FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I / We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I / We* understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Jacques Vella Cutajar
Student Name

Signature

Jonathan Cutajar
Student Name

Signature

Student Name

Signature

Student Name

Signature

CPS2002
Course Code

Assignment - Web Titles Game
Title of work submitted

23/06/2020
Date

CPS2002 - Software Engineering

Water Tiles Game

https://github.com/jacquesvcschool/CPS2002

Jacques Vella Critien, Jonathan Cutajar

Contents

1 Task 1A: Initial Setup

The first thing we have done for this assignment was the initial setup for our working environment. As you can see in the below section, we first created a repository on Github for source control and then proceeded by setting up Jenkins for automation.

1.1 Github

The link to the repository is **https://github.com/jacquesvcschool/CPS2002**.
Here, one can see all the commits, source code and branches used for different features which were created using the git flow commands.

1.2 Jenkins

The next step was creating a new Maven project on the University's Jenkins server.
The configuration for our project can be found at:
https://jenkins-ict.research.um.edu.mt/job/CPS2002-Jacques-Jonathan/

1.2.1 Linking project to Github

The first part of the configuration consisted of linking our Jenkins project to our source code repository by doing the below setup.

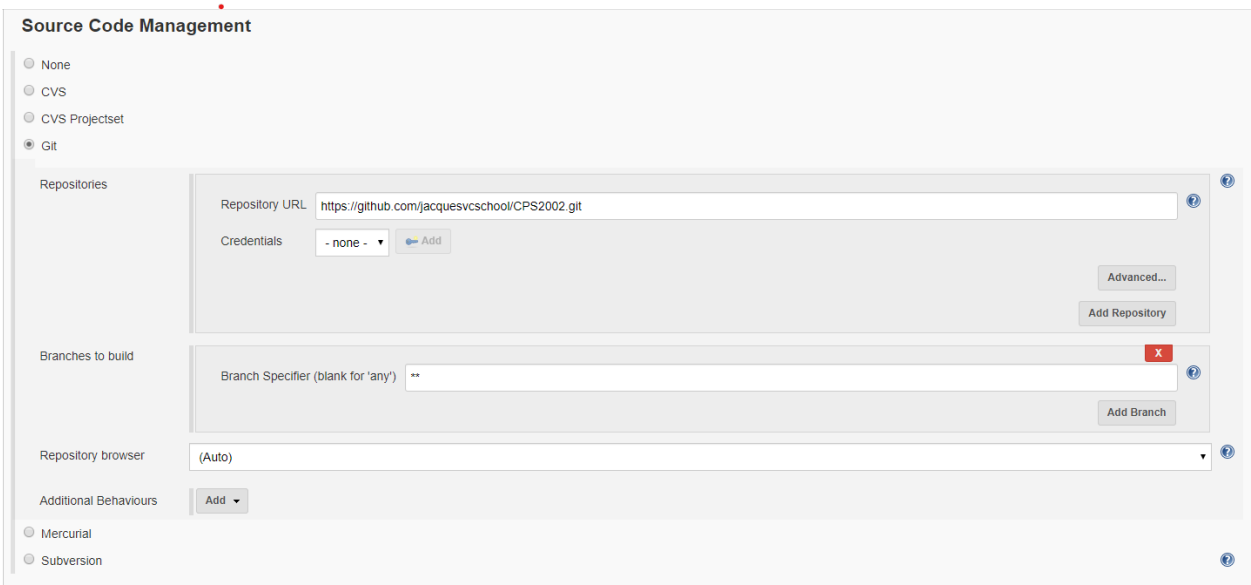

Figure 1: Linking Jenkins project to our Github repository

1.2.2 Polling for changes

This will enable the source code to be pulled, compiled on tested on every change committed to github. This is done by polling the repository for changes every two minutes. The setup for this can be found in the figure below.

1.2.3 Building the code and testing

The setup in the figure below shows the maven command we will be using to build and test the code upon signalling a change.

