Architectural Improvement Summary

"BetaGo"

Proposal for Architectural Change

After reflecting on the current source code and architectural design for BetaGo, it appears that the program could benefit from improving and redesigning the loose MVC pattern that has already been attempted. Since BetaGo seems to be a project that a pair of students worked on in their leisure time, architecture may not have been a concept that was being considered during the developmental process, it is common students create games such as BetaGo in order to test their programming skills, rather than develop on higher level concepts such as architecture.

The current architectural issue is that there is a limited separation of concerns in the system because of the View and Controller aspects of the MVC being compressed into the Main class of BetaGo. Main consists of roughly 600 lines of code and has 10 nested classes, which are all responsible for different aspects of the View and Controller. Having this many nested classes seems a little unnecessary since they don't seem to necessarily **need** to be nested classes. Therefore, I propose that an architectural change that would benefit the system would be to extract the nested classes from within the main, and to separate the View from the Controller in order to promote separation of concerns.

By implementing this change, I would be dividing out the BetaGo's tasks more evenly so that each component in the system would be responsible for only a small number of things, rather than acting as a God Class and having an overwhelming number of responsibilities. This would achieve the desire of having overall higher cohesion in the system, which is regarded as being important in the overall design of a program. Following the MVC pattern is also considered to be good architecture because it de-tangles what could otherwise be "spaghetti code" that is hard to read and make sense of and splits it into three distinct components. Making this modification would be particularly beneficial to the programmers of BetaGo because it makes the code a lot easier to digest and understand, but also to build upon in future which is important because the project is currently active and is still being developed.

The modifications that I am proposing will bear no harm to the existing functionalities of BetaGo, and will simply enhance its readability and extensibility to improve the overall architecture of the code.

The Architectural Change

By changing the architecture of BetaGo, I will not be affecting any of the functionalities of the program, and will simply be improving the program's overall design. The program should run exactly the same as it did previously, and all previous passing unit tests will still pass. My aim was to separate the concerns of the system more appropriately so that the code would be more extensible and maintainable.

An obvious starting point for improving the architecture of BetaGo was to split all the nested classes in the Main class into their own independent classes. These classes would make up the View part of the MVC, and then I would create a Controller to control these View classes. The nested classes have now been removed from the Main class and put in their own package, titled "View". Previously, the Main class was responsible for controlling all of the View as well as the Controller, and was the largest class by far, at 600 lines. Since the classes have been extracted, Main is now only 34 lines long and simply runs the start method to begin the program, rather than having an unnecessary number of nested classes and having too many responsibilities. I then transformed the GameView class into my Controller and renamed it GameViewController, since it seemed to be performing no View functions at all, and put it in a separate Controller package.

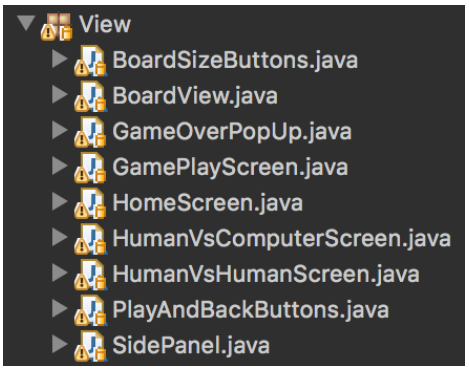

Figure 1

Of course when splitting program up into its MVC components, the classes did not automatically compile and fit together, so I had to do a lot of restructuring in order for compilation to be successful. Initially, the Game object was being created in the main class, so all nested classes were able to make references straight to the Model (since the Game object is in the Model), but since redesigning the system all references from View to the Game object are now directed through the Controller first, which acts as a middle man of communication. So now in order to make any UI changes, the View classes communicate with the GameViewController class, which then communicates to the Model classes to ferry the information back to the View. I did this because it cleans up the mess of communication that was present in the original implementation of BetaGo, so there is now a very clear and structured flow of communication between all components of the system.

