Software Systems
Programming Project
Abalone

Bozhidar Petrov
Daan Pluister

January 2020

1 Introduction

The goal of this project is to develop a distributed client/server program for playing the board game Abalone. This report will focus on the design and implementation of the program.

2 Design

2.1 Class diagrams

The Visual Paradigm files are attached to the report if closer inspection is required. In figure 1 a overview of the classes can be found which includes classes for the server and client, a local game and for the game.

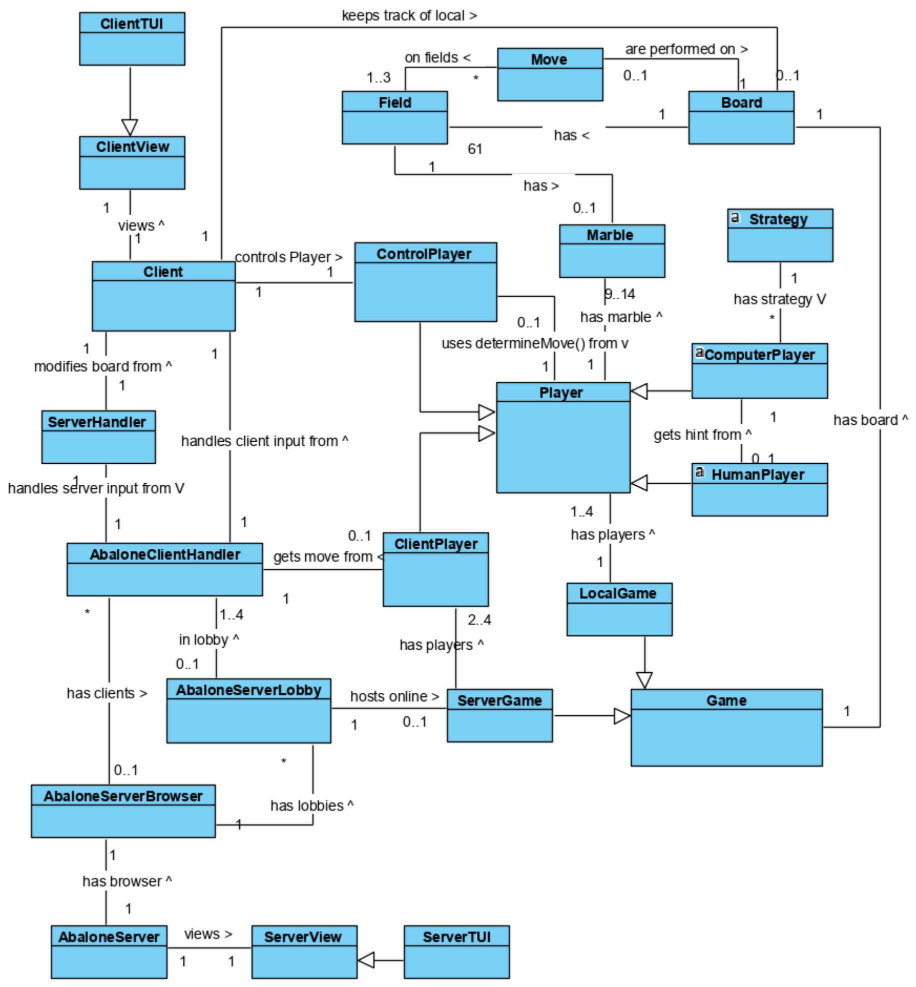


Figure 1: Overview class diagram for Abalone

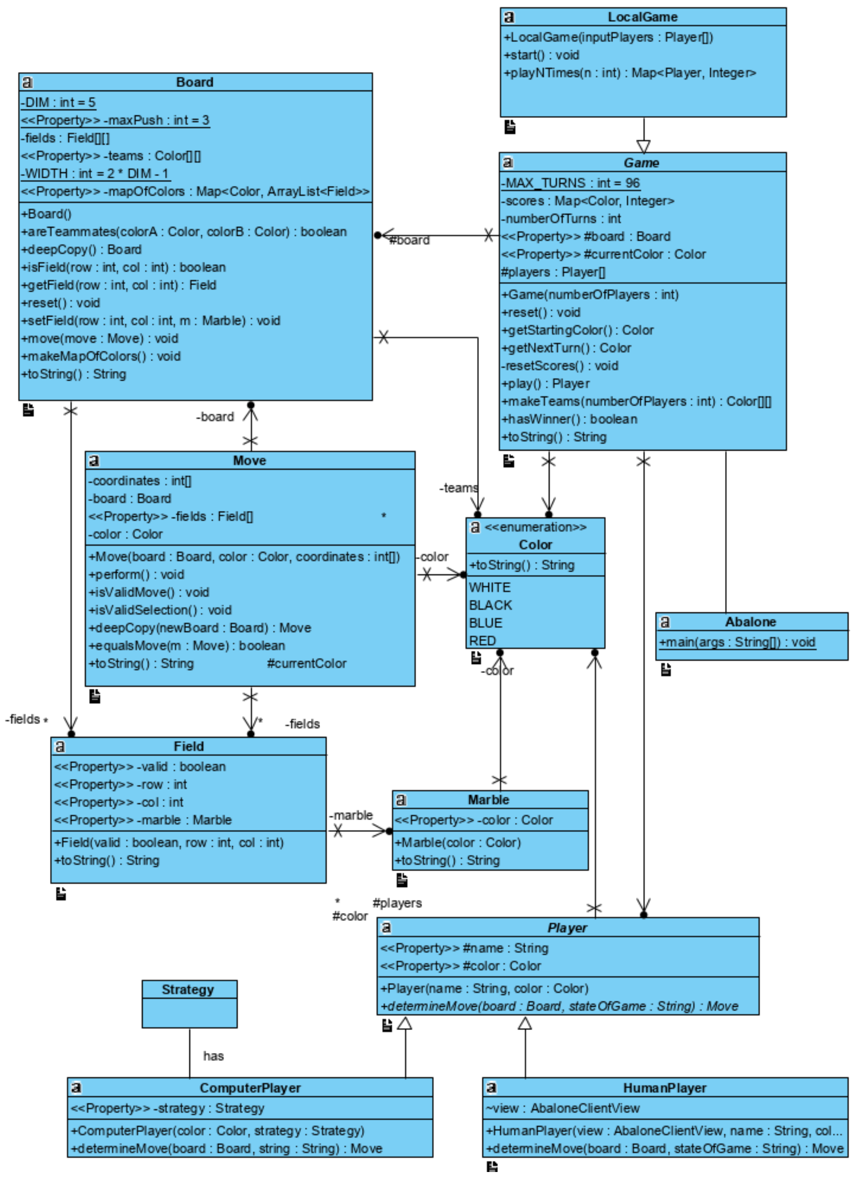

LocalGame
+LocalGame(inputPlayers : Player[])
+start() : void
+playNTimes(n : int) : Map<Player, Integer>
Board
-DIM : int = 5
<<Property>> -maxPush : int = 3
-fields : Field[][]
<<Property>> -teams : Color[][]
-WIDTH : int = 2 * DIM - 1
<<Property>> -mapOfColors : Map<Color, ArrayList<Field>>
+Board()
+areTeammates(colorA : Color, colorB : Color) : boolean
+deepCopy() : Board
+isField(row : int, col : int) : boolean
+getField(row : int, col : int) : Field
+reset() : void
+setField(row : int, col : int, m : Marble) : void
+move(move : Move) : void
+makeMapOfColors() : void
+toString() : String
#board
Game
-MAX_TURNS : int = 96
-scores : Map<Color, Integer>
-numberOfTurns : int
<<Property>> #board : Board
<<Property>> #currentColor : Color
#players : Player[]