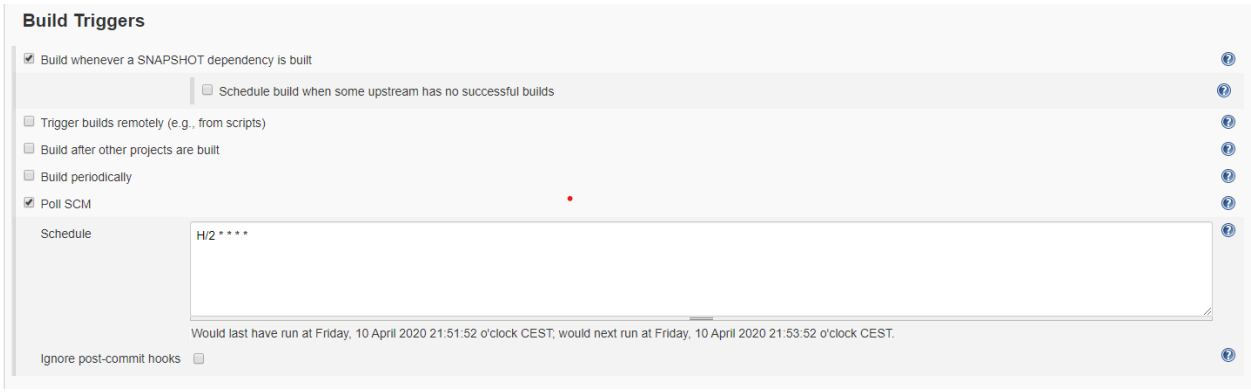

Figure 2: Polling our Github repository for changes

Figure 3: Setup for building and testing the change

1.2.4 Build notifications

The below setup was done so that we could receive notifications via email whenever a build is triggered and there were errors or test failures

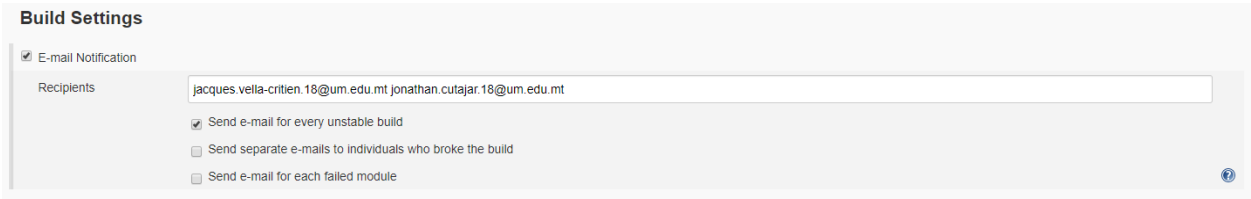

Figure 4: Setup for notifcations for failing builds

1.2.5 Testing the environment

To test the environment, we created a simple class called ShellCode which just returns a 'Hello World' string and a test class which asserts the output of the function returning the string. Then we did some commits to make sure that the code fails and an email is sent. Then, we recommitted the fixes to make sure that the build successfully completes. The build status of these commits can be seen in the figure below.

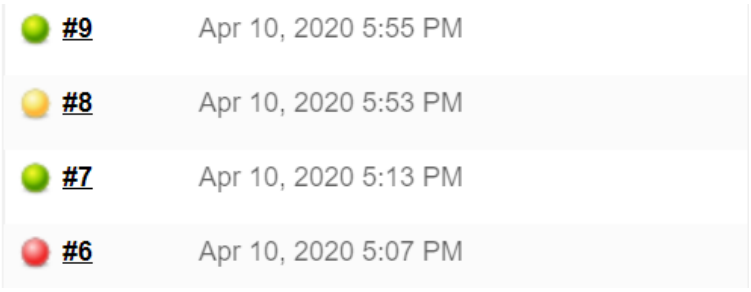

Figure 5: Build statuses for build fails, test fails, and successful builds

2 Task 1B: System Delivery

For this part of the assignment, we were required to create a multiplier game in which 2 or more players compete to explore the map and trying to find the treasure before any other player. Moreover, for this implementation we were also required to make use of the Gitflow process when implementing this task. After the game was implemented, a tag had to be created to be able to pinpoint the end of this part of development for this game. The tag can be found in the following link: `https://github.com/jacquesvcschool/CPS2002/releases/tag/1.1`. It is important to note that the GitFlow process was also used to release the first part of the assignment, however the branch named release/part1 had to be removed in order to be able to start the release for part two. The figure below proves that there was an actual branch for this release.