Figure 2 (below) shows the original class diagram of BetaGo, whereas Figure 3 shows the class diagram after I have implemented a correct MVC into the system. It is clear to see that the new implementation is a lot cleaner because the responsibilities between classes are far more shared and cohesion has increased. There is a clear divide between the Model, View and Controller, compared to the initial diagram in which the division is almost indistinguishable because of Main acting as a god class. The communication between the components is also a lot more restricted, and there is no direct communication between the Model and the View, with the Controller being the intermediary of all this communication. This cleans up the code a lot more and reduces the spaghetti-code-like structure of the first class diagram.

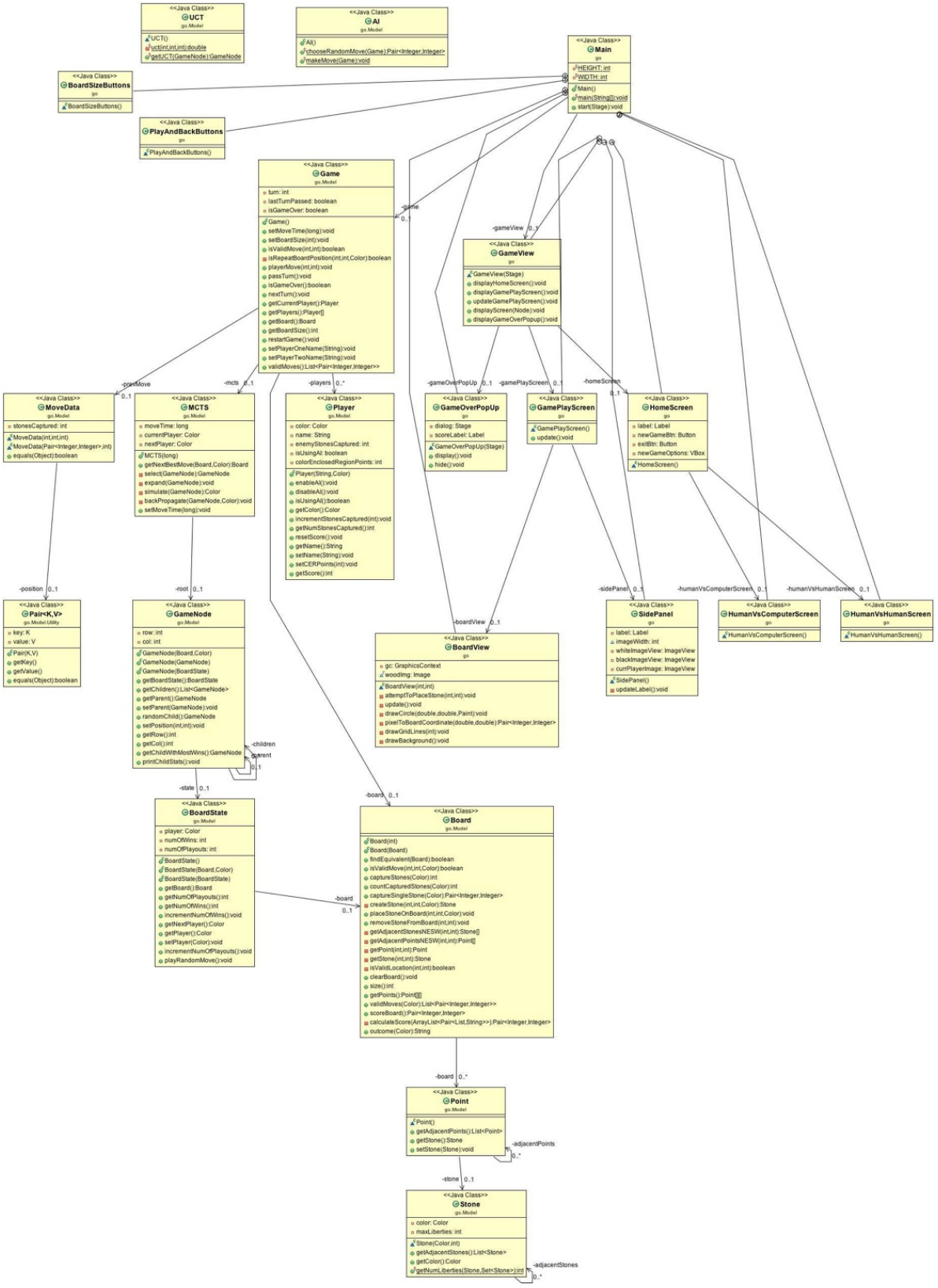
<<Java Class>>
UCT
go.Model
UCT()
uct(int,int,int):double
getUCT(GameNode):GameNode
<<Java Class>>
AI
go.Model
AI()
chooseRandomMove(Game):Pair<Integer,Integer>
makeMove(Game):void
<<Java Class>>
Main
go
HEIGHT: int
WIDTH: int
Main()
main(String[]):void
start(Stage):void
<<Java Class>>
BoardSizeButtons
go
BoardSizeButtons()
<<Java Class>>
PlayAndBackButtons
go
PlayAndBackButtons()
<<Java Class>>
Game
go.Model
turn: int
lastTurnPassed: boolean
isGameOver: boolean
Game()
setMoveTime(long):void
setBoardSize(int):void
isValidMove(int,int):boolean
isRepeatBoardPosition(int,int,Color):boolean
playerMove(int,int):void
passTurn():void
isGameOver():boolean
nextTurn():void
getCurrentPlayer():Player
getPlayers():Player[]
getBoard():Board
getBoardSize():int
restartGame():void
setPlayerOneName(String):void
setPlayerTwoName(String):void
validMoves():List<Pair<Integer,Integer>>
-game
0..1
-gameView 0..1
<<Java Class>>
GameView
go
GameView(Stage)
displayHomeScreen():void
displayGamePlayScreen():void
updateGamePlayScreen():void