+Game(numberOfPlayers : int)
+reset() : void
+getStartingColor() : Color
+getNextTurn() : Color
-resetScores() : void
+play() : Player
+makeTeams(numberOfPlayers : int) : Color[][]
+hasWinner() : boolean
+toString() : String
-board
Move
-coordinates : int[]
-board : Board
<<Property>> -fields : Field[]
-color : Color
+Move(board : Board, color : Color, coordinates : int[])
+perform() : void
+isValidMove() : void
+isValidSelection() : void
+deepCopy(newBoard : Board) : Move
+equalsMove(m : Move) : boolean
+toString() : String
#currentColor
-teams
-color
<<enumeration>>
Color
+toString() : String
WHITE
BLACK
BLUE
RED
-color
Abalone
+main(args : String[]) : void
-fields *
*
-fields
Field
<<Property>> -valid : boolean
<<Property>> -row : int
<<Property>> -col : int
<<Property>> -marble : Marble
+Field(valid : boolean, row : int, col : int)
+toString() : String
-marble
Marble
<<Property>> -color : Color
+Marble(color : Color)
+toString() : String
*
#color
#players
Player
<<Property>> #name : String
<<Property>> #color : Color
+Player(name : String, color : Color)
+determineMove(board : Board, stateOfGame : String) : Move
Strategy
has
ComputerPlayer
<<Property>> -strategy : Strategy
+ComputerPlayer(color : Color, strategy : Strategy)
+determineMove(board : Board, string : String) : Move
HumanPlayer
~view : AbaloneClientView
+HumanPlayer(view : AbaloneClientView, name : String, col...
+determineMove(board : Board, stateOfGame : String) : Move

