

CPS2002 – Assignment Report

B. Sc. (Hons) Computing Science



**Contents**

[Github and Jenkins Links 1](#_Toc514501384)

[CPS2002 Assignment – Part 2 1](#_Toc514501385)

[UML Diagram 1](#_Toc514501386)

[Implementation Design 2](#_Toc514501387)

[Testing and Coverage. 6](#_Toc514501388)

[User Manual 7](#_Toc514501389)

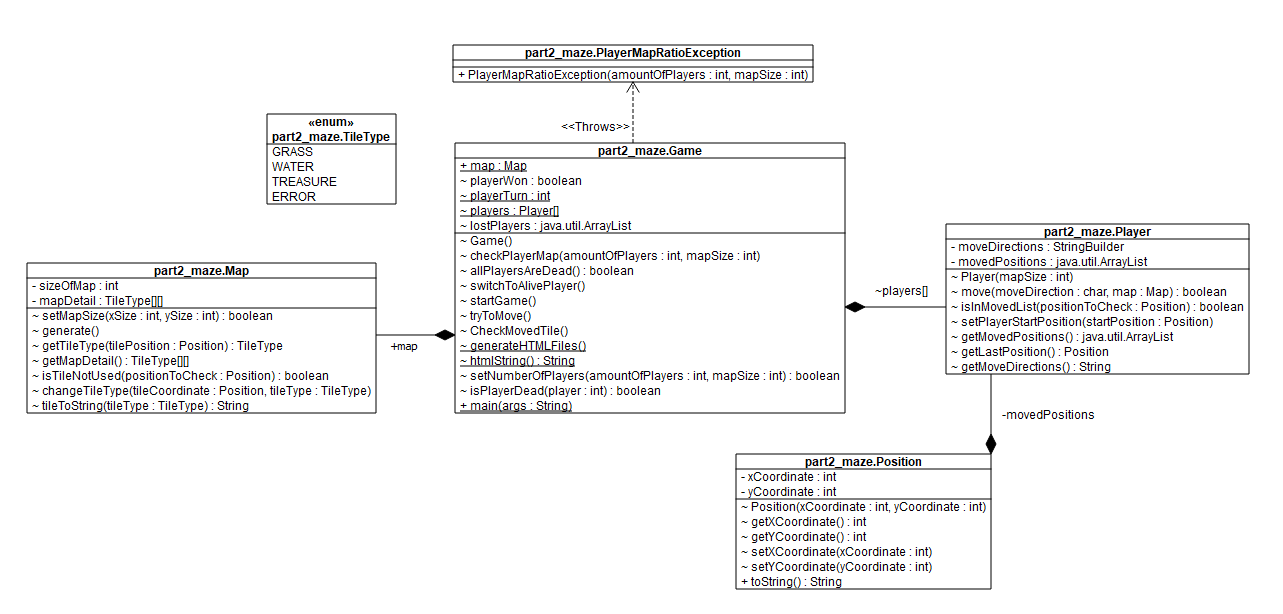
# Github and Jenkins Links

|  |  |
| --- | --- |
| Assignment Material | Link |
| GitHub | <https://github.com/brandonabela/CPS2002-Software-Engineering-Assignment> |
| Jenkins | <https://jenkins-ict.research.um.edu.mt/job/CPS2002-Software-Engineering-Assignment%20BA_DC/> |

# CPS2002 Assignment – Part 2

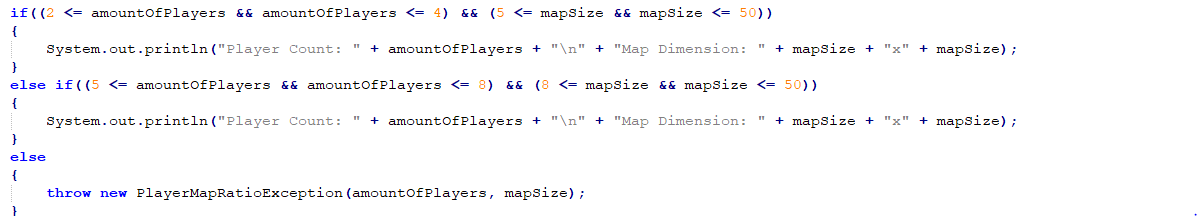
## UML Diagram

The below UML diagram shows all the classes used and the dependencies between each class. Till this part in the assignment, there was not any sort of inheritance needed. Although, it can be easily seen that the majority of the data is being stored in the class named **Game**.