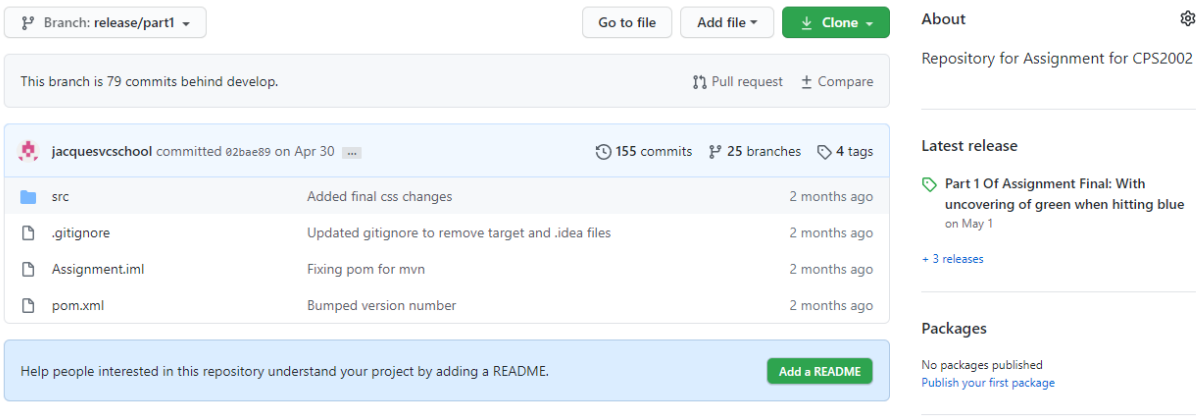

Figure 6: Branch for first release

2.1 Code Coverage Metrics

One of the requirements at the end of the implementation of part 1 of this assignment was to take note and record the code coverage metrics at this point. As can be seen in the image below, all the code was covered by testing classes, however some classes were omitted from the coverage counts as will be explained further below. A full report of the coverage can be found in the HTML file called index in the path '\target\site\jacoco'. Moreover, if one would want to regenerate the metrics, this can be done by running **mvn clean install** in the source directory.

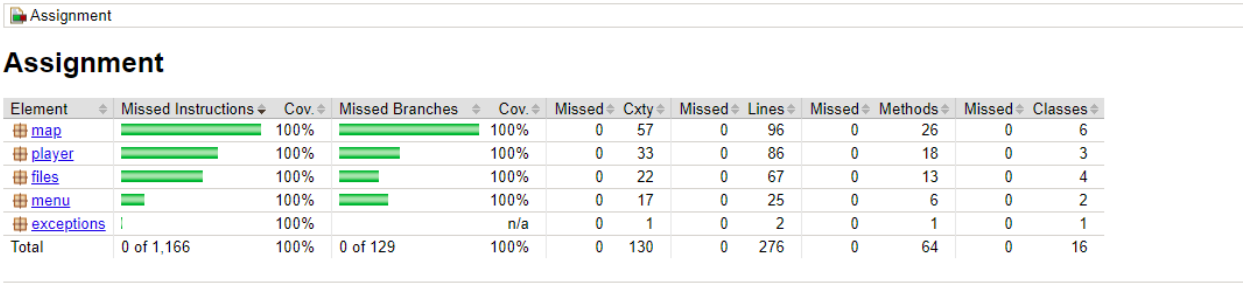

Figure 7: Code coverage metrics for part 1.

