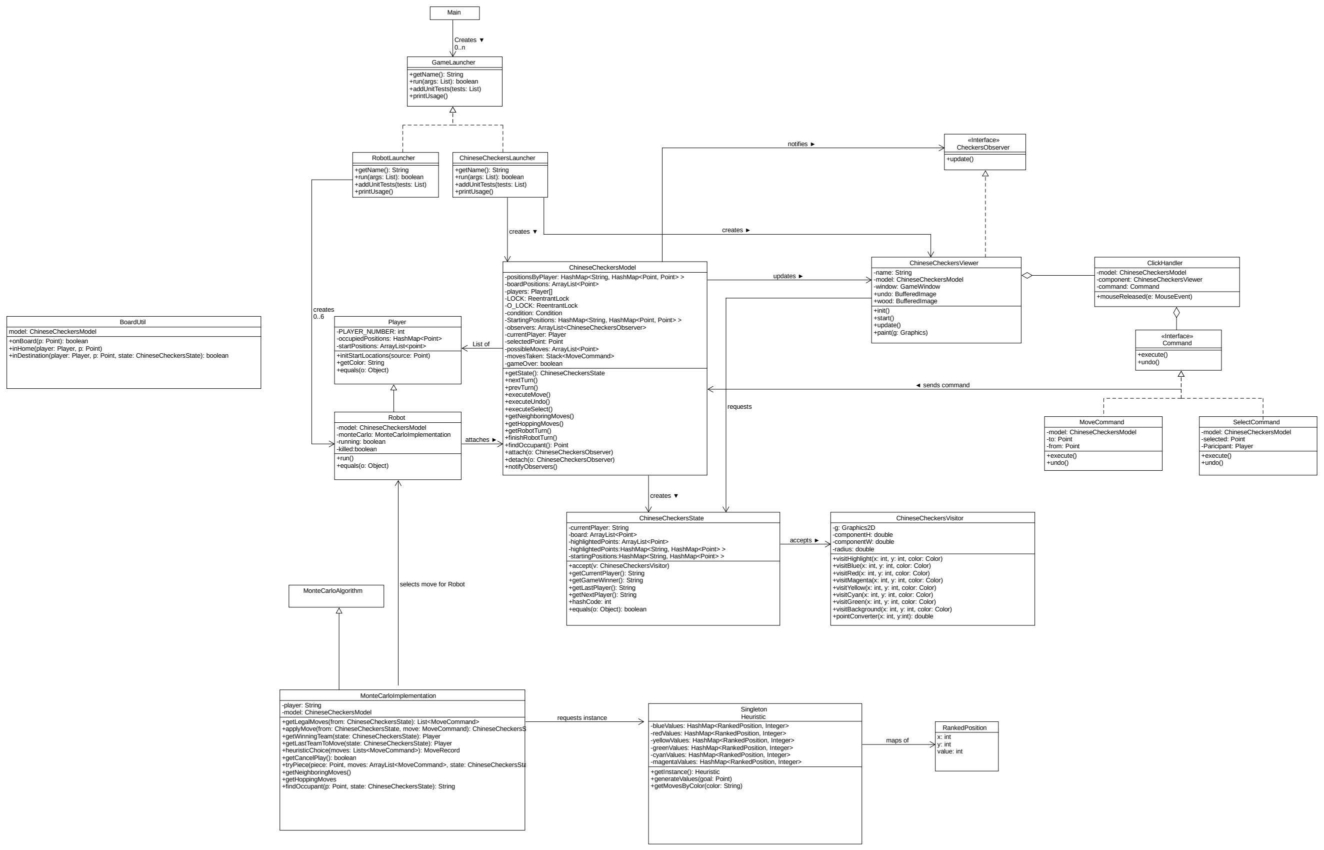
James Asbury
Assignment 5 Design Document
12 March 2019

Overview of entire design:


Main
Creates ▼
0..n
GameLauncher
+getName(): String
+run(args: List): boolean
+addUnitTests(tests: List)
+printUsage()
RobotLauncher
+getName(): String
+run(args: List): boolean
+addUnitTests(tests: List)
+printUsage()
ChineseCheckersLauncher
+getName(): String
+run(args: List): boolean
+addUnitTests(tests: List)
+printUsage()
creates ▼
creates ►
notifies ►
«Interface»
CheckersObserver
+update()
ChineseCheckersModel
-positionsByPlayer: HashMap<String, HashMap<Point, Point> >
-boardPositions: ArrayList<Point>
-players: Player[]
-LOCK: ReentrantLock
-O_LOCK: ReentrantLock
-condition: Condition
-StartingPositions: HashMap<String, HashMap<Point, Point> >
-observers: ArrayList<ChineseCheckersObserver>
-currentPlayer: Player
-selectedPoint: Point
-possibleMoves: ArrayList<Point>
-movesTaken: Stack<MoveCommand>
-gameOver: boolean
+getState(): ChineseCheckersState
+nextTurn()
+prevTurn()
+executeMove()
+executeUndo()
+executeSelect()
+getNeighboringMoves()
+getHoppingMoves()
+getRobotTurn()
+finishRobotTurn()
+findOccupant(): Point
+attach(o: ChineseCheckersObserver)
+detach(o: ChineseCheckersObserver)
+notifyObservers()
updates ►
ChineseCheckersViewer
-name: String
-model: ChineseCheckersModel
-window: GameWindow
+undo: BufferedImage
+wood: BufferedImage
+init()
+start()
+update()
+paint(g: Graphics)
ClickHandler
-model: ChineseCheckersModel
-component: ChineseCheckersViewer
-command: Command
+mouseReleased(e: MouseEvent)
«Interface»
Command
+execute()
+undo()
◄ sends command
requests
MoveCommand
-model: ChineseCheckersModel
-to: Point
-from: Point
+execute()
+undo()
SelectCommand
-model: ChineseCheckersModel
-selected: Point
-Paricipant: Player
+execute()
+undo()
BoardUtil
model: ChineseCheckersModel
+onBoard(p: Point): boolean
+inHome(player: Player, p: Point)
+inDestination(player: Player, p: Point, state: ChineseCheckersState): boolean
creates
0..6
Player
-PLAYER_NUMBER: int
-occupiedPositions: HashMap<Point>
-startPositions: ArrayList<point>
+initStartLocations(source: Point)
+getColor: String
+equals(o: Object)
List of
Robot
-model: ChineseCheckersModel
-monteCarlo: MonteCarloImplementation
-running: boolean
-killed:boolean
+run()
+equals(o: Object)
attaches ►
creates ▼
ChineseCheckersState
-currentPlayer: String
-board: ArrayList<Point>
-highlightedPoints: ArrayList<Point>
-highlightedPoints:HashMap<String, HashMap<Point> >
-startingPositions:HashMap<String, HashMap<Point> >
+accept(v: ChineseCheckersVisitor)
+getCurrentPlayer(): String
+getGameWinner(): String
+getLastPlayer(): String
+getNextPlayer(): String
+hashCode: int
+equals(o: Object): boolean
accepts ►
ChineseCheckersVisitor
-g: Graphics2D
-componentH: double
-componentW: double
-radius: double
+visitHighlight(x: int, y: int, color: Color)
+visitBlue(x: int, y: int, color: Color)
+visitRed(x: int, y: int, color: Color)
+visitMagenta(x: int, y: int, color: Color)
+visitYellow(x: int, y: int, color: Color)
+visitCyan(x: int, y: int, color: Color)
+visitGreen(x: int, y: int, color: Color)
+visitBackground(x: int, y: int, color: Color)
+pointConverter(x: int, y:int): double
MonteCarloAlgorithm
selects move for Robot
MonteCarloImplementation
-player: String
-model: ChineseCheckersModel
+getLegalMoves(from: ChineseCheckersState): List<MoveCommand>
+applyMove(from: ChineseCheckersState, move: MoveCommand): ChineseCheckersS
+getWinningTeam(state: ChineseCheckersState): Player
+getLastTeamToMove(state: ChineseCheckersState): Player
+heuristicChoice(moves: Lists<MoveCommand>): MoveRecord
+getCancelPlay(): boolean
+tryPiece(piece: Point, moves: ArrayList<MoveCommand>, state: ChineseCheckersSta
+getNeighboringMoves()
+getHoppingMoves
+findOccupant(p: Point, state: ChineseCheckersState): String
requests instance
Singleton
Heuristic
-blueValues: HashMap<RankedPosition, Integer>
-redValues: HashMap<RankedPosition, Integer>
-yellowValues: HashMap<RankedPosition, Integer>
-greenValues: HashMap<RankedPosition, Integer>
-cyanValues: HashMap<RankedPosition, Integer>
-magentaValues: HashMap<RankedPosition, Integer>
+getInstance(): Heuristic
+generateValues(goal: Point)
+getMovesByColor(color: String)
maps of
RankedPosition
x: int
y: int
value: int