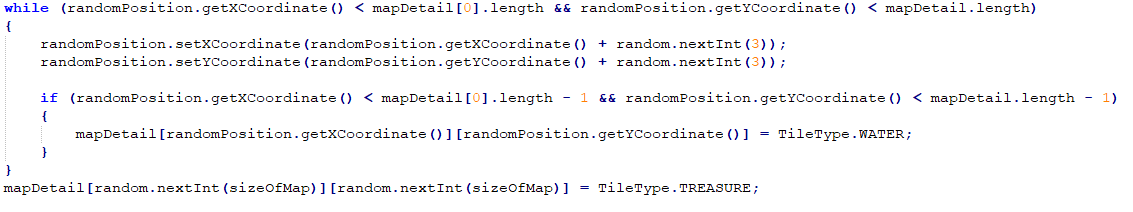


## Implementation Design

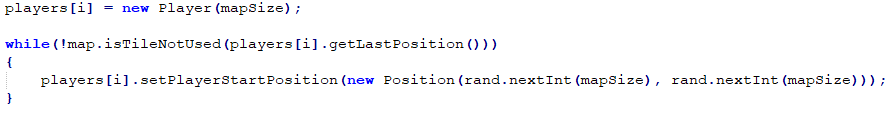
The **Game** class constructor first starts by asking the user to input the number of players and the size of the map. To verify that the user input was correct a method called **checkPlayerMap** was implemented to determine if it is a correct ratio between the number of players and the map size as seen below.



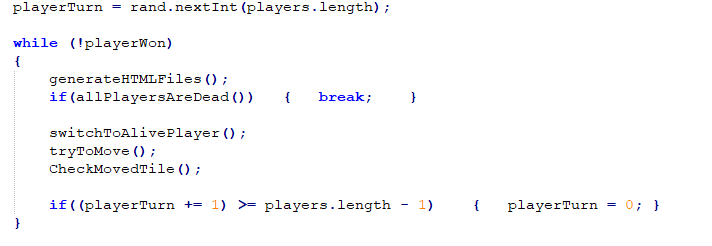
Based on the map to player ratio, a map was generated by using a method within the **Map** class called **generate**. The backend for the map generation is a two-dimensional array of type **TileType** which is an enumerator that contains all possible type of tiles. The generated map will mainly consist of green tiles, some water tiles, and 1 treasure tile. The amount of water tiles within the generated map is randomly generated, however, making sure that there is at least one path which the user can choose to reach the treasure.



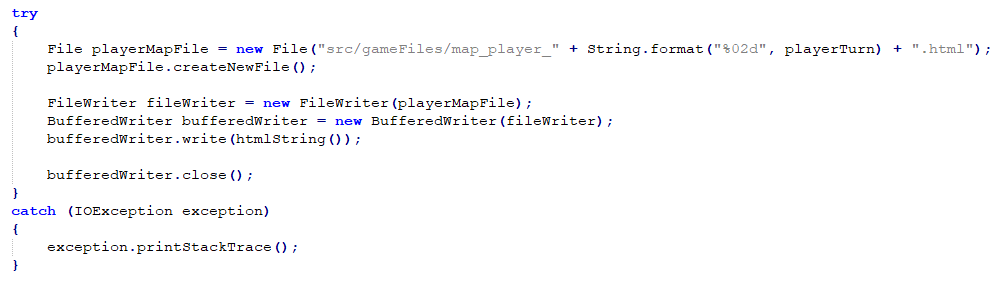
A number of players are created using the method called **setNumberOfPlayers** which can be found in the **Game** class. Every created player has a random starting position with the restriction that the random player position will be a grass tile, meaning that if the random starting position is a water tile or treasure tile the starting position of that player will change to a valid start position.



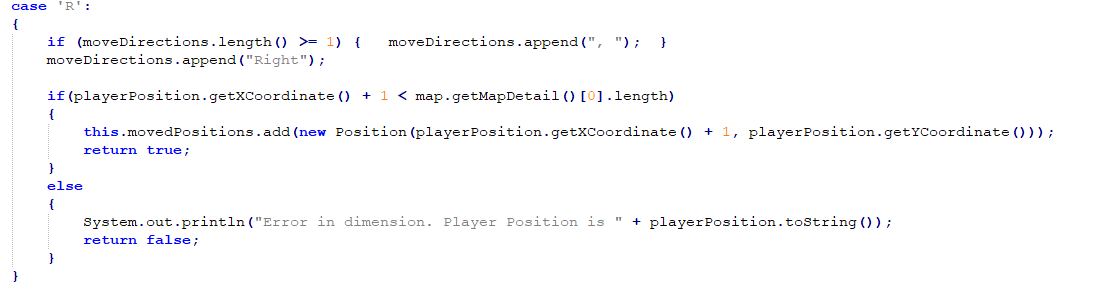
When the generated map and the players are created successfully the **startGame** method is called from the main method of the **Game** class. This illustrated method is responsible for choosing a random player to make the first move. The method continues looping until a player successfully finds the treasure within the randomly generated map.



