a 2d array of Marble objects which then reference a player object using the Flyweight and Singleton pattern explained above. All the board logic for Chinese Checkers is in the model which includes, initializing the the marbles and there adjacent marbles, starting piece locations, players home and destination triangles, making a move, undoing a move, getting all possible moves, checking to see if a player has won the game, keeping track of moveHistory using the moveSnapshot class, and much more. All the heavy lifting of board logic is done here.

1.7.2 The Controller

The controller in my program is defined by the ChessController class. The chess controller at this point has two responsibilities which I understand is not the best but considering the struggle to complete this assignment it will be something I have to fix later on. The controller as of right now is responsible for creating View objects and adding those to a List of threads. When the time is right we invoke the start() method on those threads which invoke the run() method in each View. The controller is really the middleware that was is responsible for manipulating the model and in some cases setting up booleans for the view to know when to highlight etc. Pretty much same idea for checkers but it doesn't handle network play and is much more light weight.

1.7.3 The View

My view is responsible for how the board will look on the screen. It does not communicate with the Model in anyway. The View has two subclasses which are WhiteView, and BlackView. The WhiteView creates the board for when a white player is on the bottom of the board, and the blackView creates a board for when its black player and the pieces need to be on the bottom. Likewise for Chinese Checkers we have a StarUI class which is the main View where all the mouseClick listeners reside, and it creates a component object

which does all the painting for how the View look mostly. The view gets the updated board from the Model through the observer pattern and then re-renders the window or component based on where it finds certain piece objects in the 2d array given by the model. The model has no idea about how its being displayed it just says here you go View do what you will with this 2d array of pieces or marbles. The view for chess is the only one that has access to the image files stored in the HashMap in a ChessPieces class which i will have to rename in the future.