displayScreen(Node):void
displayGameOverPopup():void
-prevMove
0..1
-mcts 0..1
-players 0..*
-gameOverPopUp 0..1
-gamePlayScreen 0..1
-homeScreen
0..1
<<Java Class>>
MoveData
go.Model
stonesCaptured: int
MoveData(int,int,int)
MoveData(Pair<Integer,Integer>,int)
equals(Object):boolean
<<Java Class>>
MCTS
go.Model
moveTime: long
currentPlayer: Color
nextPlayer: Color
MCTS(long)
getNextBestMove(Board,Color):Board
select(GameNode):GameNode
expand(GameNode):void
simulate(GameNode):Color
backPropagate(GameNode,Color):void
setMoveTime(long):void
<<Java Class>>
Player
go.Model
color: Color
name: String
enemyStonesCaptured: int
isUsingAI: boolean
colorEnclosedRegionPoints: int
Player(String,Color)
enableAI():void
disableAI():void
isUsingAI():boolean
getColor():Color
incrementStonesCaptured(int):void
getNumStonesCaptured():int
resetScore():void
getName():String
setName(String):void
setCERPoints(int):void
getScore():int
<<Java Class>>
GameOverPopUp
go
dialog: Stage
scoreLabel: Label
GameOverPopUp(Stage)
display():void
hide():void
<<Java Class>>
GamePlayScreen
go
GamePlayScreen()
update():void
<<Java Class>>
HomeScreen
go
label: Label
newGameBtn: Button
exitBtn: Button
newGameOptions: VBox
HomeScreen()
-position 0..1
-root 0..1
-boardView 0..1
-sidePanel 0..1
-humanVsComputerScreen 0..1
-humanVsHumanScreen 0..1
<<Java Class>>
Pair<K,V>
go.Model.Utility
key: K
value: V
Pair(K,V)
getKey()
getValue()
equals(Object):boolean
<<Java Class>>
GameNode
go.Model
row: int
col: int
GameNode(Board,Color)
GameNode(GameNode)
GameNode(BoardState)
getBoardState():BoardState
getChildren():List<GameNode>
getParent():GameNode
setParent(GameNode):void
randomChild():GameNode
setPosition(int,int):void
getRow():int
getCol():int
getChildWithMostWins():GameNode
printChildStats():void
-children
-parent
0..1
<<Java Class>>
BoardView
go
gc: GraphicsContext
woodImg: Image
BoardView(int,int)
