CPS2002 - Software Engineering

Course Assignment

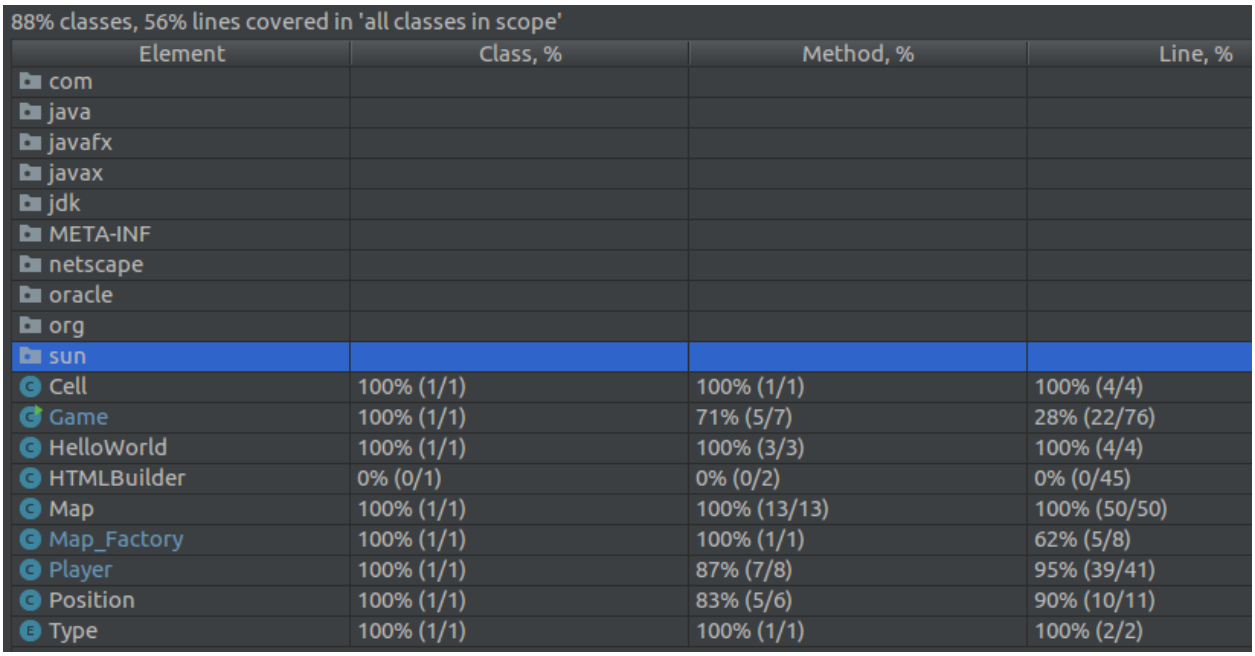
Neil Sciberras 79798M
Franklyn Sciberras 41498M

University of Malta
Department of Computer Science
21st May 2018

Contents

1 Code Coverage Analysis

In the first section, we were to provide the code coverage metrics for the code provided for the game. The results we achieved were the following:



88% classes, 56% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %
com			
java			
javafx			
javax			
jdk			
META-INF			
netscape			
oracle			
org			
sun			
Cell	100% (1/1)	100% (1/1)	100% (4/4)
Game	100% (1/1)	71% (5/7)	28% (22/76)
HelloWorld	100% (1/1)	100% (3/3)	100% (4/4)
HTMLBuilder	0% (0/1)	0% (0/2)	0% (0/45)
Map	100% (1/1)	100% (13/13)	100% (50/50)
Map_Factory	100% (1/1)	100% (1/1)	62% (5/8)
Player	100% (1/1)	87% (7/8)	95% (39/41)
Position	100% (1/1)	83% (5/6)	90% (10/11)
Type	100% (1/1)	100% (1/1)	100% (2/2)

Figure 1: Code Coverage Metrics

From the previous figure we can notice that even though most of the classes have a 100% code coverage, there are some which lack the percentage, either by a small or even a large margin. In the following subsection we will explain the reasons behind such coverage.