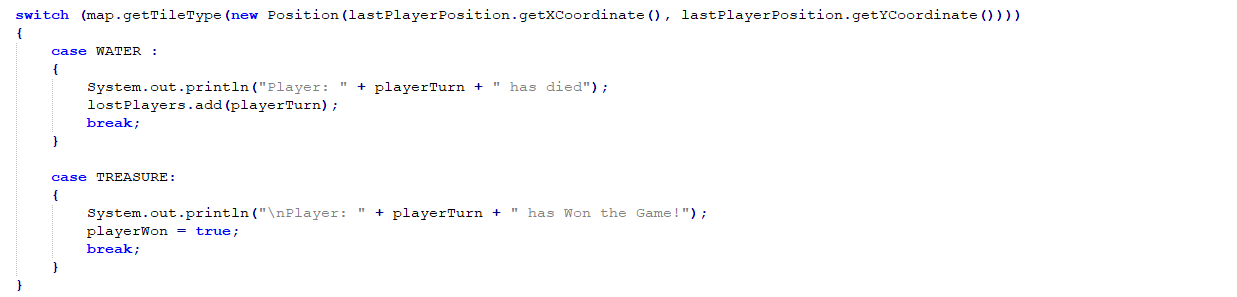
After a player has made a correct move a method is called which is responsible for generating the HTML for the current player who is playing the game. This is done by using the method **generateHTMLFiles** which creates an HTML file which can be found in the ‘**gameFiles**’ folder. A generated HTML file is generated for every player who is currently playing the game. The method which is responsible for generating the HTML makes used of a string builder to make sure each player views the tiles that were visited by the current player.



After the HTML file is generated the **startGame** method will make use of another method to check if all players are dead, if so the game stops. However, if the current player is dead the next player will be chosen to play. The **tryToMove** method takes input from the user and calls the **move** method within that player instance to change the current position. This method checks if the position requested is valid if this move is incorrect or not within the game map the method returns false. For example, when a player tries to move to the right bounds, this method makes sure that the current player does not attempt to go outside the game.



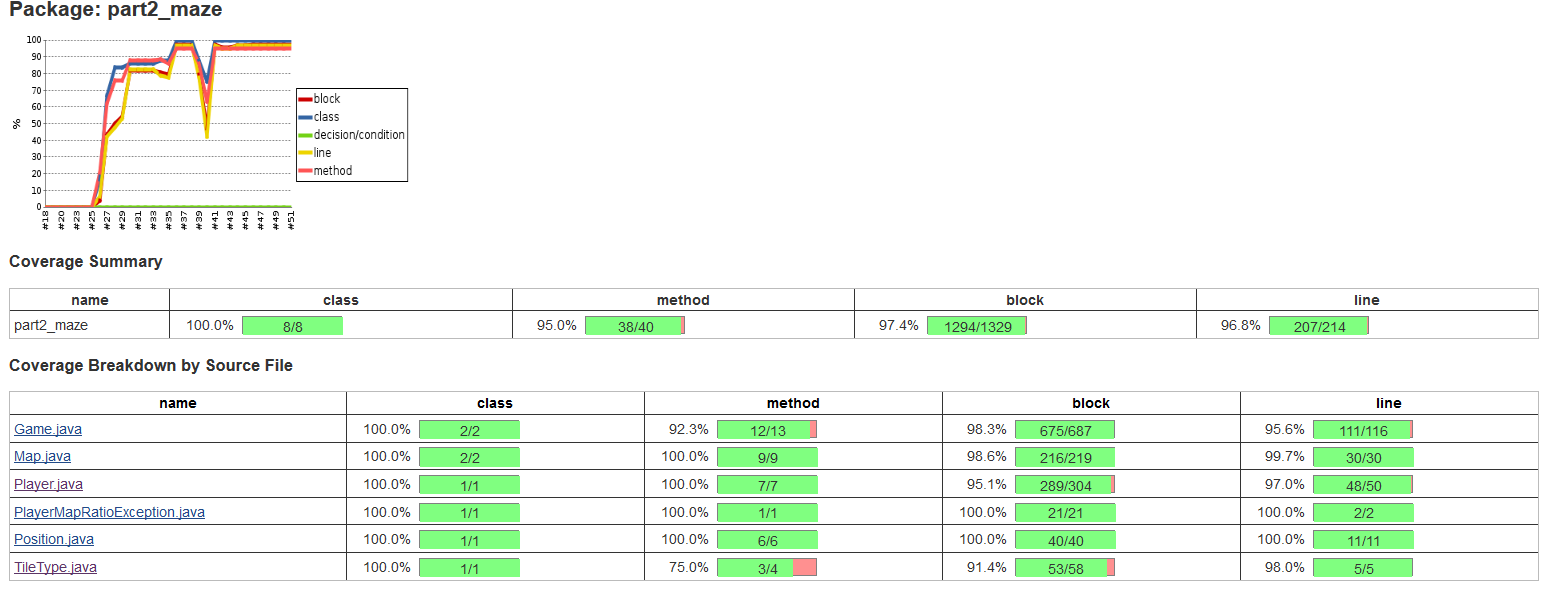
When the player performs a move the method **CheckMovedTile** in the **Game** class is used to check whether the new visited tile is a water tile or a treasure. If it is a water tile the player dies and cannot input any more moves. If the player wins the program notifies all the users that the current player won.