attemptToPlaceStone(int,int):void
update():void
drawCircle(double,double,Paint):void
pixelToBoardCoordinate(double,double):Pair<Integer,Integer>
drawGridLines(int):void
drawBackground():void
<<Java Class>>
SidePanel
go
label: Label
imageWidth: int
whiteImageView: ImageView
blackImageView: ImageView
currPlayerImage: ImageView
SidePanel()
updateLabel():void
<<Java Class>>
HumanVsComputerScreen
go
HumanVsComputerScreen()
<<Java Class>>
HumanVsHumanScreen
go
HumanVsHumanScreen()
-state 0..1
-board 0..1
<<Java Class>>
BoardState
go.Model
player: Color
numOfWins: int
numOfPlayouts: int
BoardState()
BoardState(Board,Color)
BoardState(BoardState)
getBoard():Board
getNumOfPlayouts():int
getNumOfWins():int
incrementNumOfWins():void
getNextPlayer():Color
getPlayer():Color
setPlayer(Color):void
incrementNumOfPlayouts():void
playRandomMove():void
-board
0..1
<<Java Class>>
Board
go.Model
Board(int)
Board(Board)
findEquivalent(Board):boolean
isValidMove(int,int,Color):boolean
captureStones(Color):int
countCapturedStones(Color):int
captureSingleStone(Color):Pair<Integer,Integer>
createStone(int,int,Color):Stone
placeStoneOnBoard(int,int,Color):void
removeStoneFromBoard(int,int):void
getAdjacentStonesNESW(int,int):Stone[]
getAdjacentPointsNESW(int,int):Point[]
getPoint(int,int):Point
getStone(int,int):Stone
isValidLocation(int,int):boolean
clearBoard():void
size():int
getPoints():Point[][]
validMoves(Color):List<Pair<Integer,Integer>>
scoreBoard():Pair<Integer,Integer>
calculateScore(ArrayList<Pair<List,String>>):Pair<Integer,Integer>
outcome(Color):String
-board 0..*
<<Java Class>>
Point
go.Model
Point()
getAdjacentPoints():List<Point>
getStone():Stone
setStone(Stone):void
-adjacentPoints
0..*
-stone 0..1
<<Java Class>>
Stone
go.Model
color: Color
maxLiberties: int
Stone(Color,int)
getAdjacentStones():List<Stone>
getColor():Color
getNumLiberties(Stone,Set<Stone>):int
-adjacentStones
0..*

