Spectrangle Project Report

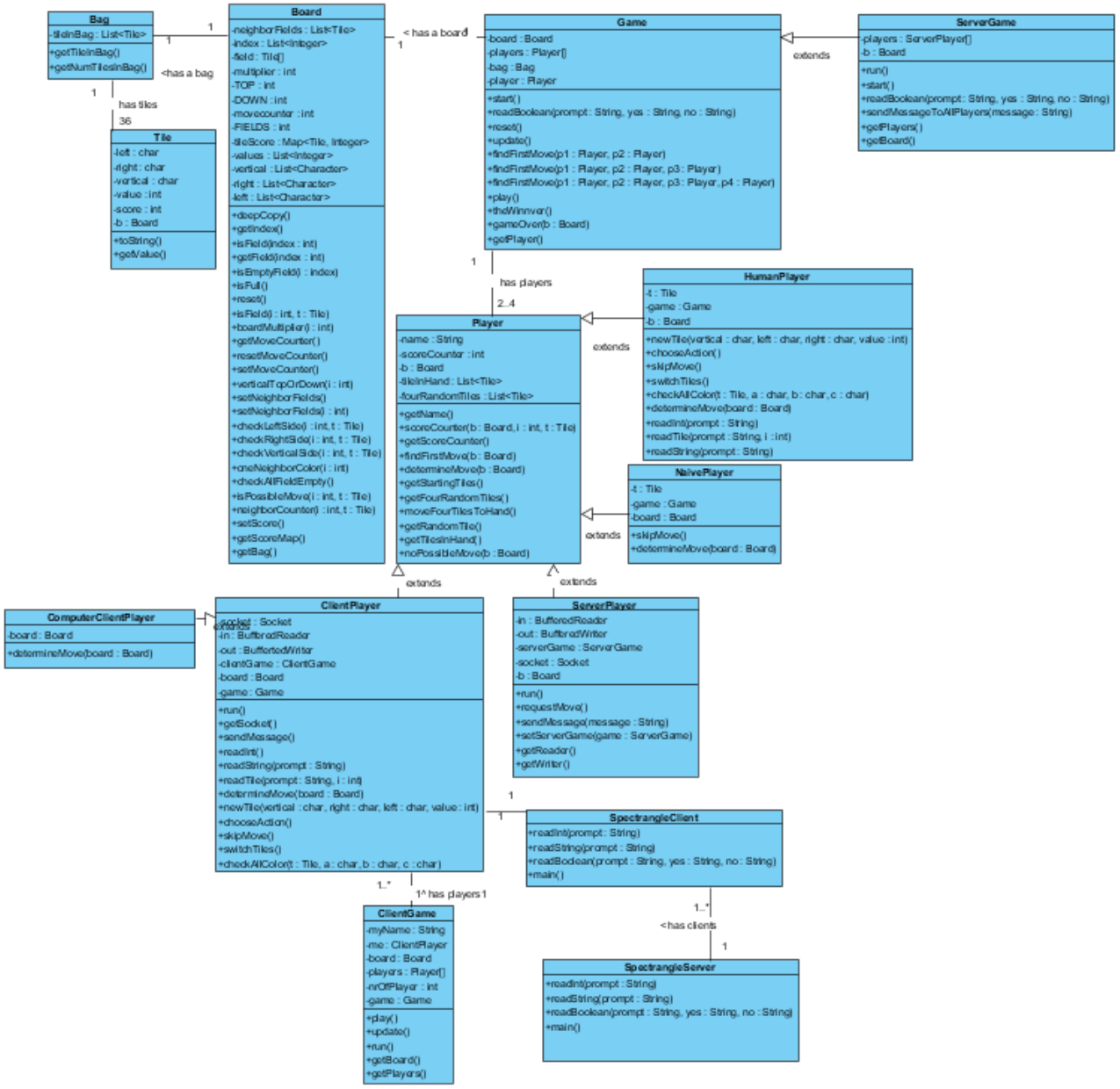
CS 4 - 4

Christophorus Jeremy Wicaksana s2096862
Melvin Sumon Chandra s2132257

Contents:

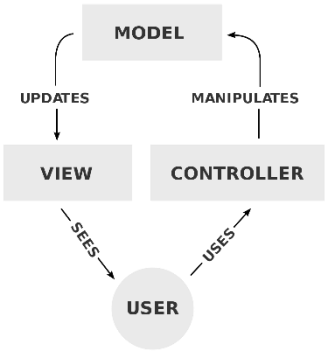
Discussion of the Overall Design

Class Diagram



Pictured above is the class diagram for the Spectrangle game package.

MVC (Model-View-Controller)



Based on our classes for the Spectrangle game, it can be seen that the classes could be split into 3 sub-groups which are Model, View, and Controller.

The Model of our Spectrangle is the Game class since this class can update the current board condition, also it can be controlled by players to manipulate the outcome of the game then updates it into the view. The Game class is considered as the model of the game as it determines the rules and regulations that the player needs to obey in order to play the game normally.

While the Controller is the Player class which can be used as a computer player of human player. The players of the game controls how the game is being played out and makes use of the rules given based on the model, which is the Game class. The behavior of human player and computer player is different because the computer player plays the game according to the given code, while human player is played by a person so the result could vary.

Lastly, the View is the Board class. With the help of SpectrangleBoardPrinter class, the Board could generate the board string with definite rules created within the board class. Each turn, the model (Game class) updates the view (Board class) based on the progress made by the controller (Player class). If the input from the players in the game does not fulfill the rules, then the board will not be updated. The board acts as a platform for the game, hence it is the view.

Discussion per Class

The Game package is divided into 9 classes, including the Spectrangle Board Printer. Each class will be briefly explained in this part. The classes are namely:

Tile

- The Tile class declares the 6 colors that are available in the 36 tiles that will be used in the game. The colors are of type ‘char’ and is assigned to the ‘left’, ‘right’ or ‘vertical’ part of the Tile (of type ‘char’ as well). Besides the position of the colors, the Tile also has its own point/value for the game. This is declared of type ‘int’. The original orders for the Tile that we used is in the form of ‘vertical’, ‘right’, and ‘left’. Based on the protocol, this means that the original Tile is facing downwards.

Bag

- The Bag class declares all the 36 possible tiles in the original game and is compiled in the form of an ‘ArrayList’. In short, this means that all the 36 tiles are inside the bag. In this class, the methods getTileInBag and getNumTilesInBag are specified which will retrieve one tile from the bag and return the number of remaining tiles in the bag respectively. This method will be called in the other classes.

Board

- The Board class is all about assigning the fields and the tiles. The components of the board, finding neighboring fields for the tiles, and also matching the tiles with its corresponding neighbors are all part of the Board class, hence it’s one of the most important aspects of the game. The constructor of the class sets the fields in the board as indexes which are divided into 36. It also uses the help of the Tile class. The methods in this class are:
    - deepCopy(): creates a deep copy of the fields.
    - getIndex(): returns a list of the indexes in the board.
    - isField(): checks if the index is part of the field on the board.

- getField(): returns the field of a certain index, uses the Tile class as a type.
- isEmptyField(): checks if the field in the board is empty.
- isFull(): checks if the field in the board is occupied.
- reset(): clears the board and resets the move counter.
- setField(): places a tile on the field of a chosen index.
- boardMultiplier(): sets the multipliers in some of the fields in the board.
- getMoveCounter(): returns the number of moves a player has done.
- resetMoveCounter(): sets the move counter to 0.
- setMoveCounter(): increments the move counter after a player's turn.
- verticalTopOrDown(): checks if the field is facing up or down.
- setNeighborFields(): sets the neighboring fields of each field on the board.
- checkLeftSide(): checks if the left side of the tile has the same color as the right side of the left neighboring tile.
- checkRightSide(): checks if the right side of the tile has the same color as the left side of the right neighboring tile.
- checkVerticalSide(): checks if the vertical side of the tile has the same color as the vertical side of the vertical neighboring tile.
- checkAllFieldEmpty(): checks if all the fields on the board are empty.
- isPossibleMove(): checks if a chosen move by a player is possible.
- setScore(): sets the score for each tile in the bag.
- getScoreMap(): returns the score of a tile.