## Testing and Coverage.

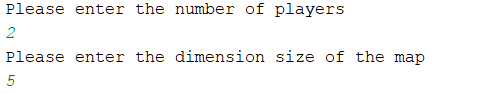
All the main classes were successfully tested with a total of **58** tests were created which test different features of all the classes. The tests were separated by each individual class and can be found in the **part2\_maze** folder in the test folder. Each test is labelled with the method used and output to generate. Each test class has a before and after method, however, the object created in the before method is not necessarily used within the test method. The amount of coverage in part 2 was **95%** for the methods and **96%** for the number of code lines.

The only method that was covered by the tested was the main method of the **Game** class since testing is generally not done for these types of methods. The main method only responsibility is creating an object for the class **Game** and calls the **startGame** method. In terms of lines, only the **IOException** of the method called **generateHTMLFiles** was not tested since the files were being constantly being created unlike when reading a file. Therefore, we could not lock the file stream buffer correctly to generate that exception.

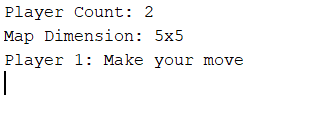


## User Manual

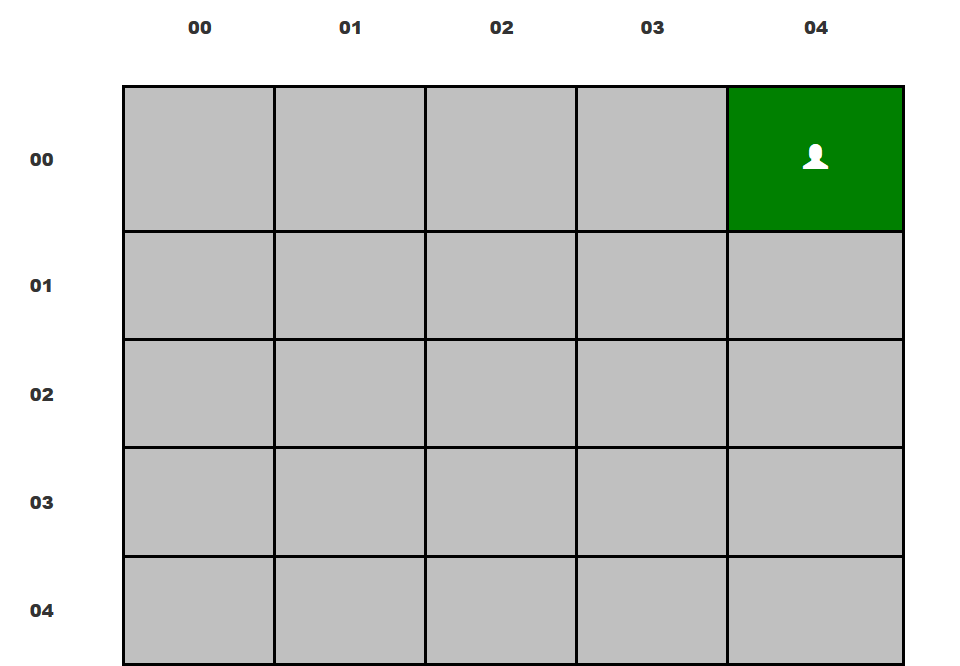
Initially, the user is asked for the number of players and the size of the generated map. If the inputs are incorrect the user re asked to enter these two values again and the program keeps on asking the user for the number of players and the map size until a valid input is made.



The game will choose a random player to make the first move hence, the program will request the user to input the direction where the player would like to move using the **U**, **D**, **L** and **R** keys.



The user currently playing can see the HTML file and choose where to move next. If the input is incorrect the program will ask the user to input a valid direction to move.



At the top of the generated HTML file, the player has the ability to view all the moves that were performed by the user including the attempted moves.