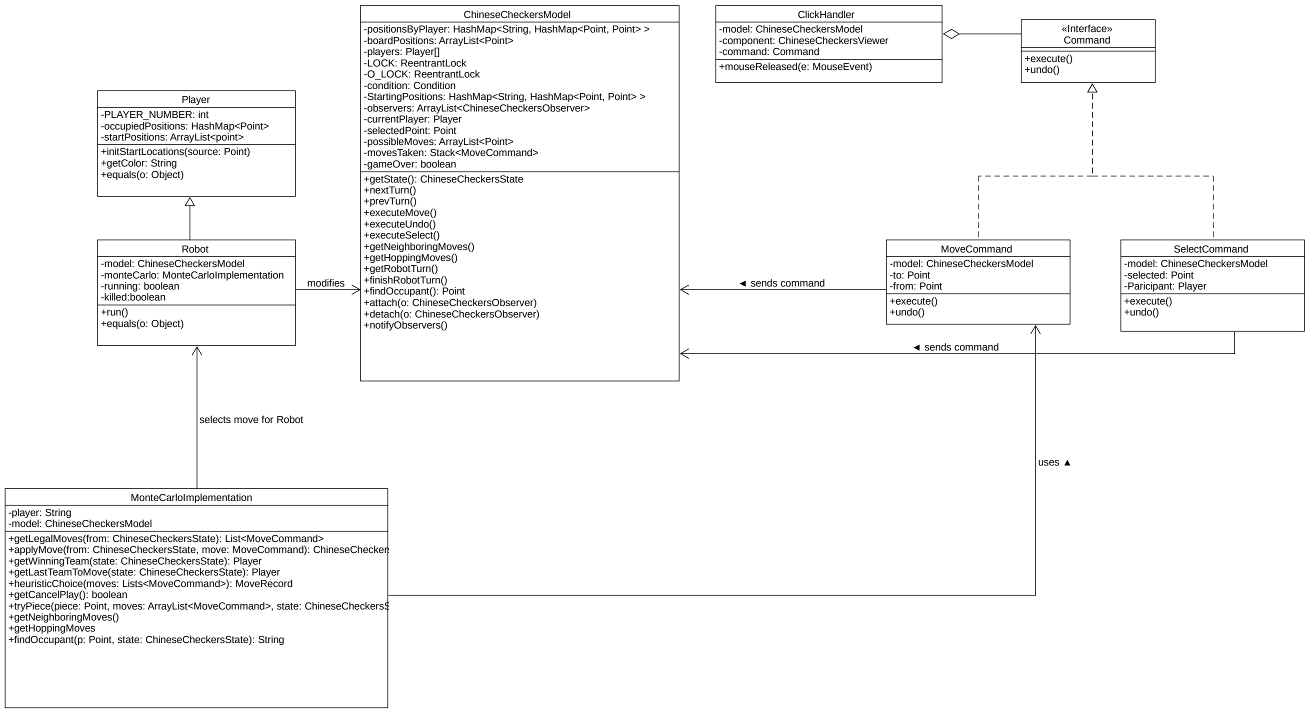
Class Breakdown:

Model:
Although the model may appear large and complex, it has one purpose in accordance to the single responsibility principle. It manages the positions and changes in position of the robots and players. Efficiency is important to the model! Movement should happen very fast depending on how fast the robots work. And undo should appear instantaneous to prevent confusion. Because of this, board locations are stored in a HashMap. String player names (“red”, “blue”, etc.) are hashed to another map of all of the point locations that that specific player occupies. This provides quick lookup for checking to see if a player occupies a specific point. When a point is moved, it can also quickly be removed and the new location hashed in. The model contains a locking system for the board (LOCK) and a locking system for the observers (O_LOCK). Methods are carefully designed to avoid deadlocks and to make sparing but effective use of locking. Lock calls were minimized to specific methods that either read or write from shared data.

Robots:
Different robots, when launched by the RobotLauncher, can “attach” themselves to the model. They then update the model when it’s their turn. There is also a condition inside the model used to let the robots know when it is their turn to move. This prevents the robots from “eating up the CPU” as they would if they were constantly checking to see if it’s their turn to move. They instead patiently wait until signalAll tells them it’s their turn.

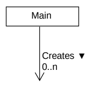
ChineseCheckersState:
Many precautions were undertaken to ensure this class is immutable. All data members (Lists, HashMaps, etc.) were recreated and new Points were created to ensure no references exist to mutable data. Instead of storing any references to players, the state uses string identifiers (“red”, “blue”, etc.). These can then be converted back into player objects by other classes such as the Monte Carlo Algorithm classes.

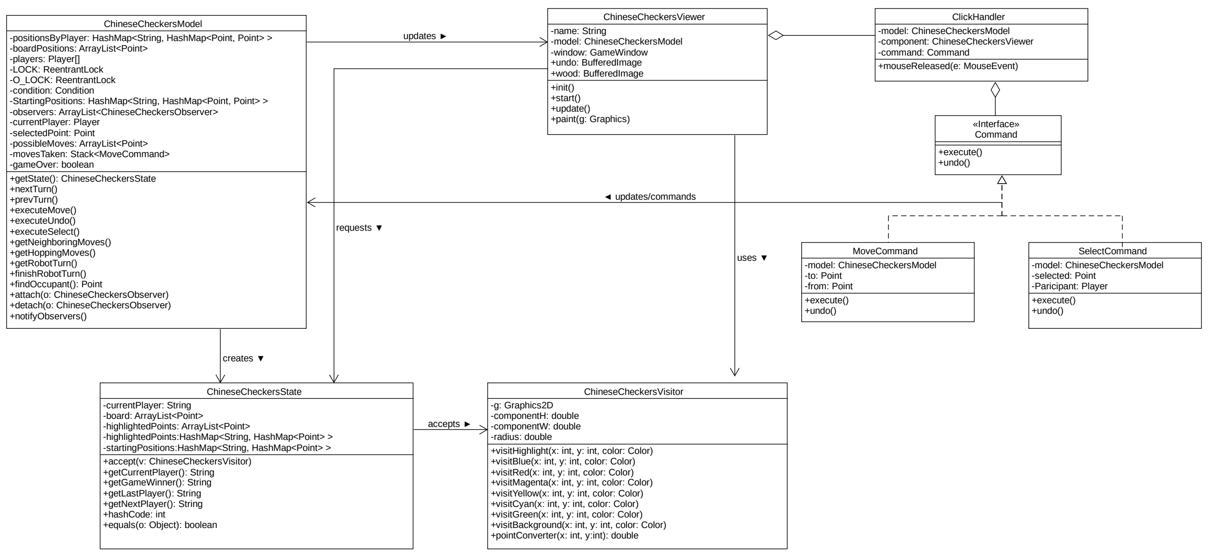
Command Pattern UML:


ChineseCheckersModel
-positionsByPlayer: HashMap<String, HashMap<Point, Point> >
-boardPositions: ArrayList<Point>
-players: Player[]
-LOCK: ReentrantLock
-O_LOCK: ReentrantLock
-condition: Condition
-StartingPositions: HashMap<String, HashMap<Point, Point> >
-observers: ArrayList<ChineseCheckersObserver>
-currentPlayer: Player
-selectedPoint: Point
-possibleMoves: ArrayList<Point>
-movesTaken: Stack<MoveCommand>
-gameOver: boolean
+getState(): ChineseCheckersState
+nextTurn()
+prevTurn()
+executeMove()
+executeUndo()
+executeSelect()
+getNeighboringMoves()
+getHoppingMoves()
+getRobotTurn()
+finishRobotTurn()
+findOccupant(): Point
+attach(o: ChineseCheckersObserver)
+detach(o: ChineseCheckersObserver)
+notifyObservers()
ClickHandler
-model: ChineseCheckersModel
-component: ChineseCheckersViewer
-command: Command
+mouseReleased(e: MouseEvent)
«Interface»
Command
+execute()
+undo()
Player
-PLAYER_NUMBER: int
-occupiedPositions: HashMap<Point>
-startPositions: ArrayList<point>
+initStartLocations(source: Point)
+getColor: String
+equals(o: Object)
Robot
-model: ChineseCheckersModel
-monteCarlo: MonteCarloImplementation
-running: boolean
-killed:boolean
+run()
+equals(o: Object)
modifies
◄ sends command
MoveCommand
-model: ChineseCheckersModel
-to: Point
-from: Point
+execute()
+undo()
SelectCommand
-model: ChineseCheckersModel
-selected: Point
-Paricipant: Player
+execute()
+undo()
◄ sends command
selects move for Robot
uses ▲
MonteCarloImplementation
-player: String
-model: ChineseCheckersModel
+getLegalMoves(from: ChineseCheckersState): List<MoveCommand>
+applyMove(from: ChineseCheckersState, move: MoveCommand): ChineseChecker
+getWinningTeam(state: ChineseCheckersState): Player
+getLastTeamToMove(state: ChineseCheckersState): Player
+heuristicChoice(moves: Lists<MoveCommand>): MoveRecord
+getCancelPlay(): boolean
+tryPiece(piece: Point, moves: ArrayList<MoveCommand>, state: ChineseCheckersS
+getNeighboringMoves()
+getHoppingMoves
+findOccupant(p: Point, state: ChineseCheckersState): String

Command Pattern
The command pattern was used for all manual player actions (eg. those not executed by a robot). ClickHandler notices a click took place, and decides what type of command that click should represent. It creates and calls execute on that command. The move command translates between the original piece location and the new one, and calls the proper corresponding methods in the model to make this change. A stack of MoveCommand objects is maintained in order to implement undo. When undo is pressed, the previous move is pulled off the stack and reversed.
The Monte Carlo Algorithm will make use of these MoveCommand objects as well as a place to test and store different moves before a final “best choice” is selected.