Game

- The class Game represents the components of the rules of the game itself. It uses the help of other classes such as Board, Player and Bag (by first calling it in the constructor) and includes the methods:
  - start(): starts a new Spectrangle game
  - reset(): resets and clears the Board. The Board class is being called in this method.
  - findFirstMove(): Finding the first player to start the turn. In the Player class this method retrieves a random tile from the bag for every player

and in the Game class the player with the highest tile value starts first. The Player class is being called in this method.

- play(): plays the Spectrangle game according to the rules (based on the methods on the class Player/HumanPlayer such as determineMove and moveFourTilesToHand, also the method getMoveCounter from the class Board). This method includes taking four tiles from the bag for each player and placing the tiles on the board, basically playing the game in general.
- update(): prints the current state of the board after every turn. The Board class is being called in this method.
- gameOver(): ends the game when there are no more tiles inside the bag and/or the players have no more possible move. The classes Board, Player and Bag are being called in this method.
- theWinner(): counts the total score of each player and announce the player with the highest score as the winner of the game. The class Player is being called in this method.

Player

- In the Player class, the actions that can be done by the players are specified. In this case it's mostly about the moves and the tiles. The methods are general actions that can be done by all players, and specific moves for HumanPlayer and NaiveComputer will be specified in their respective classes. The class Board, Bag and Tile is called in the Player class. The Tile class is being declared of type ArrayList. It includes the methods:
    - getName(): returns the name of the player
    - scoreCounter(): counts the current score of each player after every turn. The class Board and Tile is used to help determine the score achieved by players after every turn (since there are multipliers on the board and each tile have different values).
    - getScoreCounter(): returns the scoreCounter method.
    - findFirstMove(): retrieves the value of a random tile from the bag for every player. The Tile class is called in this method to get the value of the tile for each player which will then be compared in the Game class to determine which player starts first.

- getStaringTiles(): gets four random tiles from the bag to be placed in each player's hand. The class Bag is used in this method to retrieve the Tile.
- getFourRandomTiles(): returns the four random tiles that was took.
- moveFourTilesToHand(): adds the four random tiles to the player's hand (tileInHand).
- getRandomTileForFirstMove(): retrieves a random tile from the bag for every player.
- getRandomTile(): gets a new random tile from the bag after placing one tile to the board. The class Bag and Tile is used in this method.
- getTilesInHand(): returns the tiles in a player's hand.
- noPossibleMove(): checks whether a player can place one of the tiles in their hand into an empty field in the board. The class Board and Tile is used in this method.

HumanPlayer

- The HumanPlayer class extends Player class and has a more specific actions that only human players can do. It uses the help of classes Tile, Game, Bag, and Board in order to function properly. Also since it extends Player, it refers to the Player class as a super type which calls for 'name' in the constructor. Other methods include:
    - chooseAction(): If a player has no possible moves, he/she can choose to either switch tiles or skip his/her turn.
    - skipMove(): end the player's turn without doing any action.
    - switchTiles(): If the tiles inside the bag are still available, a player can choose to switch one of their tiles in their hand with a random tile from the bag. If there are no more tiles in the bag, player skips his/her turn. The classes Bag and Tile is used in this method.
    - checkAllColor(): checks if the Tiles have the correct color arrangement. The class Tile is being called in this method.
    - determineMove(): the process of placing a tile on the board. Players can pick a tile in their hand to be placed in the board index of their choice. It checks if the player's choice is a valid move and gives error messages if a move is invalid. The classes Board, Tile, and also Player is being used in this method.