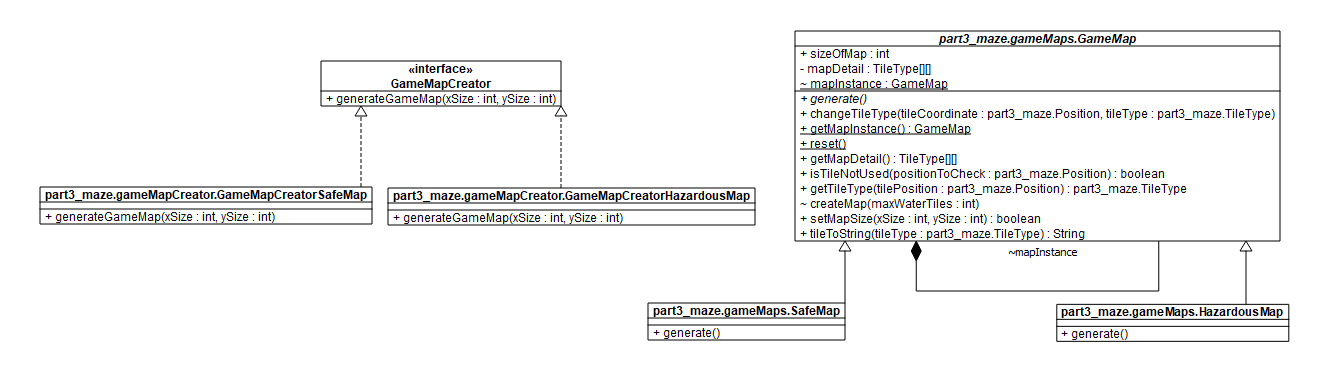


The game will ask the next player using a descending fashion. This process continues being done until one of the player successfully finds the treasure or until all the players die

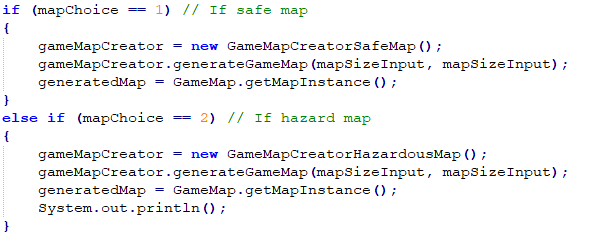
# CPS2002 Assignment – Part 3

## Feature 1: Different Map Types

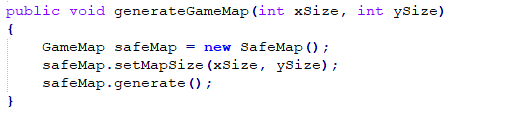
The first feature requested by the client was to create two different map types which are the safe and hazardous maps. For this feature to be implemented, the team implemented a factory design pattern which consists of two hierarchies one for the Game Map Creator and another one for Game Map.



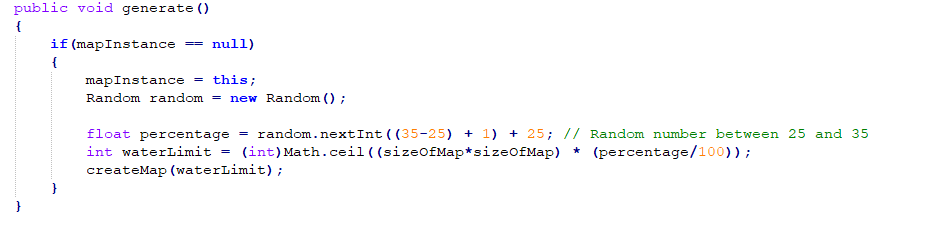
The factory design pattern has the producer which is the **GameMapCreator** hierarchy and the map generator being the **GameMap** hierarchy. In the **Game** class, an object of type **GameMapCreator** is used to create a map through the use of polymorphism. The object can be either **GameMapCreatorSafeMap** or **GameMapCreatorHazardousMap**. This is done in the method called **inputMapChoice** where the user is requested to input which map to generate.



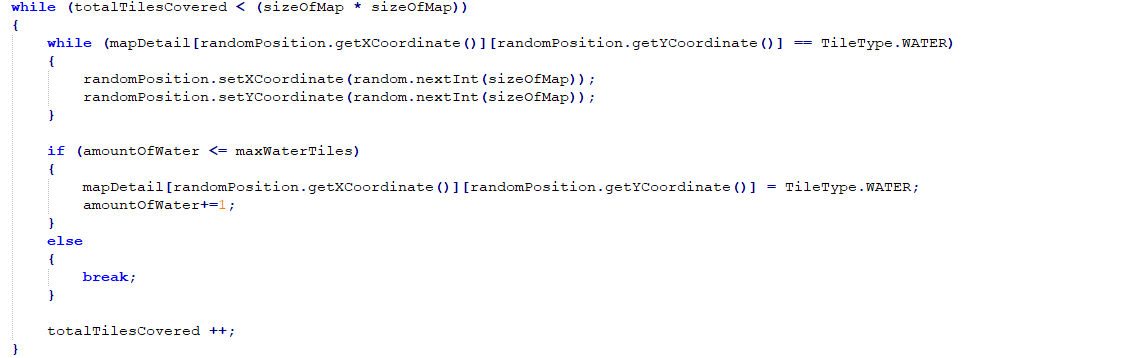
Each producer requests the respective class from the other hierarchy to create a map of that type and trough polymorphism the map is generated can either be a safe map or hazardous map. One can note that these methods do not return the object created which is the Game Map.



The key difference between each map type is the amount of water tiles generated. For example, a hazardous map has between **25%** to **35%** of the map filled with water tiles. Both map use the same **createMap** method with different percentages of water tile.