Visitor Pattern and MVC:




ChineseCheckersModel
-positionsByPlayer: HashMap<String, HashMap<Point, Point> >
-boardPositions: ArrayList<Point>
-players: Player[]
-LOCK: ReentrantLock
-O_LOCK: ReentrantLock
-condition: Condition
-StartingPositions: HashMap<String, HashMap<Point, Point> >
-observers: ArrayList<ChineseCheckersObserver>
-currentPlayer: Player
-selectedPoint: Point
-possibleMoves: ArrayList<Point>
-movesTaken: Stack<MoveCommand>
-gameOver: boolean
+getState(): ChineseCheckersState
+nextTurn()
+prevTurn()
+executeMove()
+executeUndo()
+executeSelect()
+getNeighboringMoves()
+getHoppingMoves()
+getRobotTurn()
+finishRobotTurn()
+findOccupant(): Point
+attach(o: ChineseCheckersObserver)
+detach(o: ChineseCheckersObserver)
+notifyObservers()
updates ►
ChineseCheckersViewer
-name: String
-model: ChineseCheckersModel
-window: GameWindow
+undo: BufferedImage
+wood: BufferedImage
+init()
+start()
+update()
+paint(g: Graphics)
ClickHandler
-model: ChineseCheckersModel
-component: ChineseCheckersViewer
-command: Command
+mouseReleased(e: MouseEvent)
«Interface»
Command
+execute()
+undo()
◄ updates/commands
requests ▼
uses ▼
MoveCommand
-model: ChineseCheckersModel
-to: Point
-from: Point
+execute()
+undo()
SelectCommand
-model: ChineseCheckersModel
-selected: Point
-Paricipant: Player
+execute()
+undo()
creates ▼
ChineseCheckersState
-currentPlayer: String
-board: ArrayList<Point>
-highlightedPoints: ArrayList<Point>
-highlightedPoints:HashMap<String, HashMap<Point> >
-startingPositions:HashMap<String, HashMap<Point> >
+accept(v: ChineseCheckersVisitor)
+getCurrentPlayer(): String
+getGameWinner(): String
+getLastPlayer(): String
+getNextPlayer(): String
+hashCode: int
+equals(o: Object): boolean
accepts ►
ChineseCheckersVisitor
-g: Graphics2D
-componentH: double
-componentW: double
-radius: double
+visitHighlight(x: int, y: int, color: Color)
+visitBlue(x: int, y: int, color: Color)
+visitRed(x: int, y: int, color: Color)
+visitMagenta(x: int, y: int, color: Color)
+visitYellow(x: int, y: int, color: Color)
+visitCyan(x: int, y: int, color: Color)
+visitGreen(x: int, y: int, color: Color)
+visitBackground(x: int, y: int, color: Color)
+pointConverter(x: int, y:int): double

Visitor and MVC Pattern

The visitor pattern and MVC are implemented similarly to previous projects in this class. The visitor pattern is used to decouple the model from the view.

Heuristic and Monte Carlo

The heuristic for my project uses a singleton Heuristic class. This singleton class pre-calculates different "score" values for each location on the board based on how far those locations are from the opposite side. These values are calculated using breadth first search starting at the farthest opposite point. Scores are stored as RankedPosition objects. Six maps of these RankedPositions exist, one for each of the six players. A red robot for example can select from the singleton class the redValues data member, and analyze the various point values of different moves in order to choose the best one in the time it is given.

Future Design thoughts and Program Adaptability

The setup of the project uses many different classes and as little coupling between classes as possible to allow the design to be adaptable. If future commands (such as pause?) were added, they could be incorporated with the existing Command Pattern structure. It could tell the model that it is paused. Clicks and other actions would be disabled for the viewer, and nextPlayer locked with a variable so that robots cannot place their move. If multiplayer networking was added, commands read from the socket could create moveCommand objects that would be handled in the same way by the model. Our

Heuristic class is decoupled from other classes, so it could easily be modified to include more advanced point calculation without affecting other classes. The observer pattern exists, and although the ChineseCheckersViewer is currently the only class to implement it, other classes could be added in the future that make use of this. Perhaps a kibitzer?

In summary, our design is inherently adaptable due to its use of design patterns, multiple classes, high cohesion within classes, and low coupling between classes. Improvements and additions to the game would be easy and require minimal reworking of existing code.