NaiveComputer

- The NaiveComputer class is also an extension of the Player class, where it has the role of a computer player (bot/AI). However, this bot is classified as a naive bot as it randomizes each decision it makes. Also since it extends Player, it refers to the Player class as a super type which calls for ‘name’ in the constructor. The methods in this class are:
    - skipMove(): skips turn of a player without placing a tile.
    - determineMove(): determines move for the bot. It checks for the tiles in its hand in order and checks if it is possible to place that tile in one of the indexes. It places the tile on the first index it checks that can be placed.

Spectrangle

- This class only has one main method and it is used to determine the number of players and the nickname for each player and when this class is run, the game starts.

SpectrangleBoardPrinter

- Provided by the TA’s, this class is being called in some classes to print the state of the board during the game.

For the Networking package, there are 7 classes in total, in which the contents of the classes are similar to the game classes, except for SpectrangleClient and SpectrangleServer, where in these classes the client and the server are initialized. The other classes include ServerPlayer, ServerGame, ClientPlayer, ClientGame, and ComputerClientPlayer. These classes are used to set up playing the game online, with other people from different laptops. The classes Game and Player are used because it will be used for the commands based on the protocol given.

Test Report

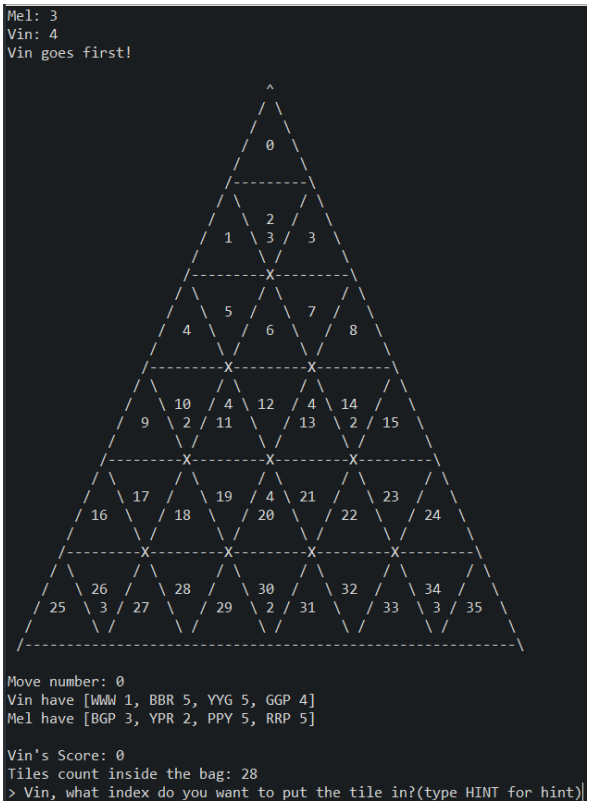
We have done two types of testing for the game, namely the unit testing and also the system testing. For the unit testing, we used JUnit for the three most complex classes; Board, Game, and Player. These three classes are unit tested in isolation by creating a JUnit test class for each of them, which can be found in the 'tests' package of the project.

Board:
The BoardTest has a 11/11 test runs with 0 errors and failures, having covered 62.9% of the Board class. This is of course expected, as more than half of the methods in the class can be tested, where some of the methods were just not possible to be tested using JUnit.

Player:
The PlayerTest has a 6/6 test runs with 0 errors and failures, having covered 56.7% of the Player class. Again, this is because more than half of the methods in the class can be tested, where some methods are just not possible to be tested.

Game:
The GameTest initially has a 4/4 test runs with 2 errors and 0 failures, only covering 2.8% of the Game class. The low coverage is due to the fact that there were many different methods made for different number of players and we were not able to cover every single method for this class. Again, some methods were not possible to be tested as well. However, there were some errors that we encountered, and the TA didn't understand what the problem was as well. The resetTest on itself does run normally and does not give any errors. The problem starts when testPlay is included in the test, giving an IndexOutOfBoundsException for the resetTest, which was odd. Another problem that we were not able to solve is the testGameOver, which gives a failure and an AssertionError on its own, but when it is tested with the other methods, it gives an error and an IndexOutOfBoundsException, similar to the resetTest. Hence, we decided to command out the resetTest and testGameOver, giving a successful 2/2 runs with 0 errors and failure and also the same coverage.