Figure 2: Detailed class diagram for the game

2.2 Systematic overview

The project is divided in a few main packages:

- Abalone: the game package. This contains the base game with all of its properties.

- Server and client: with all of their properties like views and handlers
- AI: contains the interface Strategy and the different AI that implement it
- Exceptions, protocol, test

2.2.1 Board and move

The implementation of the rules of the game is mainly in the Move and Board classes. These are explained in detail in the readMe and the JavaDocs.

2.2.2 Server and client

The server and the client classes allow for online play. The communication between these is specified by a protocol which can be found in the git repository attached with this report. The server runs a game, while the clients connecting to it only have a board which they update based on what the server sends them.

2.3 Model-view-controller

This project can be described by the Model-view-controller pattern. The model is the board where the state of the game is stored. The view is the TUI in our case, which is the connection between the user and the model. The server and the client handler are the controller, since they give the commands to the model.

3 Features

This report will not go over the parts of the program which were given as requirements. However some extra features were implemented.

3.1 Scalability

The program works for different sizes of the board and can determine the starting positions of the marbles based on it. Furthermore the number of marbles that can be moved is also scalable, also the number of teams and player per team. However most of these were not properly tested. A useless feature but it is implemented nonetheless

3.2 AI

Two AI were made: one basic one that chooses a move at random, and one advanced one. The advanced one simulates all possible moves it can make and evaluates them based on 4 metrics. To determine how the metrics should be weighed in the decision, an algorithm is made that mutates AI and puts them against each other in order to find the optimal one. This could also be extended to look several moves ahead for better results. However due to lack of time this

AI was not developed extensively. Despite that it still performs significantly better than the random AI.

4 Testing

JUnit was used to make tests for the game. Other than that, a lot of things were simply checked visually, which is often easier due to the nature of the game.

4.1 Board

For the board class tests were made to check whether the conversion between the user coordinate system and the internal one works. Also a test to visually check whether the toString() method works and whether the starting distributions of marbles are correct. There is a test that checks whether the team configuration works and some other methods of the class like rotating a point around the middle of the board.