The map creation process has also changed to cater for different map types. This was done to make it easier whenever a client requests more map types. Whenever so only a new producer and a generator class are required.

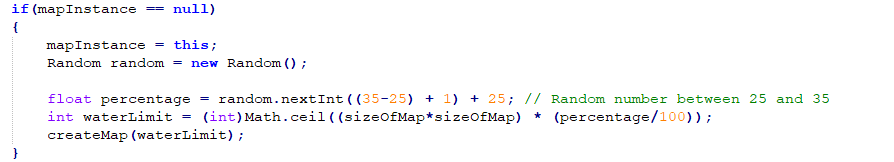


## Feature 2: Single Map Instance

The second request from the client was to create a single instance so that the server would not be overloaded with different maps for each player. This feature was implemented by using the singleton pattern within the **GameMap** class. A static object within the class is required to ensure only a single instance of the map is created.

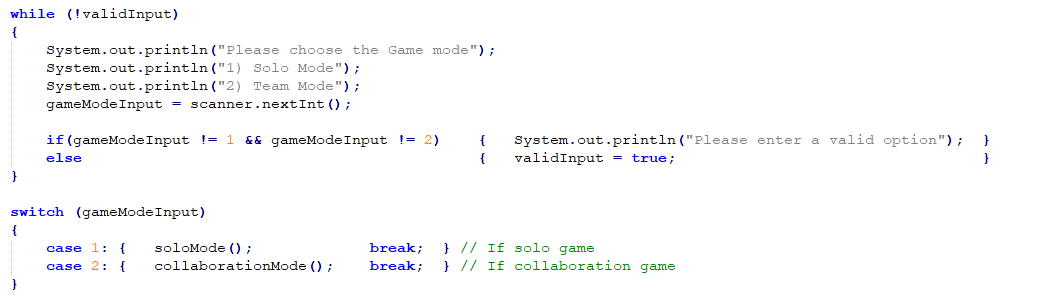


Within each sub-class of **GameMap** the static object is checked, and an instance is only created whenever the object is null. Whenever the Game class requests a new map multiple times only the first instance creates the map the others will be the same instance as the first regardless of map type.

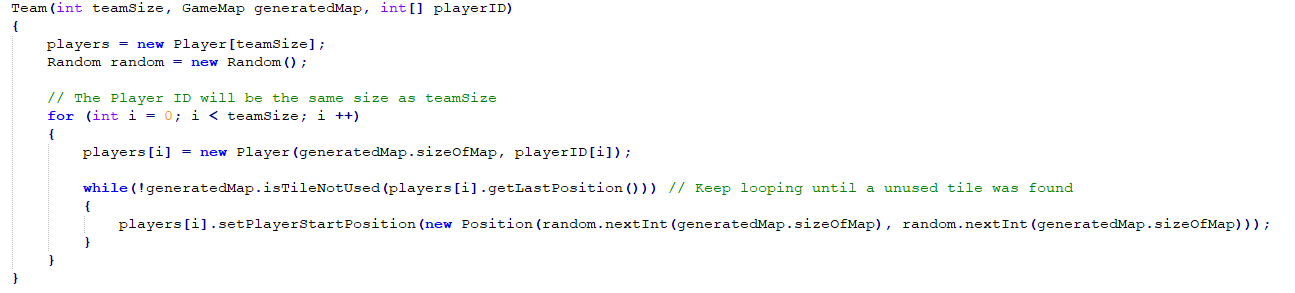


## Feature 3: Team Based Game

The last request from the client was to create two modes, solo mode which is the normal game mode and collaboration mode where the user is asked for the number of teams.

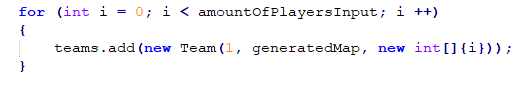


For both modes, a new class was created called Team. This class is used in both solo mode and collaboration mode. This class contains a list of players replacing the old method in the **Game** class that created players.



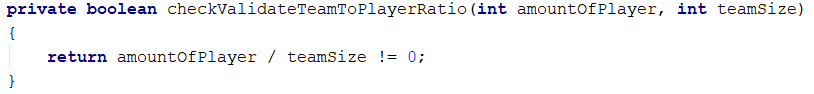
### Solo Mode

This mode works like the previous implementation; however, each player is added to a unique team creating an **ArrayList** of teams which contain only 1 player each. So, for example, if the user requests 5 players, 5 teams will be created.