For the system testing, we run the game and just test out different implementations to see if it works and just play the game normally. In the picture below, it shows that each functionality of the game works perfectly, such as players picking a random tile and compare the highest value to determine which player starts first, and also showing the tiles in each player's hand, the move counter, and the remaining tiles in the bag.



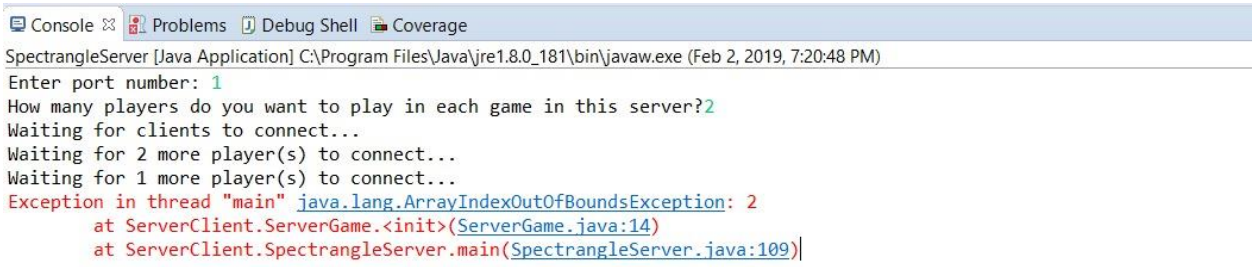
In the picture above, the noPossibleMove implementation works as ‘Mel’ does not have any tile to place, hence having to either switch tiles or skip the move.

In the picture on the left, ‘Mel’ was trying to put a tile on a field that is not empty, and an error message is received. However, this time the expected result is different from the actual result. It was supposed to print a message that says the field is not empty, but instead it prints a message that that field is not a valid choice.

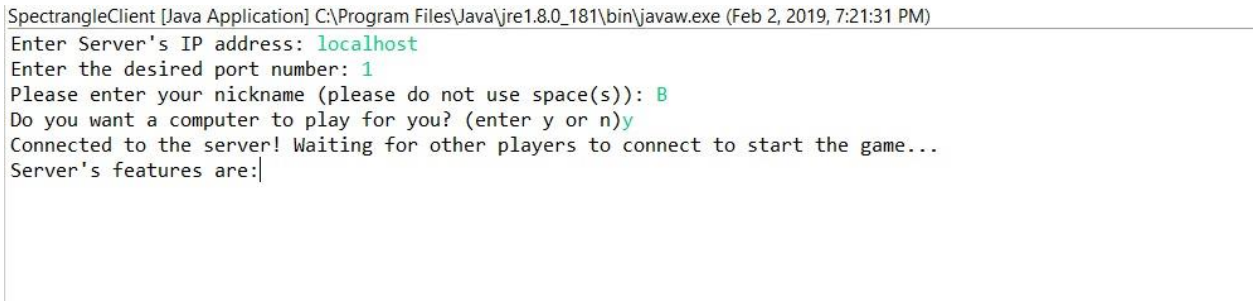
So far, the functionalities of the game works as intended. The Board class has successfully determined the colors of the neighboring tiles, the Game Class printing almost each message correctly, etc. The only problem we encountered when running the game was that after the game is finished, we implement a code that enables players to play the game again, which works the first time. However, after finishing the second game, sometimes the ‘play again?’ message does not read the input and the players are not able to play the game for the third time. Also, we were unable to implement the rotation method, and hence in our game a player will input the order of the tiles manually.