4.2 Move

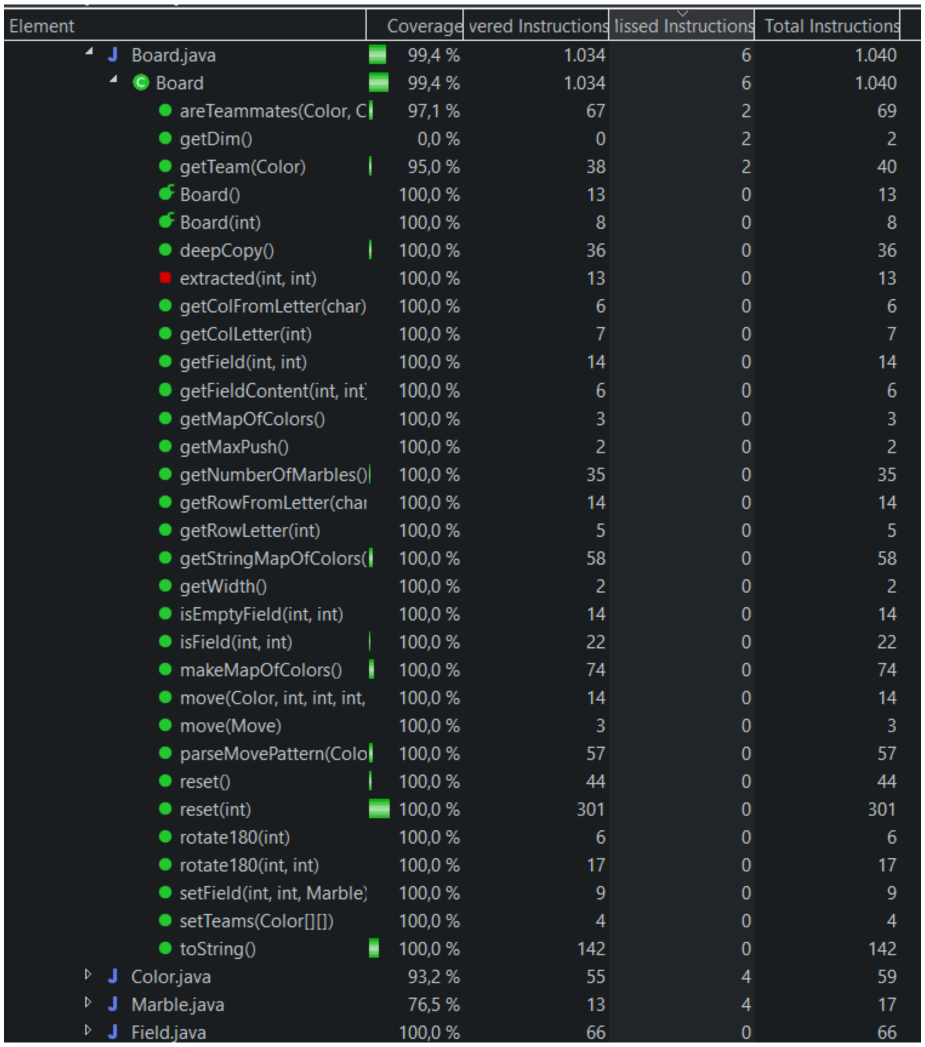
For the move class the idea is to test all different configurations for a move that can happen, including valid and invalid ones.

4.3 Strategy

Strategy has a method to determine all possible moves a given player can do, which needs to be tested. This is done by counting those moves manually and comparing their number to the result of the method. The different metrics that are used for the advanced strategy are also tested, like finding the distance of a point to the center of the board.