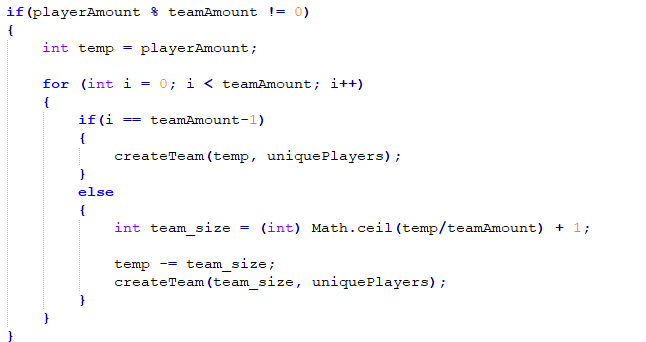


### Collaboration Mode

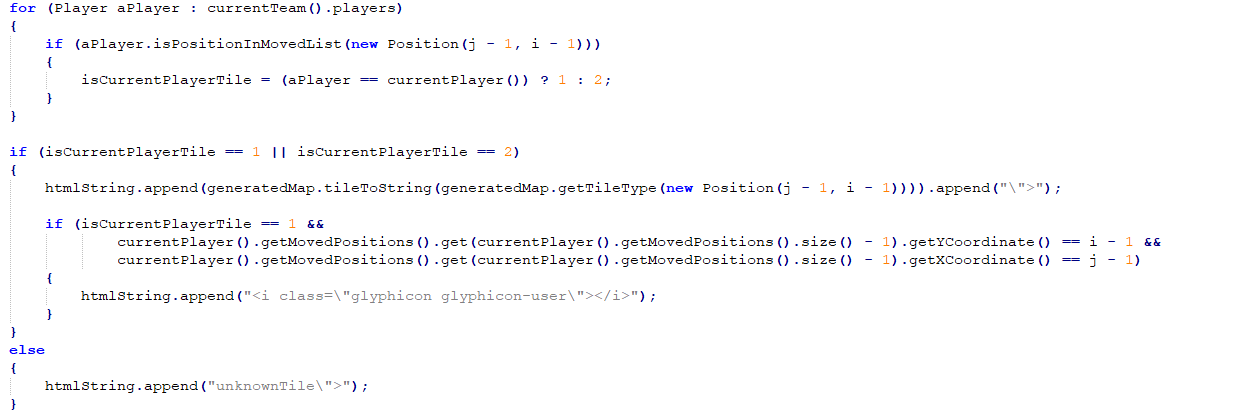
For this mode the user is requested to enter how many teams they would like to create however, a method is used to ensure that the amount of player is adequate for the number of teams.



The number of players divided by the number of teams does not have to be even the method **createTeams** handles this by creating teams with each have near to equal size each whenever the ratio is odd.

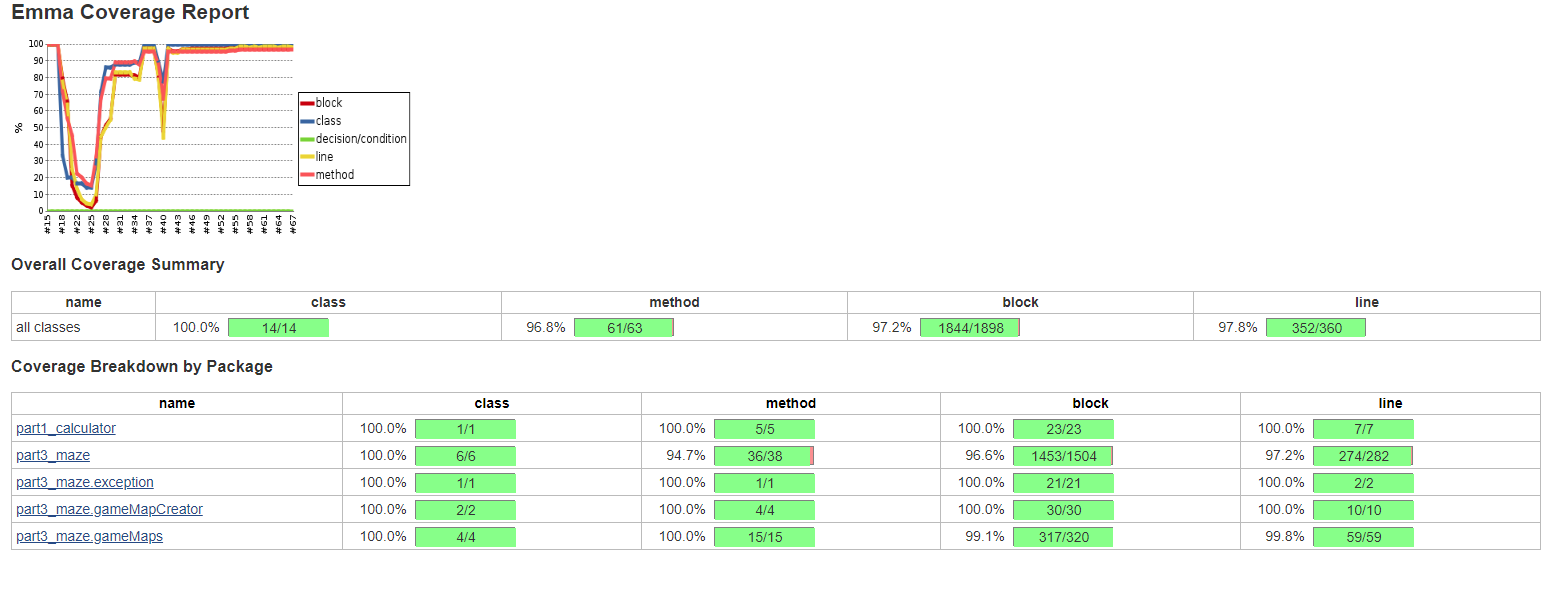


The HTML will now show all the moved positions for each player in a team to each player in that team.