1.1 Metric Explanation

1.1.1 Game

The first class that doesn't have a 100% code coverage metric, is the class **Game**. Such a class mainly has two methods which are not covered, as can be observed from the previous figure. These two methods are the methods:

1. **generateHTMLFiles()**
2. **main()**

The first method **generateHTMLFiles()** was not tested, since we did not feel the need to test it. Since the only job of such a method is to create a new html file for each player in the game, there was no need for software testing. In order to test that such a method is functioning properly, one only has to check the amount of players that are in game, and check the amount of html files that have been created. If they are equal, we know that the method is properly working.
The second and final method in this class is **main()**. The main method was was not tested since it's primary function is that to prompt the user with the required output at the proper time, and gather some required information for the game to proceed. The rest of the functions it makes use of are already defined and tested for somewhere else. For such reasons there was no need to design a specific software test for such a function, since a simple run of the program would have been a necessary enough test in itself.

1.1.2 HTMLBuilder

The second class that doesn't have a 100% code coverage metric, is the class **HTMLBuilder**. Such a class has none of its methods covered with tests.
The first method not to be tested is the **HTMLBuilder()**. The constructor was not tested since it's job is to create a text file with player id. In order to check whether this was working properly one could just go to the assigned file directory and check whether the file was there or not, with the proper player id's.
The second and final method that was not tested in this class is **writeMapToFile()**. Such method was used to go to the html file produced and fill it with the appropriate code, which could correspond to the current map status. We felt no need to test such a method's functionality, since this method produced an html file which could be inspectioned physically if an error occured. If a software error, that cannot be caught by the eye occurs, such as the file could not opening, the error would be caught through exception handling.

1.1.3 Map_Factory

In this class, even though all the methods were covered, we still didn't get 100% line coverage, since in the tests we only used the instance of a hazardous map. However since they both have the same architecture and only the amount of water tiles is different from the safe map, we felt no need to create another test using a safe map instance, since for these kind of tests, it would have changed nothing of any importance.

1.1.4 Player

Player is also another class, that doesn't have a 100% code coverage. This happens mainly because of two instances in the coding.
The first instance is the instance when a random starting position is a water or treasure tile, and how would the player in terms of starting position would behave. Even though we covered such an eventuality by code, the code covering it was not tested for. This is because the map generated is randomly generated, and we have no means of knowing beforehand where these tiles will be placed. For such reasons we could not create a possible test scenario for this. However through physical testing and inspection we made sure that this part of code works fine.
The second instance is the test regarding the **getMap()** method. This function was not tested for since it is a simple getter, which is only used to return the private map copy of the player.

1.1.5 Position

Finally the final class which was not fully covered by unit testing is the class **Position**. This class only has one method which is not covered, and that is the **toString()** method. Such a method was not tested sinc it's job is to print position co-ordinates on the console. Since this method only produces an output without any other hidden logic, there was no need for testing, since any error could be easily caught visually (apart from the fact that it's use is so basic, that there is no need to test it).

2 Design of Basic Version

Class **Game** containing the **main method**, initially asks the user to enter the number of players and the size of the map. **Map**'s **setMapSize()** will return true if the entered size and number are valid, and if not the user will keep being asked until he enters valid configuration details.

Once the game details have been entered correctly, **startGame()** is invoked which sets the map size, creates the array of requested number of players, and generates a random initial position for each of them. An **HTMLBuilder** object is created and in turn generates the HTML file for each player in the game. There only exists one map in the game, but each player will have a different HTML file, which reveals different parts of the map based on where the player is.

The game will then loop until the game is declared won. At each iteration, the users are prompted one by one to enter their next move. **Player**'s **move()** will return true when the move was a valid one and

false when not. The user will be notified when his move was invalid, but he is not prompted to enter a new one.

When the player makes a valid move, it is checked whether the move resulted in the particular player stepping on the treasure or on a water tile. When moving on a water tile the player is moved back to his initial position using **moveToInitial**, and when he reaches the treasure, the corresponding flag in **Game**'s **treasureFlags[]** is set to true. The HTML files are updated uncovering the new tiles the player moved to. Then the **treasureFlags** array is iterated once to check if any player has won the game, and if so displays the winning message and exits the loop.