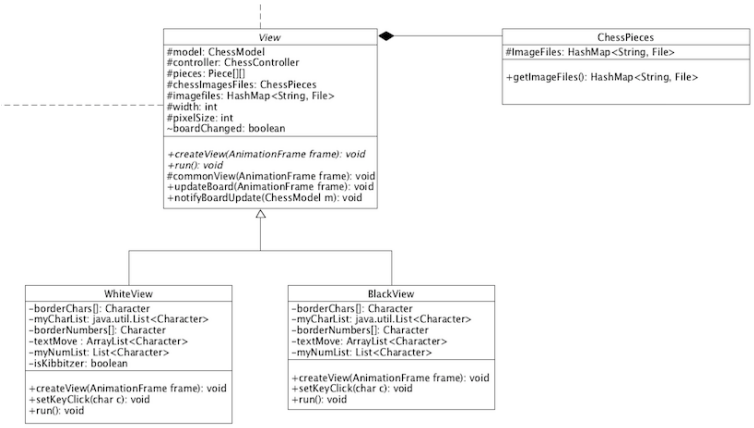


Figure 6: ChessViewClasses

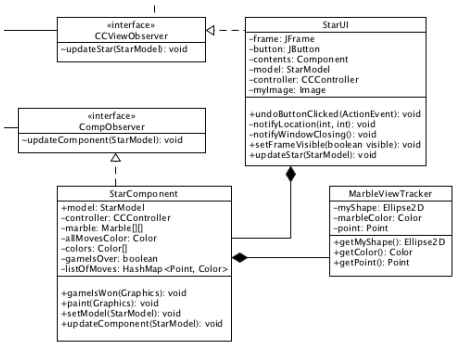
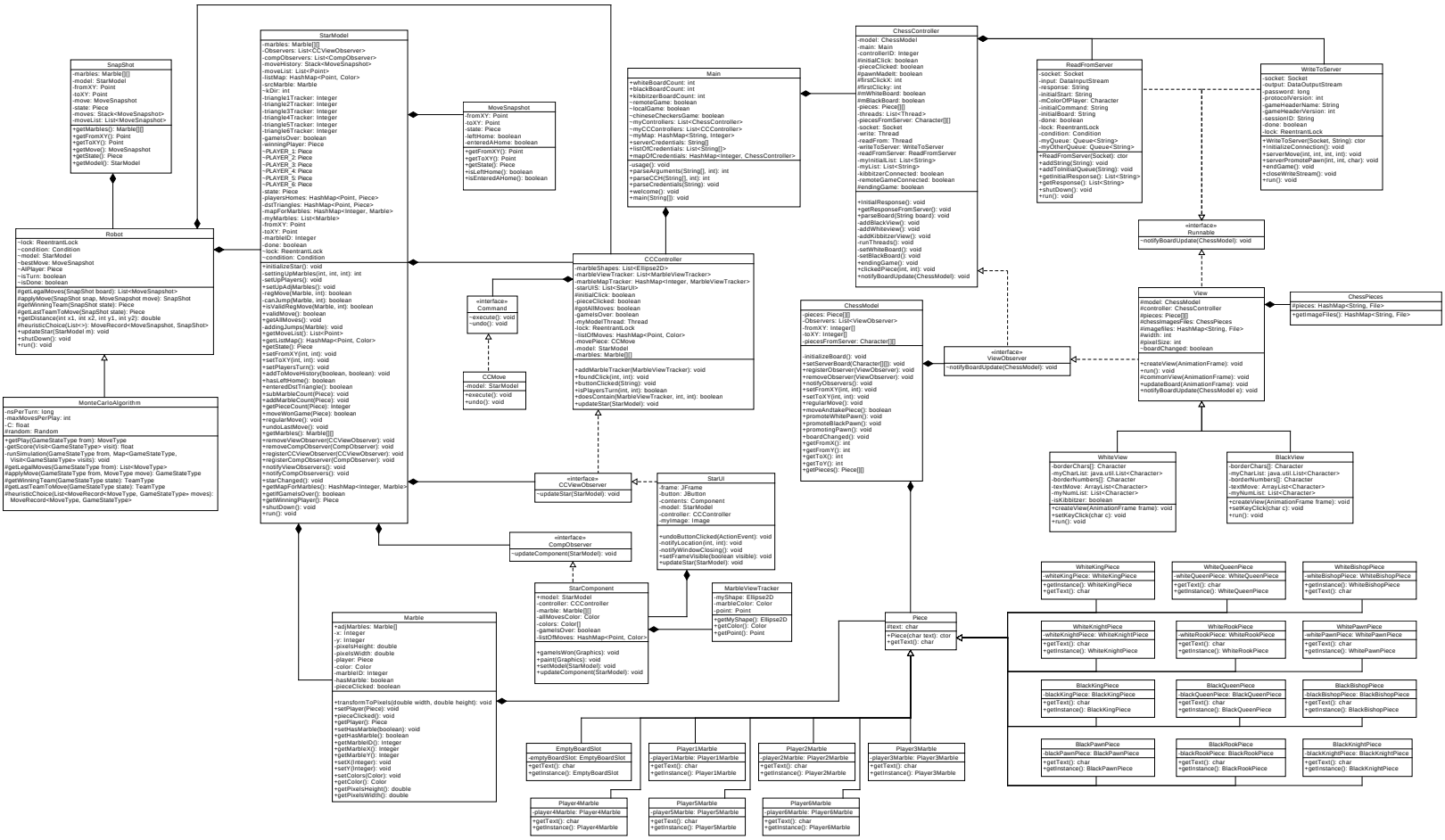


Figure 7: CheckersViewClasses

1.7.4 Robots

For this project we had to add a robot which is in the class Robot which extends the MonteCarloAlgorithm class. Robot has various methods I needed to implement as seen in class diagram. I also had to create a snapshot class which store a copy of the model at given point in time that the MonteCarloAlgorithm could use when trying to decide best possible move. In main had to parse the command line for information pertaining to robots and certain game boards, which then we gave to the controller. The controller then created Robots on there own threads for each game and ran those. When its there turn they will go, and until then they will wait. They also play on existing UI's with other players.