4.4 Coverage

4.4.1 Board test

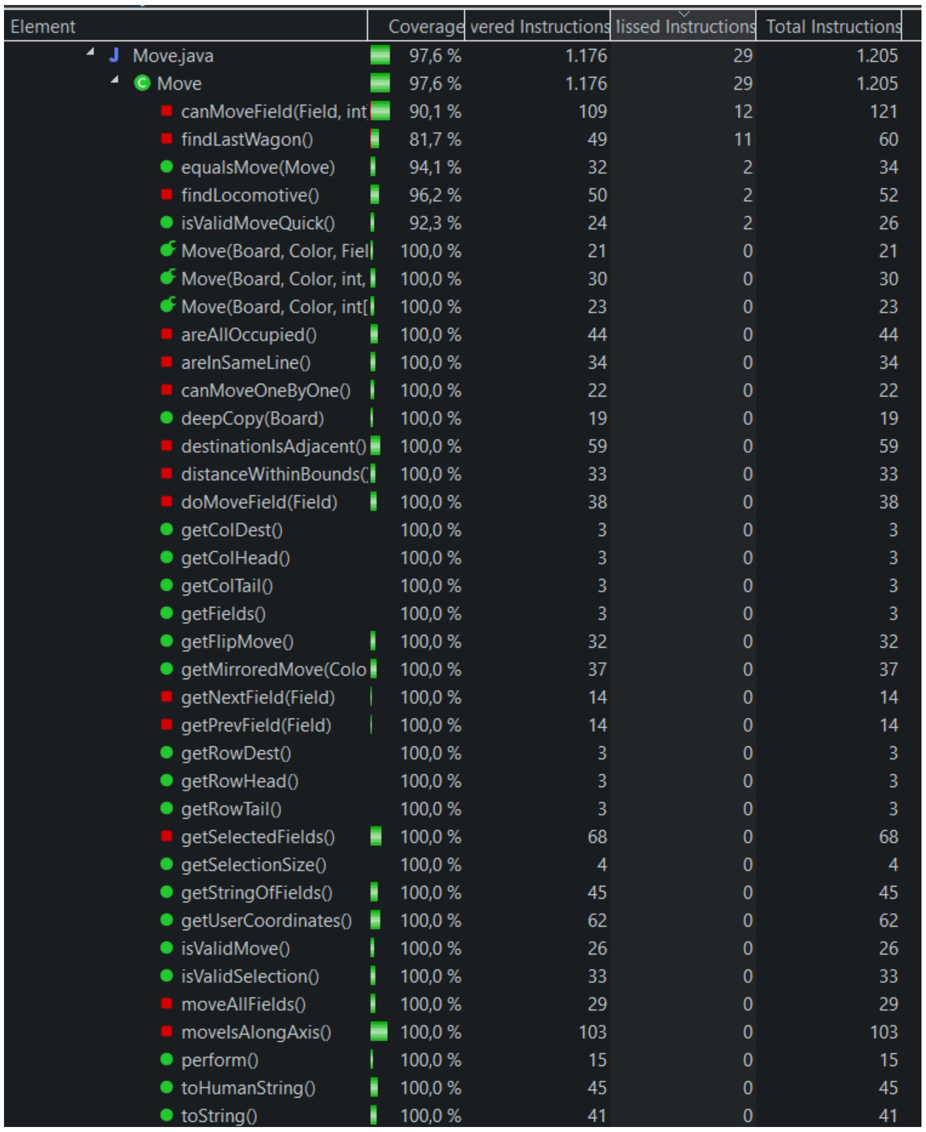


Element	Coverage	vered Instructions	lissed Instructions	Total Instructions
Board.java	99,4 %	1.034	6	1.040
Board	99,4 %	1.034	6	1.040
areTeammates(Color, C	97,1 %	67	2	69
getDim()	0,0 %	0	2	2
getTeam(Color)	95,0 %	38	2	40
Board()	100,0 %	13	0	13
Board(int)	100,0 %	8	0	8
deepCopy()	100,0 %	36	0	36
extracted(int, int)	100,0 %	13	0	13
getColFromLetter(char)	100,0 %	6	0	6
getColLetter(int)	100,0 %	7	0	7
getField(int, int)	100,0 %	14	0	14
getFieldContent(int, int	100,0 %	6	0	6
getMapOfColors()	100,0 %	3	0	3
getMaxPush()	100,0 %	2	0	2
getNumberOfMarbles()	100,0 %	35	0	35
getRowFromLetter(char	100,0 %	14	0	14
getRowLetter(int)	100,0 %	5	0	5
getStringMapOfColors(	100,0 %	58	0	58
getWidth()	100,0 %	2	0	2
isEmptyField(int, int)	100,0 %	14	0	14
isField(int, int)	100,0 %	22	0	22
makeMapOfColors()	100,0 %	74	0	74
move(Color, int, int, int,	100,0 %	14	0	14
move(Move)	100,0 %	3	0	3
parseMovePattern(Colo	100,0 %	57	0	57
reset()	100,0 %	44	0	44
reset(int)	100,0 %	301	0	301
rotate180(int)	100,0 %	6	0	6
rotate180(int, int)	100,0 %	17	0	17
setField(int, int, Marble)	100,0 %	9	0	9
setTeams(Color[][])	100,0 %	4	0	4
toString()	100,0 %	142	0	142
Color.java	93,2 %	55	4	59
Marble.java	76,5 %	13	4	17
Field.java	100,0 %	66	0	66

Figure 3: Coverage BoardTest

4.4.2 Move test

Aside from the normal tests Move test also tests whether every move defined in the rules are valid and checks if invalid moves are invalid.