The classes used and their design can be seen in **figure 2**.

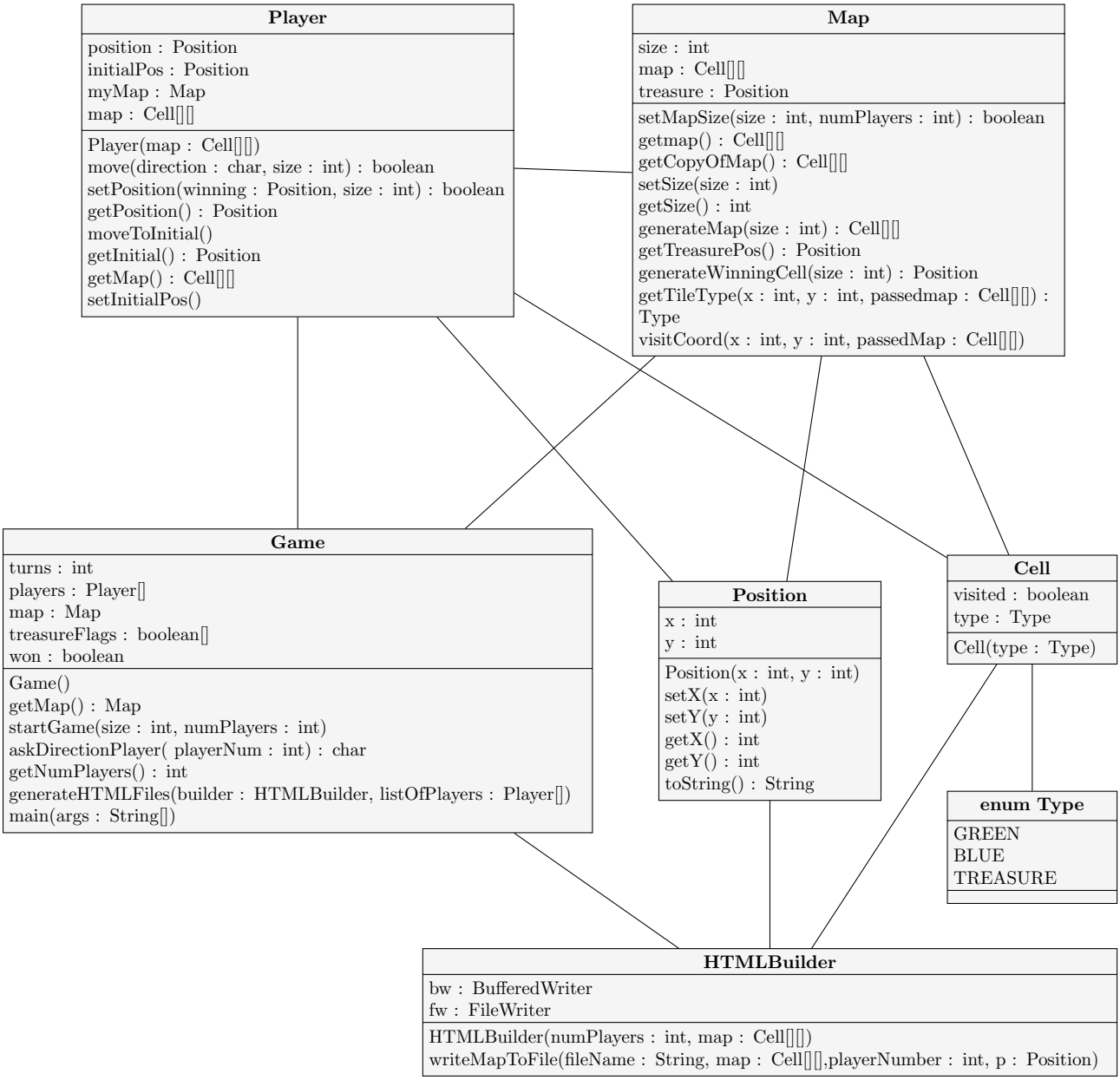


Figure 2: UML Class Diagram of the basic version of the game

3 Enhancements

3.1 Different Map Types

The **Factory Design Pattern** is used for the problem of having multiple types of maps, and possibly more map types created in the future. This design pattern shifts the responsibility of creating the map, from the **Game** class to a newly created **Map_Factory** class. This does all the checking of which map should be created, and in turn returns the correct map instance. It makes it possible to add more map types by simply amending the factory class, and leave the **Game** intact. In order to test this, two different map types, were introduced. The first map is a safe map, where only 10% of the tiles are marked as water tiles. The second kind of map is the hazardous map. This map will have between 25% and 35% of it's tiles marked as water tiles.