The following are the two classes omitted from the testing coverage count:

1. **Game.java**: This is the class which contains the methods used int the **main()** method, also found in this class, which is used to execute the game. Despite not being in the code coverage count, one can still find a GameTest test class. This contains tests which test the main method and some of the class' methods. In fact, there is also a simulation of the game. However, It was omitted from the coverage class because since some of our methods, for instance, the main method, are static, PowerMockito had to be used and Jacoco's code coverage does not count these tests and hence, was excluded from the test classes to count just for the coverage numbers to look better.

2. **Helper.java**: This is a class which contains methods to copy, read and write from and to files. Since these are general methods used and tested by the programming community, tests for this class were not implemented. For this reason, this class was omitted from the tests to check for coverage so that the coverage numbers actually show what really needed to be tested.

2.2 Design Details

Another requirement in this specification was that we had to give an explanation about the design of the basic version of the part, being this part of the implementation. In order to this, below will be a list of all the classes and the methods found inside them followed by a class diagram which shows how these classes interact with one another to produce the final implementation of this game.

2.2.1 MapNotSetException

: This is an exception class found in the exceptions package and is thrown when a map operation is performed but the map is not yet generated. The only method in this class is a pubic constructor which throws the exception with the message passed

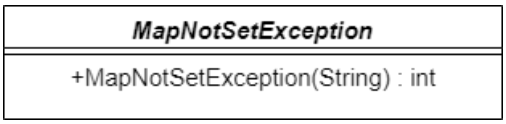

Figure 8: MapNotSetException Class diagram

2.2.2 Helper

This is a helper class found in the files package and is used to read, write and copy to and from directories and files.
These following is a list of methods found in this class:

1. **getAbsPath(String)**: This is a method used to get the absolute path of the path passed in the parameters. The path passed is usually a path relative to the resources folder in the src directory.

2. **readResourcesFileAsString(String)**: This is a method used to get the contents as a string from the file path passed as parameter.

3. **writeFile(String, String, String)**: This is a method used to create a file with contents. The first parameter is the directory where to place the new file, the second parameter is the file name and the third parameter are the file contents as a string.

4. **copyDirectory(String, String)**: This is a method used to copy a directory. The first parameter is the source directory to copy while the second parameter is the destination where to place the directory.

5. **copyFile(String, String)**: This is a method used to copy a file. The first parameter is the source file path to copy while the second parameter is the destination where to place the file.

6. **createDirectory(String)**: This is a method used to create a directory with the file path provided as parameter.

7. **getOccurences(String, String)**: This is a method used to check how many times a sequence occurs in a string and it is used in testing. The first parameter is the actual string while the second parameter is the sequence to check for.

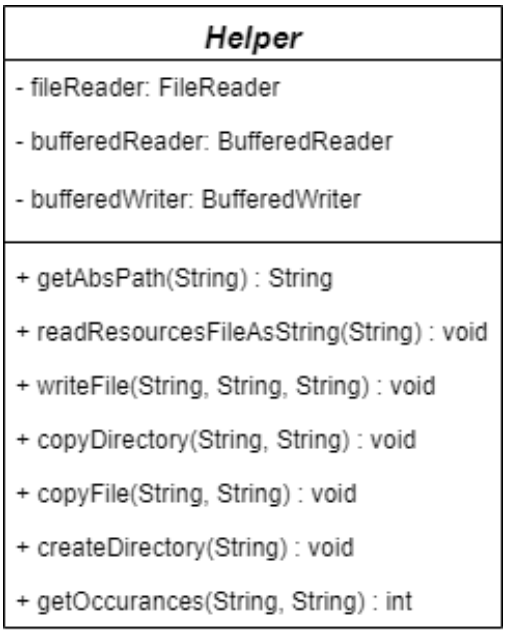

Figure 9: files.Helper Class diagram

2.2.3 Page

This is a class found in the files package and is used to represent an HTML page. As can be seen below this class contains one string member variable to hold the HTML of the page, a getter to get the HTML and a setter for the HTML.