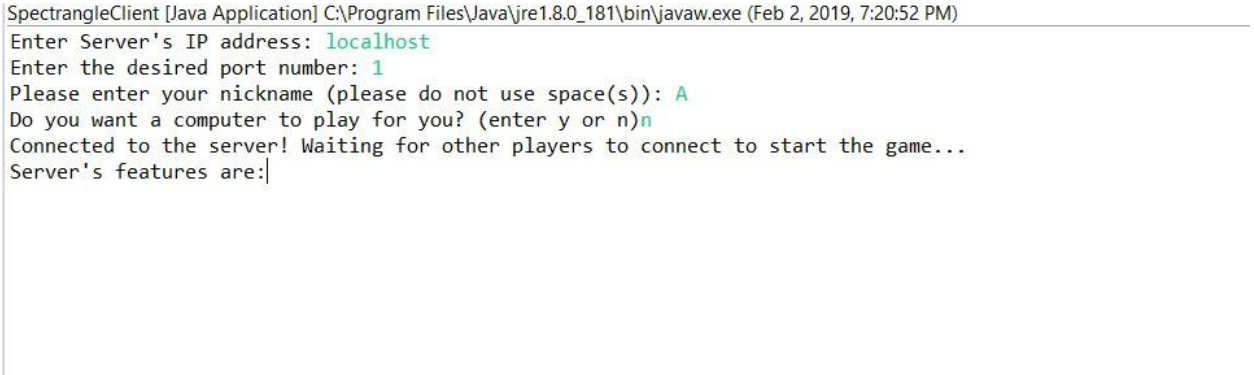
Unrelated to the tests, but we have decided to also document the classes Board, Game, and Player with JML and Javadoc. We again chose these 3 classes as they are the most interesting classes and could be considered as the ‘backbone’ of the game. They have the most complex methods that makes the game working as is. In addition, we were unable to remove some Checkstyles on some classes because of a ‘hidden field’ which we could not resolve after looking for solutions on the internet. However, the rest of the Checkstyle has been removed.

For the networking classes, unfortunately even though the codes for the server, client, and the commands based on the protocol given (CS4 https://git.snt.utwente.nl/snippets/23) are complete, we were unable to connect both our server and client to others. We were not able to solve the problem and hence we submitted what we had. Since our server was not able to connect with the other clients, and our client was not able to connect with the other servers, unfortunately we were not able to test the networking implementation. However, we tested and took pictures of the errors we received when trying to connect to others.



In the picture above, the error message appears when clients have connected to the server, unable to play the game online. Below is another picture of the clients connecting to servers without anything happening. The console shows that our client is connected to the server, however nothing happens after and we were not able to play the game online.





Metrics Report

Problems | Javadoc | Declaration | Console | Debug | Metrics - game - Method Lines of Code (avg/max per method) | Coverage

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Method Lines of Code (avg/max per	1194	15.92	32.603	180	/Spectrangle/src/game/Game.java	findFirstMove
McCabe Cyclomatic Complexity (avg		6.52	17.689	146	/Spectrangle/src/game/Game.java	findFirstMove
Weighted methods per Class (avg/ma	489	54.333	81.249	257	/Spectrangle/src/game/Game.java	
Lack of Cohesion of Methods (avg/m		0.436	0.392	0.86	/Spectrangle/src/game/Player.java	
Afferent Coupling	8					
Efferent Coupling	0					

The principles of MLOC states that larger methods are more difficult to comprehend and maintain. Generally it is better to have less than or equal to 15 for MLOC. In the whole package for game, the mean MLOC is 15.92, which is not far from the guideline for Java. The classes with high MLOC are from Game (42.083) and HumanPlayer (29.7). In the Game class, this high result is caused by having different methods for different number of players playing the game (more player, more MLOC). In the HumanPlayer class, the actions that can be done by a player is programmed in the determineMove method, which explains the high value for the lines of code for that method.

For Cyclomatic Complexity, it is recommended to keep CC low as it keeps the maintainability of the codes (easier to understand, thus easier to modify). The mean CC for the game package is 6.52, which is a little bit high. Again, this is caused by the Game class with a CC of 21.417. The methods inside the Game class can be considered complex, having different methods for different number of players, and hence the high value for CC.