Game will hold an instance of a **Map_Factory** object, which will be used to call its **getMap()** method. This method will in turn do the necessary checks to identify the type of map that is to be created and then call the constructor of **Map**, seen in **Listing 1**.

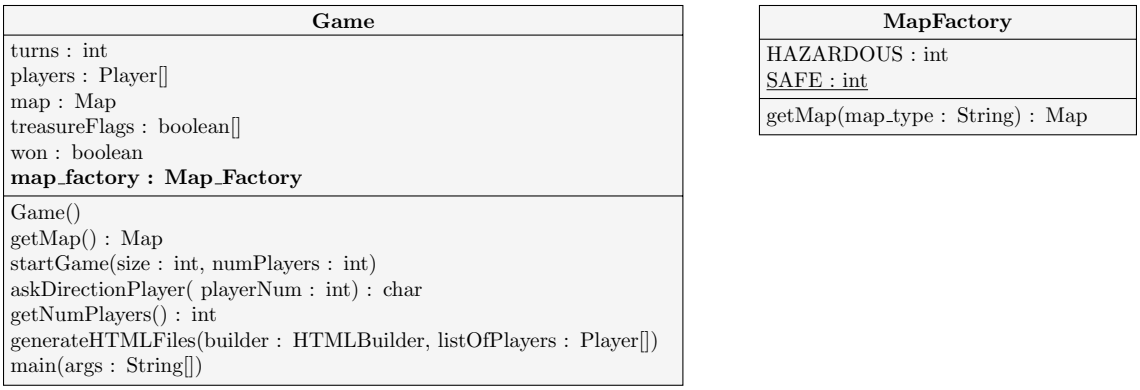


Figure 3: Factory Design Pattern UML Class Diagram[1]

```
1 public Map(int probability_water)
2 {
3     this.probability_water = probability_water;
4 }
```

Listing 1: Constructor of class Map

3.2 Storing One Map in Memory

To ensure that a single map is stored in memory, rather than one for every player, the **Singleton Design Pattern** is employed here. This only lets one instance of a class be created, and normally sets the constructor as **private** so it cannot be called. But it was decided that all the constructor does is it sets the **probability_water** field of the map instance to the argument passed.

When **getMapInstance()** does, is it returns the instance of the map if there is one, and if there is no instance yet, it creates one using the constructor mentioned above. Then if there is the need to remove the only instance of map there is, **resetInstance()** can be called which just sets the instance to NULL. Having these two basic methods, now every other class that needs a map instance, can only call **getMapInstance()** and not the constructor.

[1]Static attributes are denoted by underline, final attributes are denoted by uppercase naming, and newly added attributes or operations are denoted by bold text