1.8 Class Diagram


SnapShot
-marbles: Marble[][]
-model: StarModel
-fromXY: Point
-toXY: Point
-move: MoveSnapshot
-state: Piece
-moves: Stack<MoveSnapshot>
-movesList: List<MoveSnapshot>
+getMarbles(): Marble[][]
+getFromXY(): Point
+getToXY(): Point
+getMove(): MoveSnapshot
+getState(): Piece
+getModel(): StarModel
StarModel
-marbles: Marble[][]
-Observers: List<CCViewObserver>
-compObservers: List<CompObserver>
-moveHistory: Stack<MoveSnapshot>
-movesList: List<Point>
-listMap: HashMap<Point, Color>
-srcMarble: Marble
-kDir: int
-triangle1Tracker: Integer
-triangle2Tracker: Integer
-triangle3Tracker: Integer
-triangle4Tracker: Integer
-triangle5Tracker: Integer
-triangle6Tracker: Integer
-gameIsOver: boolean
-winningPlayer: Piece
~PLAYER_1: Piece
~PLAYER_2: Piece
~PLAYER_3: Piece
~PLAYER_4: Piece
~PLAYER_5: Piece
~PLAYER_6: Piece
-state: Piece
-playersHomes: HashMap<Point, Piece>
-dstTriangles: HashMap<Point, Piece>
-mapForMarbles: HashMap<Integer, Marble>
-myMarbles: List<Marble>
-fromXY: Point
-toXY: Point
-marbleID: Integer
-done: boolean
~lock: ReentrantLock
~condition: Condition
+initializeStar(): void
-settingUpMarbles(int, int, int): int
-setUpPlayers(): void
+setUpAllMarbles(): void
-regMove(Marble, int): boolean
-canJump(Marble, int): boolean
+isValidRegMove(Marble, int): boolean
+validMove(): boolean
+getAllMoves(): void
-addingJumps(Marble): void
+getMovesList(): List<Point>
+getListMap(): HashMap<Point, Color>
+getState(): Piece
+setFromXY(int, int): void
+setToXY(int, int): void
+setPlayersTurn(): void
+addToMoveHistory(boolean, boolean): void
+hasLeftHome(): boolean
+enteredDstTriangle(): boolean
+subMarbleCount(Piece): void
+addMarbleCount(Piece): void
+getPieceCount(Piece): Integer
+moveWonGame(Piece): boolean
+regularMove(): void
+undoLastMove(): void
+getMarbles(): Marble[][]
+removeViewObserver(CCViewObserver): void
+removeCompObserver(CompObserver): void
+registerCCViewObserver(CCViewObserver): void
+registerCompObserver(CompObserver): void
+notifyViewObservers(): void
+notifyCompObservers(): void
+starChanged(): void
+getMapForMarbles(): HashMap<Integer, Marble>
+getIfGameIsOver(): boolean
+getWinningPlayer(): Piece
+shutDown(): void
+run(): void
MoveSnapshot
-fromXY: Point
-toXY: Point
-state: Piece
-leftHome: boolean
-enteredAHome: boolean
+getFromXY(): Point
+getToXY(): Point
+getState(): Piece
+isLeftHome(): boolean
+isEnteredAHome(): boolean
Main
+whiteBoardCount: int
+blackBoardCount: int
+kibitzerBoardCount: int
-remoteGame: boolean
-localGame: boolean
-chineseCheckersGame: boolean
-myControllers: List<ChessController>
-myCCControllers: List<CCCcontroller>
-myMap: HashMap<String, Integer>
+serverCredentials: String[]
+ListOfCredentials: List<String[]>
+mapOfCredentials: HashMap<Integer, ChessController>
-usage(): void
+parseArguments(String[], int): int
+parseCCH(String[], int): int
+parseCredentials(String): void
+welcome(): void
+main(String[]): void
ChessController
-model: ChessModel
-main: Main
-controllerID: Integer
#initialClick: boolean
-pieceClicked: boolean
#pawnMadeIt: boolean
#firstClickX: int
#firstClickY: int
#mWhiteBoard: boolean
#mBlackBoard: boolean
-pieces: Piece[][]
-threads: List<Thread>
-piecesFromServer: Character[][]
-socket: Socket
-write: Thread
-readFrom: Thread
-writeToServer: WriteToServer
-readFromServer: ReadFromServer
-myInitialList: List<String>
-myList: List<String>
-kibbitzerConnected: boolean
-remoteGameConnected: boolean
#endingGame: boolean
+initialResponse(): void
+getResponseFromServer(): void
+parseBoard(String board): void
-addBlackView(): void
-addWhiteview(): void
-addKibbitzerView(): void
-runThreads(): void
-setWhiteBoard(): void
-setBlackBoard(): void
+endingGame(): void
+clickedPiece(int, int): void
+notifyBoardUpdate(ChessModel): void