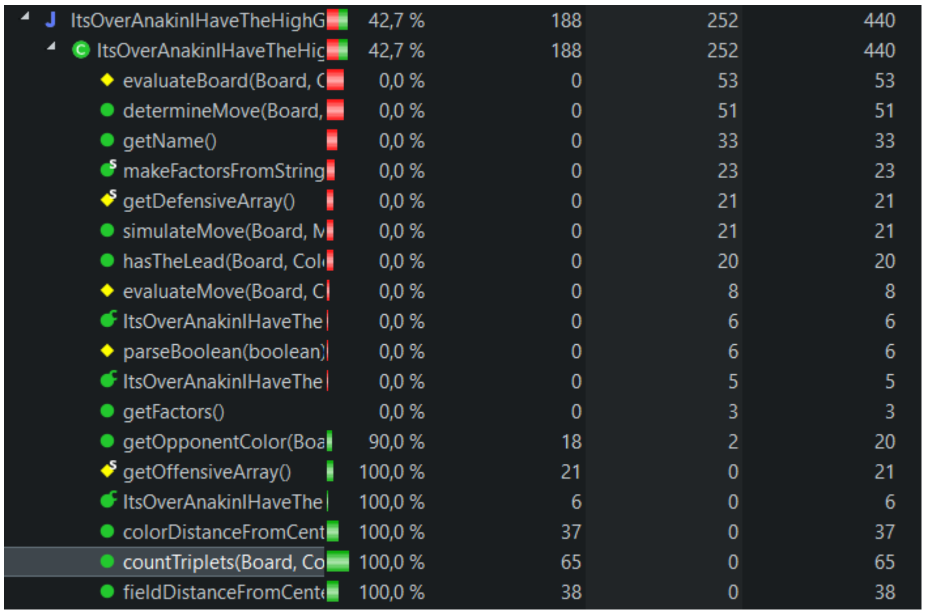


Element	Coverage	vered Instructions	lissed Instructions	Total Instructions
Move.java	97,6 %	1.176	29	1.205
Move	97,6 %	1.176	29	1.205
🟥 canMoveField(Field, int	90,1 %	109	12	121
🟥 findLastWagon()	81,7 %	49	11	60
● equalsMove(Move)	94,1 %	32	2	34
🟥 findLocomotive()	96,2 %	50	2	52
● isValidMoveQuick()	92,3 %	24	2	26
● Move(Board, Color, Fiel	100,0 %	21	0	21
● Move(Board, Color, int,	100,0 %	30	0	30
● Move(Board, Color, int[	100,0 %	23	0	23
🟥 areAllOccupied()	100,0 %	44	0	44
🟥 areInSameLine()	100,0 %	34	0	34
🟥 canMoveOneByOne()	100,0 %	22	0	22
● deepCopy(Board)	100,0 %	19	0	19
🟥 destinationIsAdjacent()	100,0 %	59	0	59
🟥 distanceWithinBounds(	100,0 %	33	0	33
🟥 doMoveField(Field)	100,0 %	38	0	38
● getColDest()	100,0 %	3	0	3
● getColHead()	100,0 %	3	0	3
● getColTail()	100,0 %	3	0	3
● getFields()	100,0 %	3	0	3
● getFlipMove()	100,0 %	32	0	32
● getMirroredMove(Colo	100,0 %	37	0	37
🟥 getNextField(Field)	100,0 %	14	0	14
🟥 getPrevField(Field)	100,0 %	14	0	14
● getRowDest()	100,0 %	3	0	3
● getRowHead()	100,0 %	3	0	3
● getRowTail()	100,0 %	3	0	3
🟥 getSelectedFields()	100,0 %	68	0	68
● getSelectionSize()	100,0 %	4	0	4
● getStringOfFields()	100,0 %	45	0	45
● getUserCoordinates()	100,0 %	62	0	62
● isValidMove()	100,0 %	26	0	26
● isValidSelection()	100,0 %	33	0	33
🟥 moveAllFields()	100,0 %	29	0	29
🟥 moveIsAlongAxis()	100,0 %	103	0	103
● perform()	100,0 %	15	0	15
● toHumanString()	100,0 %	45	0	45
● toString()	100,0 %	41	0	41

Figure 4: Coverage MoveTest

4.4.3 Strategy test

The coverage of the strategy class is only 42.7% this is because the main purpose of the test is that you want to test if making the move list contains all the valid moves. The methods used to make this list are covered 100%.