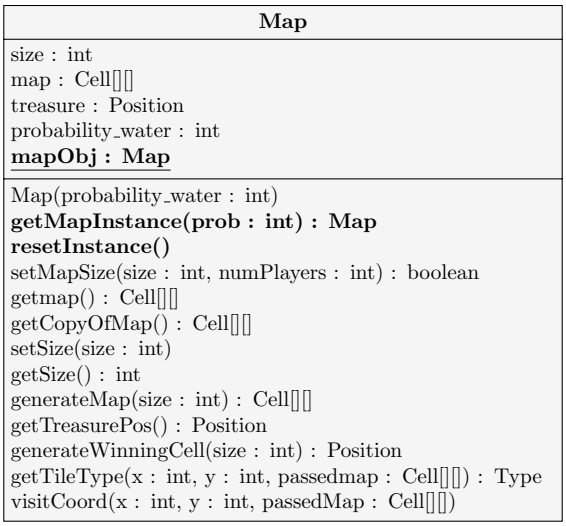


Figure 4: Singleton Design Pattern UML Class Diagram

3.3 Team Exploration

When teams are introduced into the game, players need to be able to see every tile that members in their team have visited. This means that the HTML files of players need to be amended, whenever there is a change in any of the members' files. This can be implemented by having a central HTML file for each team, and each member is required to update this central HTML file apart from updating his own. Then all the members' files are registered as observers of this central file so that they can be updated automatically. Note that this enhancement was not finished in time due to time restrictions. In the source code one can only find it's first version class **flagWatcher**. This class would be watching the array which holds who has won and who has not, and whenever a player from a team has won, it would notify all of the observers of the players of the same team that they have won.

4 Screenshots

4.1 Initialisation Of Different Maps

As previously mentioned, there are currently two kind of maps implemented in the game. Below we will see an unveiled map, one of each kind. From these figures, one can notice that **Map_Factory** class is working as designed, since we are able to see a difference in the instance of the map created, depending on the type of map.