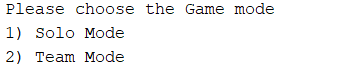
## Testing and Coverage.

All the new features were successfully tested except for the main method and **IOException** That was previously mentioned in the last section. Each design pattern was also tested to ensure they are fully operational. Tests were split in their own respective class however, only one test class was created for the factory design pattern. In total 65 tests were created with 96.8% of methods and 97.8% of lines covered.

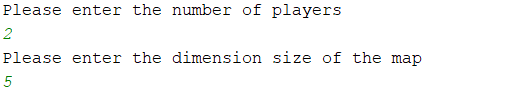


## User Manual

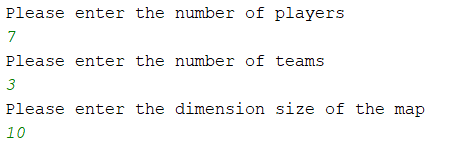
Initially the user is asked to choose between solo mode and collaboration mode.



For solo the mode the user is requested to enter the number of players and map size.



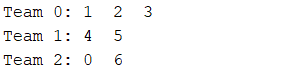
For collaboration mode, the user is requested to enter the number player, number of teams and map size.



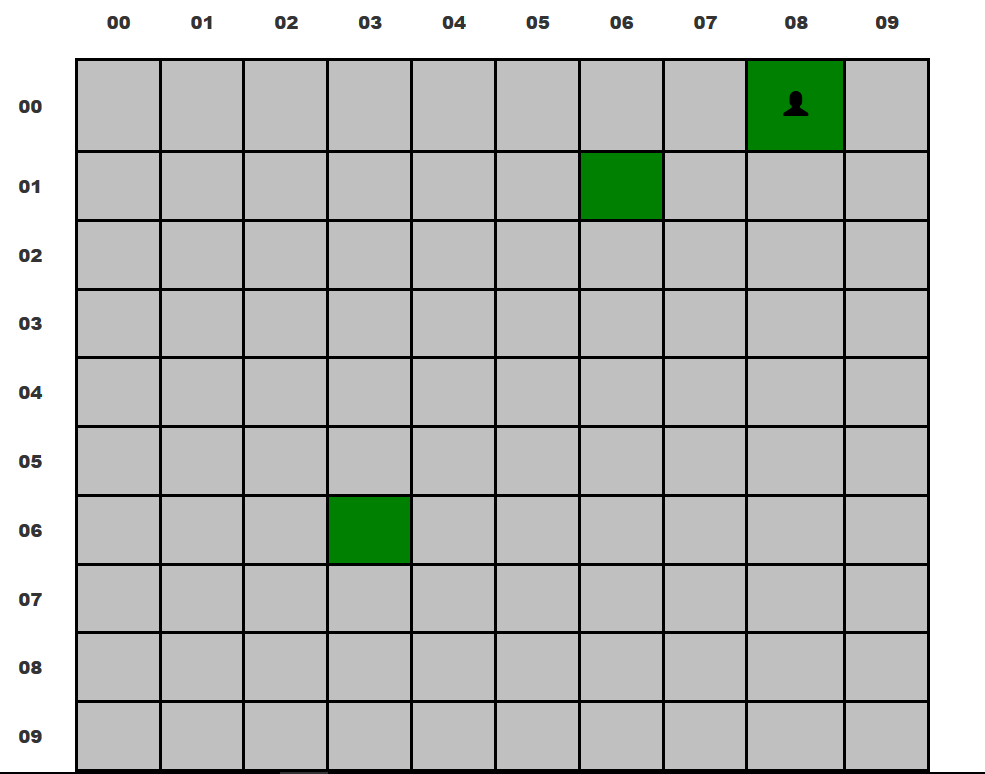
After, in both modes the user is requested to choose a map.



When doing so in collaboration mode the teams are displayed to the user. Keep in mind that when the player quantity to the number of team’s ratio is odd the size of each team might be different.



User input for both modes is the same however in collaboration mode a player can see the tiles uncovered by his teammate. Even their starting position.



Only valid moves are shown of that player in the HTML. Only the moves of that player will be shown regards of what mode the system is in.



Player cycle numerically in both modes regards of what team they are in. when a player dies the system will switch to the next alive player. Both modes also operate the same whenever a player win.