ReadFromServer
-socket: Socket
-input: DataInputStream
-response: String
-initialStart: String
-mColorOfPlayer: Character
-initialCommand: String
-initialBoard: String
-done: boolean
-lock: ReentrantLock
-condition: Condition
-myQueue: Queue<String>
-myOtherQueue: Queue<String>
+ReadFromServer(Socket): ctor
+addString(String): void
+addToInitialQueue(String): void
+getInitialResponse(): List<String>
+getResponse(): List<String>
+shutDown(): void
+run(): void
WriteToServer
-socket: Socket
-output: DataOutputStream
-password: long
-protocolVersion: int
-gameHeaderName: String
-gameHeaderVersion: int
-sessionID: String
-done: boolean
-lock: ReentrantLock
+WriteToServer(Socket, String): ctor
+initializeConnection(): void
+serverMove(int, int, int, int): void
+serverPromotePawn(int, int, char): void
+endGame(): void
+closeWriteStream(): void
+run(): void
«interface»
Runnable
~notifyBoardUpdate(ChessModel): void
Robot
~lock: ReentrantLock
~condition: Condition
~model: StarModel
~bestMove: MoveSnapshot
-AIPlayer: Piece
~isTurn: boolean
~isDone: boolean
#getLegalMoves(SnapShot board): List<MoveSnapshot>
#applyMove(SnapShot snap, MoveSnapshot move): SnapShot
#getWinningTeam(SnapShot state): Piece
#getLastTeamToMove(SnapShot state): Piece
+getDistance(int x1, int x2, int y1, int y2): double
#heuristicChoice(List<>): MoveRecord<MoveSnapshot, SnapShot>
+updateStar(StarModel m): void
+shutDown(): void
+run(): void
«interface»
Command
~execute(): void
~undo(): void
CCMove
-model: StarModel
+execute(): void
+undo(): void
CCController
-marbleShapes: List<Ellipse2D>
-marbleViewTracker: List<MarbleViewTracker>
-marbleMapTracker: HashMap<Integer, MarbleViewTracker>
-starUIS: List<StarUI>
#initialClick: boolean
-pieceClicked: boolean
#goAllMoves: boolean
-gameIsOver: boolean
-myModelThread: Thread
-lock: ReentrantLock
~listOfMoves: HashMap<Point, Color>
-movePiece: CCMove
-model: StarModel
-marbles: Marble[][]
+addMarbleTracker(MarbleViewTracker): void
+foundClick(int, int): void
+buttonClicked(String): void
+isPlayersTurn(int, int): boolean
+doesContainMarbleViewTracker, int, int): boolean
+updateStar(StarModel): void
ChessModel
-pieces: Piece[][]
-Observers: List<ViewObserver>
-fromXY: Integer[]
-toXY: Integer[]
-piecesFromServer: Character[][]
-initializeBoard(): void
+setServerBoard(Character[][]): void
+registerObserver(ViewObserver): void
+removeObserver(ViewObserver): void
+notifyObservers(): void
+setFromXY(int, int): void
+setToXY(int, int): void
+regularMove(): void
+moveAndtakePiece(): boolean
+promoteWhitePawn(): void
+promoteBlackPawn(): void
+promotingPawn(): void
+boardChanged(): void
+getFromX(): int
+getFromY(): int
+getToX(): int
+getToY(): int
+getPieces(): Piece[][]
«interface»
ViewObserver
~notifyBoardUpdate(ChessModel): void
View
#model: ChessModel
#controller: ChessController
#pieces: Piece[][]
#chessImagesFiles: ChessPieces
#imagefiles: HashMap<String, File>
#width: int
#pixelSize: int
~boardChanged: boolean
+createView(AnimationFrame): void
+run(): void
#commonView(AnimationFrame): void
+updateBoard(AnimationFrame): void
+notifyBoardUpdate(ChessModel e): void
ChessPieces
#pieces: HashMap<String, File>
+getImageFiles(): HashMap<String, File>
WhiteView
-borderChars[]: Character
-myCharList: java.util.List<Character>
-borderNumbers[]: Character
-textMove: ArrayList<Character>
-myNumList: List<Character>
-isKibitzer: boolean
+createView(AnimationFrame frame): void
+setKeyClick(char c): void
+run(): void
BlackView
-borderChars[]: Character
-myCharList: java.util.List<Character>
-borderNumbers[]: Character
-textMove: ArrayList<Character>
-myNumList: List<Character>
+createView(AnimationFrame frame): void
+setKeyClick(char c): void
+run(): void
MonteCarloAlgorithm
-nsPerTurn: long
-maxMovesPerPlay: int