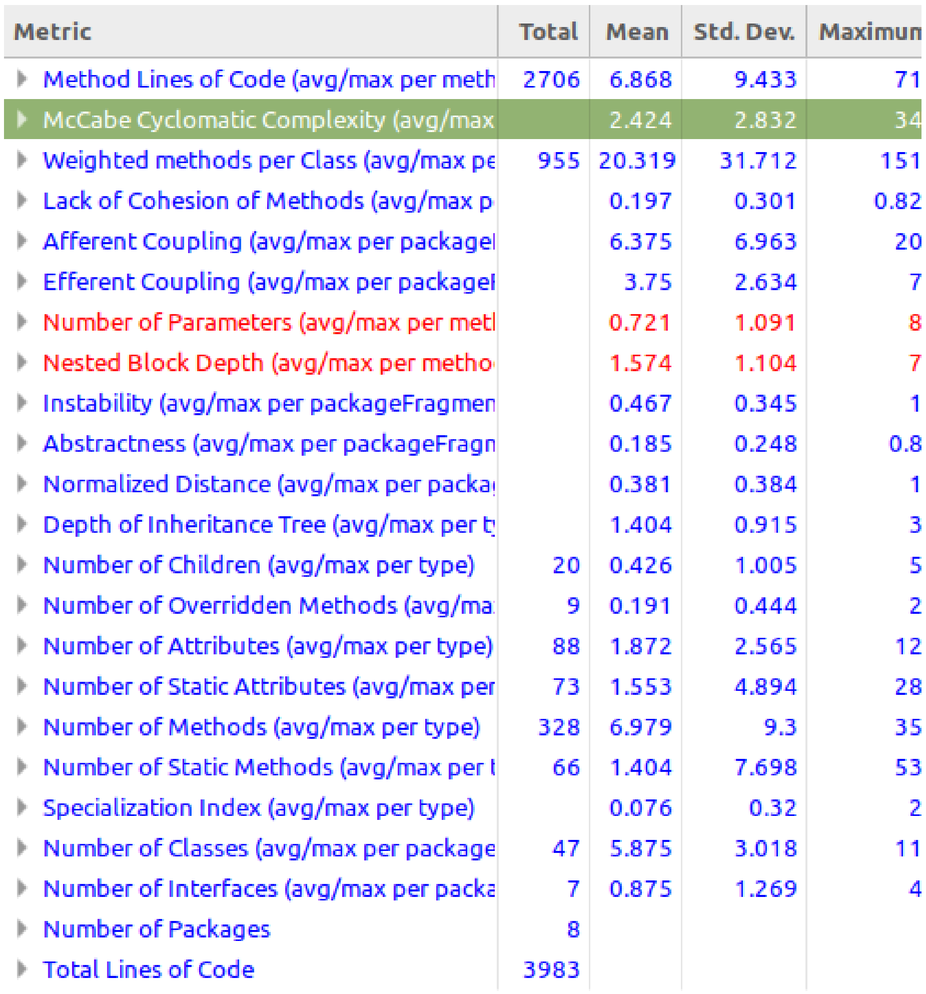


ItsOverAnakinIHaveTheHighG	42,7 %	188	252	440
ItsOverAnakinIHaveTheHig	42,7 %	188	252	440
evaluateBoard(Board, C	0,0 %	0	53	53
determineMove(Board,	0,0 %	0	51	51
getName()	0,0 %	0	33	33
makeFactorsFromString	0,0 %	0	23	23
getDefensiveArray()	0,0 %	0	21	21
simulateMove(Board, M	0,0 %	0	21	21
hasTheLead(Board, Col	0,0 %	0	20	20
evaluateMove(Board, C	0,0 %	0	8	8
ItsOverAnakinIHaveThe	0,0 %	0	6	6
parseBoolean(boolean)	0,0 %	0	6	6
ItsOverAnakinIHaveThe	0,0 %	0	5	5
getFactors()	0,0 %	0	3	3
getOpponentColor(Boa	90,0 %	18	2	20
getOffensiveArray()	100,0 %	21	0	21
ItsOverAnakinIHaveThe	100,0 %	6	0	6
colorDistanceFromCent	100,0 %	37	0	37
countTriplets(Board, Co	100,0 %	65	0	65
fieldDistanceFromCent	100,0 %	38	0	38

Figure 5: Coverage StrategyTest

5 Metrics

The following picture shows the summary of the metrics for this project. Some of them will be discussed in some detail.



Metric	Total	Mean	Std. Dev.	Maximun
Method Lines of Code (avg/max per meth	2706	6.868	9.433	71
McCabe Cyclomatic Complexity (avg/max		2.424	2.832	34
Weighted methods per Class (avg/max pe	955	20.319	31.712	151
Lack of Cohesion of Methods (avg/max p		0.197	0.301	0.82
Afferent Coupling (avg/max per packagel		6.375	6.963	20
Efferent Coupling (avg/max per packagel		3.75	2.634	7
Number of Parameters (avg/max per metl		0.721	1.091	8
Nested Block Depth (avg/max per metho		1.574	1.104	7
Instability (avg/max per packageFragmen		0.467	0.345	1
Abstractness (avg/max per packageFragn		0.185	0.248	0.8
Normalized Distance (avg/max per packa		0.381	0.384	1
Depth of Inheritance Tree (avg/max per t		1.404	0.915	3
Number of Children (avg/max per type)	20	0.426	1.005	5
Number of Overridden Methods (avg/ma	9	0.191	0.444	2
Number of Attributes (avg/max per type)	88	1.872	2.565	12
Number of Static Attributes (avg/max per	73	1.553	4.894	28
Number of Methods (avg/max per type)	328	6.979	9.3	35
Number of Static Methods (avg/max per t	66	1.404	7.698	53
Specialization Index (avg/max per type)		0.076	0.32	2
Number of Classes (avg/max per package	47	5.875	3.018	11
Number of Interfaces (avg/max per packa	7	0.875	1.269	4
Number of Packages	8			