Figure 2

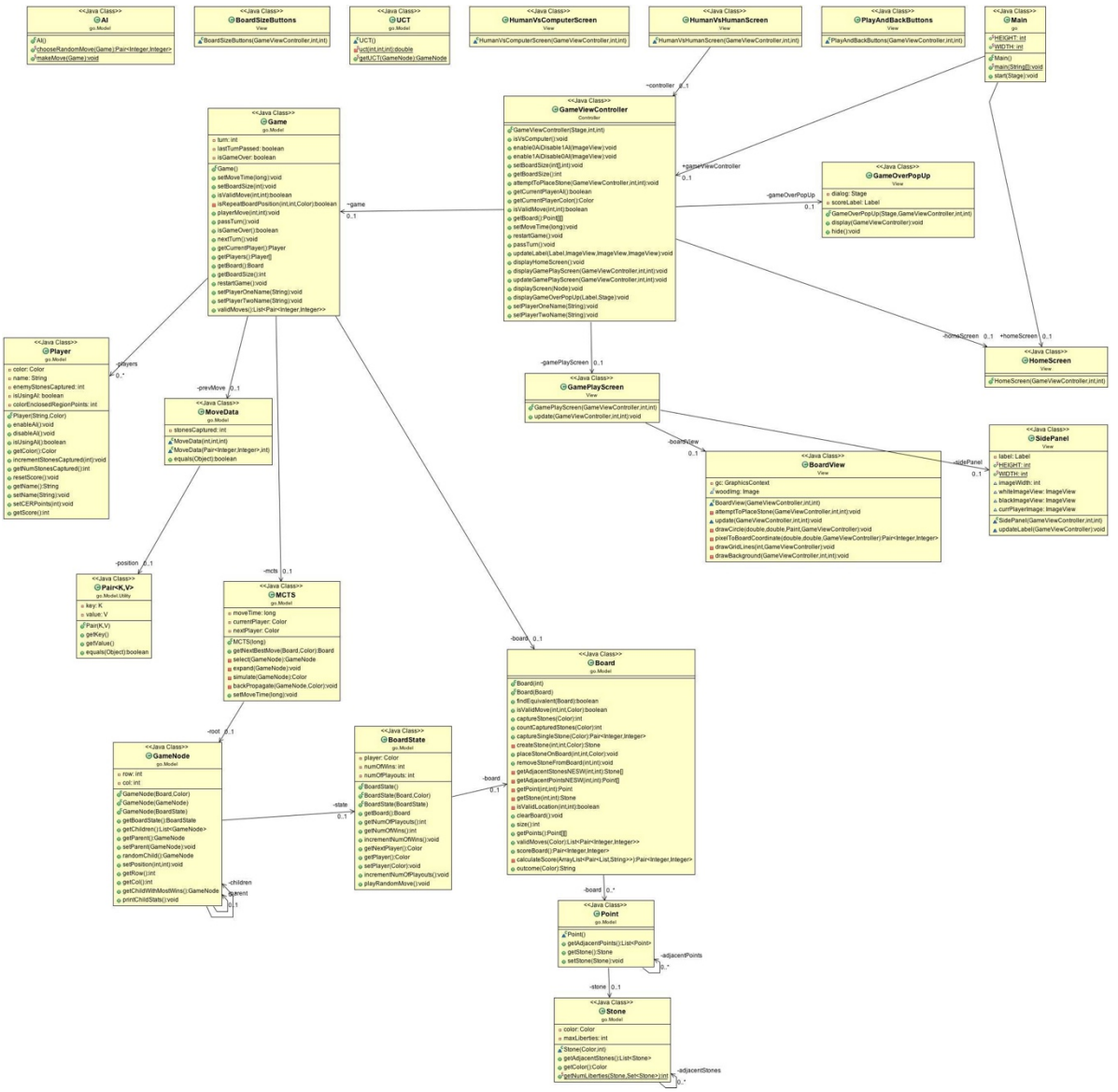

Figure 3

Evaluation

Since the change that I have made to BetaGo is an architectural change and not adding additional functionalities to the code, the players of the game will not be affected at all since the game will run in exactly the same way. However, the main stakeholder to be affected by the architectural change is the developers of BetaGo themselves. The developers should find that the code is a lot cleaner, easier to read and digest, and a lot more cohesive. The separated View classes makes the code a lot easier to read and keep track of, and makes it easy to add on more if desired. Meanwhile, the MVC makes the code easy to maintain by the developers because the project is split into distinct components that communicate with each other with a better flow, rather than the mess of mixed communication that was happening previously. The developers will therefore benefit from this new extensible system that provides them with a better foundation for building and improving on in the future.