2.2.4 Builder

This is an abstract class found in the files package and is used to represent a builder for the HTML of a page These following is a list of methods found in this class:

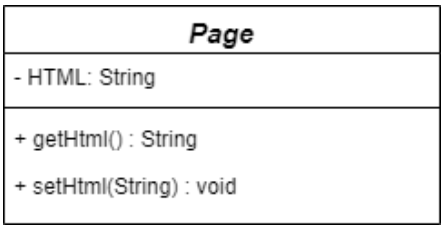

Figure 10: Page Class diagram

1. **getPage()**: This is a method used to get the page set.
2. **init()**: This is a method used to initialise the page's HTML to the template file contents.
3. **buildTitle(int)**: This is a method used to create the title for the page where the int parameter is the player id to fill.
4. **buildMapView(Player)**: This is a method used to create the map view of the player passed as parameter. This is done by using the list of visited tiles in the Player class.
5. **buildMoves(Player)**: This is a method used to create list all the moves of the player passed as parameter. This is done by using the list of moves performed by the player in the Player class.
6. **buildWinner(Player)**: This is a method used to add all the necessary HTML for the winning player passed as parameter.

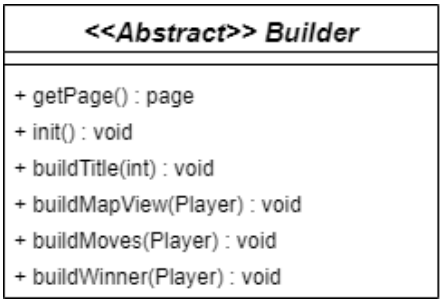

Figure 11: Builder Class diagram

2.2.5 MapResultBuilder

This is a class found in the files package and it extends the Builder class to implement the builder method for the page which shows the player progress. This includes a Page member variable and overrides all the methods explained above in the section highlighting the Builder class. Moreover, it also consists of two other string member variables used to hold the filename and the HTML for the player position.

2.2.6 Director

This is the last class found in the files package and it is used to direct the builder class on building the HTML for the page. In fact, this class only contains a member variable of type Builder, a constructor and a method called **construct(Player, index)** which given a player and an index, it constructs the HTML page by calling the builder class assigned in the constructor.

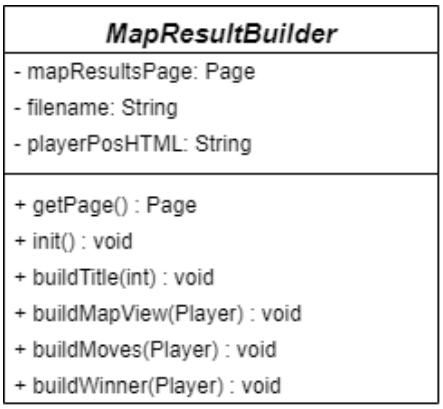

Figure 12: MapResultBuilder Class diagram

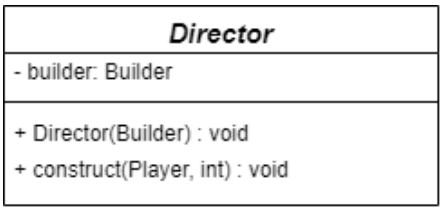

Figure 13: Director Class diagram

2.2.7 TileType

This is an enumeration found in the map package and is used to hold the three types of tiles, namely, Blue, Green and Treasure.

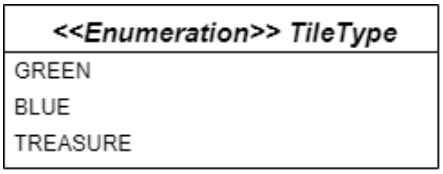

Figure 14: TileType Class diagram

2.2.8 Tile

This is an abstract class found in the map package is used to represent a tile. This class contains a member variable to hold the HTML for a tile, a getter for the HTML, a getter for the class name which is the CSS class of the Tile and a getter for the type of tile which returns a TileType enumeration value.