-C: float
#random: Random
+getPlay(GameStateType from): MoveType
-getScore(Visit<GameStateType> visit): float
-runSimulation(GameStateType from, Map<GameStateType, Visit<GameStateType> visits): void
#getLegalMoves(GameStateType from): List<MoveType>
#applyMove(GameStateType from, MoveType move): GameStateType
#getWinningTeam(GameStateType state): TeamType
#getLastTeamToMove(GameStateType state): TeamType
#heuristicChoice(List<MoveRecord<MoveType, GameStateType> moves): MoveRecord<MoveType, GameStateType>
«interface»
CCViewObserver
~updateStar(StarModel): void
StarUI
-frame: JFrame
-button: JButton
-contents: Component
-model: StarModel
-controller: CCController
-myImage: Image
+undoButtonClicked(ActionEvent): void
-notifyLocation(int, int): void
-notifyWindowClosing(): void
+setFrameVisible(boolean visible): void
+updateStar(StarModel): void
«interface»
CompObserver
~updateComponent(StarModel): void
StarComponent
+model: StarModel
-controller: CCController
-marble: Marble[][]
-allMovesColor: Color
-colors: Color[]
-gameIsOver: boolean
-listOfMoves: HashMap<Point, Color>
+gameIsWon(Graphics): void
+paint(Graphics): void
+setModel(StarModel): void
+updateComponent(StarModel): void
MarbleViewTracker
-myShape: Ellipse2D
-marbleColor: Color
-point: Point
+getMyShape(): Ellipse2D
+getColor(): Color
+getPoint(): Point
Marble
+adjMarbles: Marble[]
-x: Integer
-y: Integer
-pixelsHeight: double
-pixelsWidth: double
-player: Piece
-color: Color
-marbleID: Integer
-hasMarble: boolean
-pieceClicked: boolean
+transformToPixels(double width, double height): void
+setPlayer(Piece): void
+pieceClicked(): void
+getPlayer(): Piece
+setHasMarble(boolean): void
+getHasMarble(): boolean
+getMarbleID(): Integer
+getMarbleX(): Integer
+getMarbleY(): Integer
+setX(Integer): void
+setY(Integer): void
+setColors(Color): void
+getColor(): Color
+getPixelsHeight(): double
+getPixelsWidth(): double
Piece
#text: char
+Piece(char text): ctor
+getText(): char
WhiteKingPiece
-whiteKingPiece: WhiteKingPiece
+getInstance(): WhiteKingPiece
+getText(): char
WhiteQueenPiece
-whiteQueenPiece: WhiteQueenPiece
+getText(): char
+getInstance(): WhiteQueenPiece
WhiteBishopPiece
-whiteBishopPiece: WhiteBishopPiece
+getInstance(): WhiteBishopPiece
+getText(): char
WhiteKnightPiece
-whiteKnightPiece: WhiteKnightPiece
+getText(): char
+getInstance(): WhiteKnightPiece
WhiteRookPiece
-whiteRookPiece: WhiteRookPiece
+getText(): char
+getInstance(): WhiteRookPiece
WhitePawnPiece
-whitePawnPiece: WhitePawnPiece
+getText(): char
+getInstance(): WhitePawnPiece
BlackKingPiece
-blackKingPiece: BlackKingPiece
+getText(): char
+getInstance(): BlackKingPiece
BlackQueenPiece
-blackQueenPiece: BlackQueenPiece
+getText(): char
+getInstance(): BlackQueenPiece
BlackBishopPiece
-blackBishopPiece: BlackBishopPiece
+getText(): char
+getInstance(): BlackBishopPiece
BlackPawnPiece
-blackPawnPiece: BlackPawnPiece
+getText(): char
+getInstance(): BlackPawnPiece
BlackRookPiece
-blackRookPiece: BlackRookPiece
+getText(): char
+getInstance(): BlackRookPiece
BlackKnightPiece
-blackKnightPiece: BlackKnightPiece
+getText(): char
+getInstance(): BlackKnightPiece
EmptyBoardSlot
-emptyBoardSlot: EmptyBoardSlot
+getText(): char
+getInstance(): EmptyBoardSlot
Player1Marble
-player1Marble: Player1Marble
+getText(): char
+getInstance(): Player1Marble
Player2Marble
-player2Marble: Player2Marble
+getText(): char
+getInstance(): Player2Marble
Player3Marble
-player3Marble: Player3Marble
+getText(): char
+getInstance(): Player3Marble
Player4Marble
-player4Marble: Player4Marble
+getText(): char
+getInstance(): Player4Marble
Player5Marble
-player5Marble: Player5Marble
+getText(): char
+getInstance(): Player5Marble
Player6Marble
-player6Marble: Player6Marble
+getText(): char
+getInstance(): Player6Marble