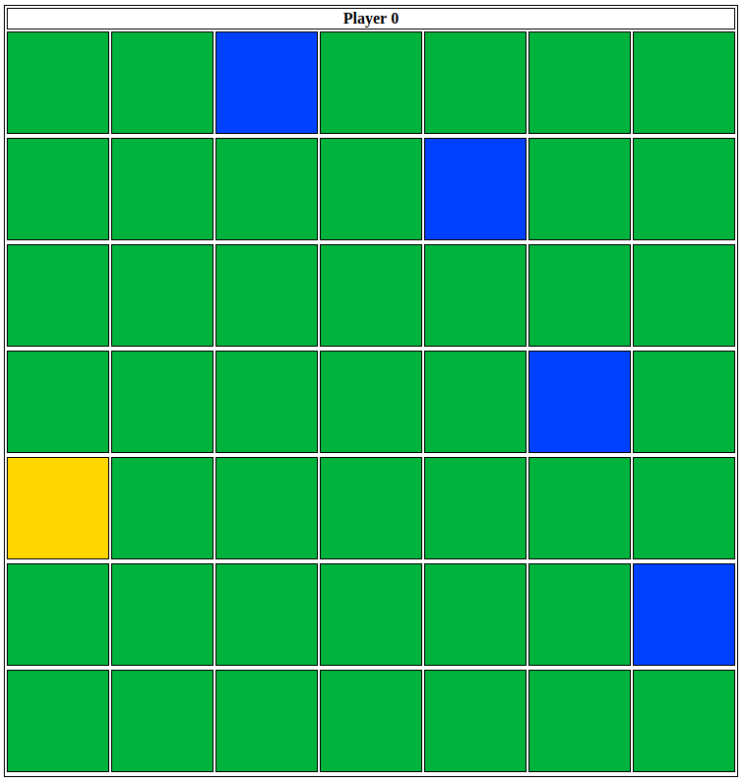


Figure 5: Safe Map

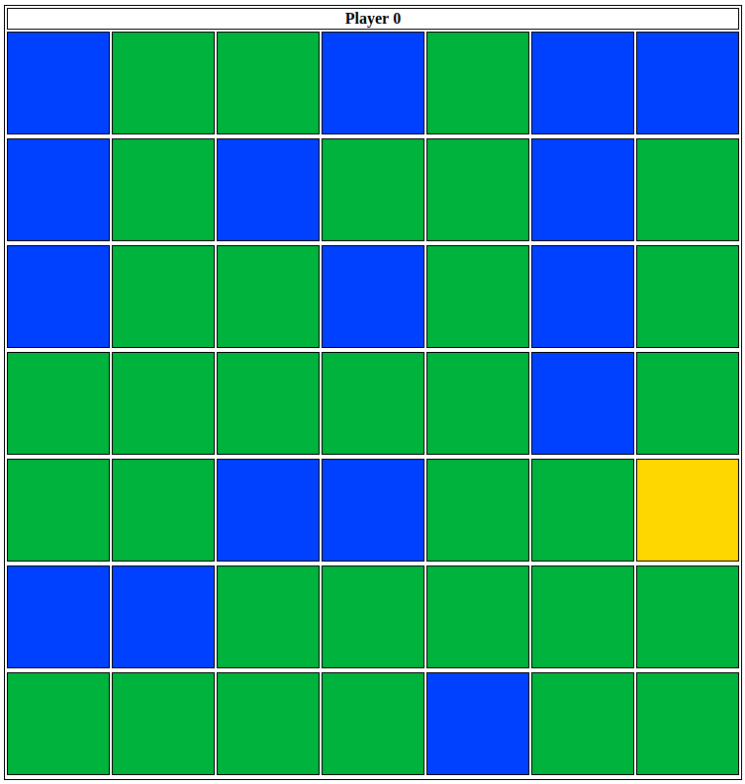


Figure 6: Hazardous Map

4.2 Player Stepping On Water Tile

From the given game description, we know that whenever a player steps on a water tile, he should be put back on his starting tile. In the following figure we can see that the player performed two left moves, however on his second left move he found himself upon a water tile. In that instance the game notified the player of the occurence and put him back on his initial tile.

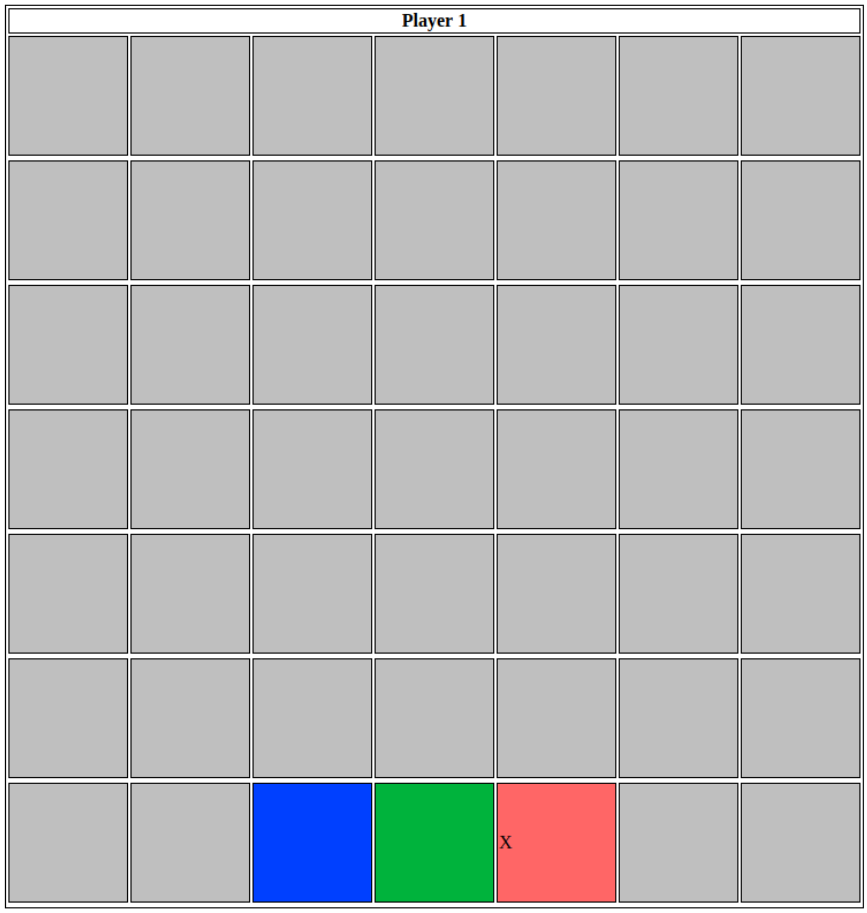


Figure 7: Map Instance - Step On Water

```
Player 1 you moved on a water tile (6,2) ! You have been sent back to the initial position (6,4)
```

Figure 8: Game Output - Step On Water

4.3 Player Winning

In order for the game to finish, at least one player has to find the treasure. If multiple players find the treasure in the same round, they all win. In the following case scenario, player 0 was the only player to find the treasure. The game got hold of that instance, and notified the player accordingly as we can see from the following two figures.