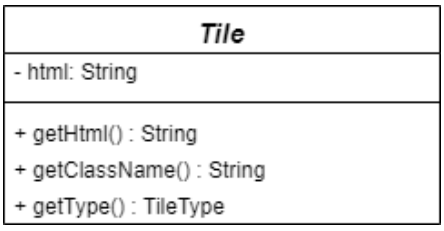

Figure 15: Tile Class diagram

2.2.9 BlueTile

This is a class in the map package and is used to represent a blue tile. In fact, it extends the Tile class. This class contains two other member variables named type and className to hold the tile type and the CSS class name respectively. Moreover, it overrides all the methods found in the Tile class.

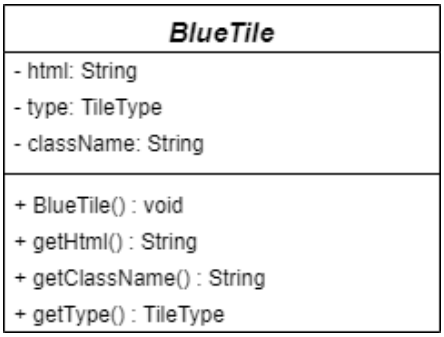

Figure 16: BlueTile Class diagram

2.2.10 GreenTile

This is a class in the map package and is used to represent a green tile. In fact, it extends the Tile class. This class contains two other member variables named type and className to hold the tile type and the CSS class name respectively. Moreover, it overrides all the methods found in the Tile class.

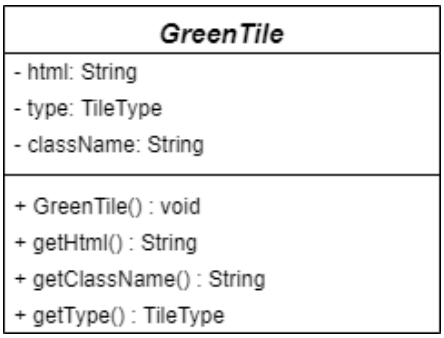

Figure 17: GreenTile Class diagram

2.2.11 GreenTile

This is a class in the map package and is used to represent a green tile. In fact, it extends the Tile class. This class contains two other member variables named type and className to hold the tile type and the CSS class name respectively. Moreover, it overrides all the methods found in the Tile class. It is important to note that this class reinstates the HTML method since a treasure class has a different HTML structure from a blue or green tile.

2.2.12 Map

This is a singleton class in the map package and is used to represent a map. It was decided to implement it as a singleton class since only one map should be available in a game as this would help in memory management. The class has four member variables used to hold the map for the singleton approach, the size of the map, the tiles which make up the map and a variable which holds the probability of finding a blue. These following is a list of methods found in this class:

1. **getMap()**: This is a method used to return the one instance of the map.

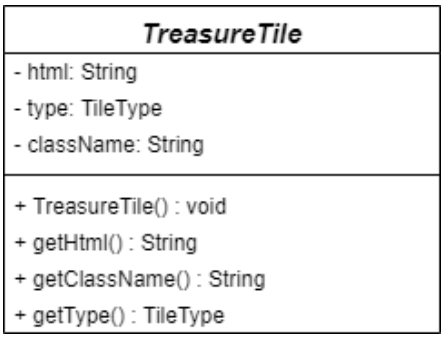

Figure 18: GreenTile Class diagram

2. **setSize(int, Random)**: This is a method used to set the size and generate the tiles and their positions using the random number generator passed as parameter by calling generate().
3. **getMapTiles()**: This is a getter method for the map tiles.
4. **getMapTile(Position)**: This is a method used to get the map tile at the passed position.
5. **getRandomCoordinates(Random)**: This is a method used to get a pair of random coordinates using the random number generator passed.
6. **generate(Random)**: This is a method used generate the tiles and their positions using the random number generator passed as parameter.
7. **goodPositionForTreasure(Tile[][], int, int)**: This is a method used to check whether the coordinates passed (second and third parameters) can be chosen as a good position for a treasure tiles by being checked against the first parameter, which holds the map tiles.
8. **reset()**: This is a method used to reset the map and it is used in testing.
9. **goodPath(Tile[][], int, int)**: This is a method used to check whether the coordinates passed (second and third parameters) can be chosen as a good starting position by being checked against the first parameter, which holds the map tiles.
10. **goodPath(Tile[][], int, int, ArrayList<Tile>)**: This is a method used by the method above to be able to check if a a set of coordinates can reach a treasure tile recursively.