Total Lines of Code	3983			

Figure 6: Summary of metrics from Eclipse

5.1 Lines of code per method (LOC)

For this project the average lines of code per method is less than 7. The guideline for this metric is 15 or less, which makes our implementation good. A large number of LOC would result in the code being more difficult to comprehend by others and thus more difficult to maintain.

5.2 Cyclomatic complexity (CC)

The cyclomatic complexity is around 2.4, whereas the guideline is 5 or lower. A low cyclomatic complexity makes the code easier to understand and modify.

6 Reflection on planning

Our planning was heavily based on the Design part of this module. The first thing we did was make a class diagram, which of course changed a lot over the weeks, but always served as a good guideline.
The outline of our planning was followed: start by making the game and its mechanics, in order to make it playable locally. Then focus on making the server and client, and making an AI.
However in our planning we underestimated how much time would be spent making the game itself, specifically the Move class, which resulted in us having less time to work on the server/client implementation.
Another setback was when we implemented the client incorrectly, at least in a very inefficient way which proved to be too difficult to make, so we had to start over from scratch only a few days before the deadline.
Furthermore some details about the mechanics of the game, which were in our planning, were changed to match the requirements of the protocol.
There are a couple of dos and don'ts we would recommend for this project.

- Do extensively test the "Move" class. This class ended up creating a lot of problems because there were some edge cases which were not considered initially. In general, do a lot of tests.
- Don't focus on things that are not crucial, like scalability or a good AI, before the main requirements of the project are finished.
- Don't neglect the protocol because even a small problem with it can make the game unplayable through a server.
- Do start on time with the implementation of the server since it can take longer than expected.