Weighted Methods per Class is the sum of cyclomatic complexities of all methods in a class. On average, the WMC for the package is 54.333. The two classes who had a high value for WMC are Board (124), and of course Game (257). WMC is the number of methods in a class weighted by their complexity. In both classes, high number of methods mean high complexity.

High Lack of Cohesion Methods indicates that different methods access the same class variables while low LCOM indicates that different class variables each have their own set of methods that access them. The average LCOM for the game package is medium (0.436), with 0 being the lowest and 1 being the highest. This

indicates that some classes (Board, Game, HumanPlayer, Player, Tile) have high LCOM while the rest have 0 LCOM, which is very low.

Last but not the least is the Afferent and Efferent Coupling. High AC of a class means that it is hard to know all the effects in other classes of a change in that class. High EC of a class means that there is a potential lack of stability, as changes in classes on which that class depends may cause errors. In this case, the game package has a high AC and low EC, which is actually true. Using the Instability Formula ($I = \frac{EC}{AC+EC}$), and with a high AC and low EC, this means that I is low and the classes in the game package are stable.

Reflection on Planning

During the Design Project, our group's planning was somehow pretty well coordinated even though we had some unexpected workload on the last few days before the deadline (some diagrams needed to be fixed, missing details, etc.). For this programming project, we basically implemented our planning for the design project with some small changes, because before we thought that if it worked for the design project, it should work with other projects. The actual planning was quite simple, we just needed to finish a certain number of tasks each week so that the workload won't pile up on the last minute. For example, one week we will start with the coding of the game implementation, the next week we started with the communication protocol, then the reports, etc.

One problem we encountered during the project is that we did not count the difficulty of the tasks into account. We somehow agreed to work on the game on our own and then compare the results, which turns out to be an inefficient idea. There was some amount of time lost because we did not divide our tasks equally and we kept referring back our programming project planning to our design project planning. Then I realized that the programming project was a 2-person project, not a 4-person project, which means there will be more work that needs to be done for each person.

We also did not take the time it takes to finish certain tasks into account. Making activity diagrams are way simpler than coding the Board class. We took a significant amount of time to code for certain classes which had effects to our planning. Once we realized this, we decided to divide our tasks in order to finish the project on time. The result of the project might not be according to plan, but we are trying our best to compensate the deviation of the original planning. We are trying hard to keep the quality of the project to be exactly the same or even better than intended with the remaining time available.

What we learned from this project is not to underestimate things and take it lightly, and to maximize the time that is given to you. We will try to make a more thorough planning, taking all the possibilities into account and sticking to the plan

as well as possible without distractions. At the end of every week we will make sure that the tasks for that week are done before starting a new task. If necessary, taking your free time to do extra work for the project might be helpful too. Also, if we were to face some difficulties with certain tasks, we need to try to get as much help as possible, because sitting there staring at the computer screen for hours doesn't help a lot.

If I'm going to be a teaching assistant for this project next year, I would give them a few advices before starting the project. Some of the do's include communicating with their partners and dividing each task equally to each other, saving time and energy. Also, following the guidelines given in the module guide is a good start since you can tell if you're on track with the expected pace for the project. Asking the TA's if you have doubts or question is also important so that you won't miss any important information. For the don'ts, we do not recommend working last minute because this project requires a lot of things and you won't have time to cramp in everything in one night and submitting a decent project. Not only that, do not, under any circumstances, try and plagiarize other's work. It might seem tempting to copy someone else's work during the last minute when the deadline is approaching, but it's better to hand in your own unfinished work rather than a plagiarized project. You've tried your best, and the consequences are not worth the risk.

In addition, one of the problems that we encountered with this project is creating the networking parts, such as the server and clients. We had trouble understanding the task and we felt that the exercises from Week 7 did not help us do the networking task properly. Personally, the networking has distorted our initial plans and we were forced to do extra work during the weekends to try and solve the problem.