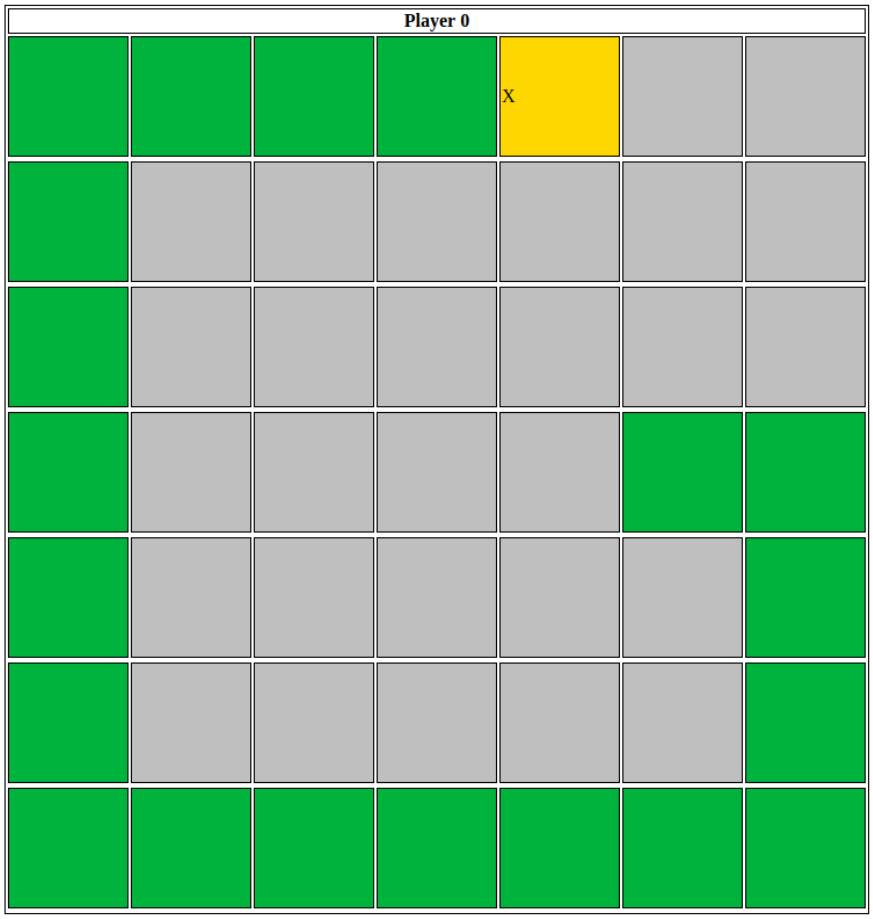


Figure 9: Map Instance - Find Treasure

Figure 10: Game Output - Find Treasure

[1]Note that the previous screenshots were captured from different games.

5 Instructions

In order to run and play the game, the following steps have to be taken:

1. Open the game project in IntelliJ
2. Build the project locally
3. Run the main class Game.java
4. The terminal will ask you to enter an amount of players. You should be able to enter between 2 to 8 players. If these numbers are not respected you will be notified accordingly.
5. Enter the map size you and your partner/s wish to be playing in. If you are between 2 and 4 players, the minimum map size you can choose will be that of 5 by 5. If you are between 5 and 8 players the minimum map size you can choose will be that of 8 by 8. The maximum map size will be that of 50 by 50 at all times. If these numbers are not respected you will be notified accordingly.
6. You are to choose between a safe version of the map or a hazardous version of the map, by pressing 1 or 2 respectively. Any other input will be assumed as the default map, i.e. the safe map.
7. Now an html file for each player will be generated in the directory where there is the project file. Each player is to open his html file, which is denoted by the player number.
8. The game will prompt each player to input a direction he wants to move in. You are to input one direction choosing from r:right, l:left, u:up and d:down.
9. After all the players have input their choice, they will be prompted with their new position, or if any special occurence, happened (ex. player/s won or player/s stepped on a water tile), they will be notified.
10. After all the players have read their notification from the terminal, each player is to press refresh on the browser holding his html file to see his current position on the map.
11. You are to go back to step 8, up untill there is a winner.