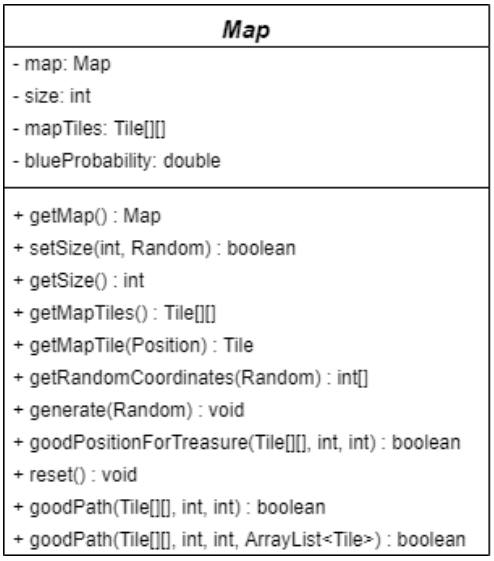

Figure 19: Map Class diagram

2.2.13 MenuHelper

This is a helper class in the menu package and is used to facilitate menu input for a user choice. As can be seen in the class diagram below, this class only contains one method which is used to keep asking for an input until the user finally enters a value in the correct format, that is, only a number. This is done by passing a scanner to read input, passing a message which is outputted before the first time is prompted for an input and by passing a third parameter which is a string outputted to the user in case of entering a value in an incorrect format.

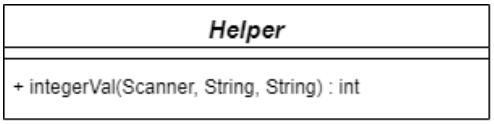

Figure 20: menu.Helper Class diagram

2.2.14 MenuValidator

This is a class in the menu package and is used to validate user input in the menu.
These following is a list of methods found in this class:

1. **amtPlayersValidator(int)**: This is a method used to validate the number of players to play. It makes sure that the players to participate is from 2 to 8.

2. **mapSize(int, int)**: This is a method used to validate the map size input. The first parameter is the amount of players and the second paramter shows the map size. It checks that if the players amount is between smaller than 5, the map size is between 5 and 50 while if the players amount is greater or equal to 5, the map size is between 8 and 50.

3. **directionCheck(int)**: This is a method used to validate the direction input for the player movement and if checks if the input ranges from 1 to 4.

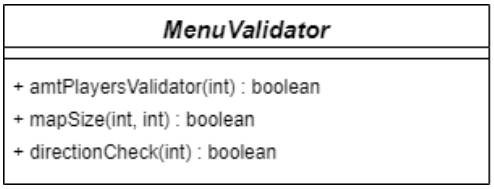

Figure 21: MenuValidator Class diagram

2.2.15 Position

This is a class found in the player package and is used to represent a position. This contains two member variables to hold the x and y coordinates, a constructor, getters and setters for the coordinates and an overridden equals() method to compare two positions.

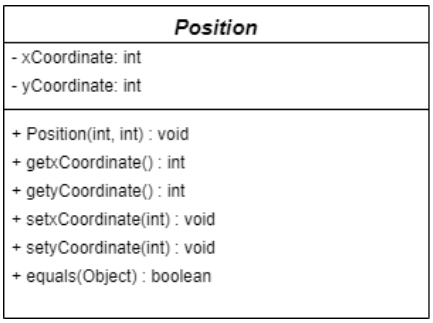

Figure 22: Position Class diagram

2.2.16 Player

This is a class found in the player package and is used to represent a player.

This contains the following member variables:

1. **position** : This is a variable of type Position and it holds the current position of the player.
2. **start** : This is a variable of type Position and it holds the starting position of the player.
3. **tilesVisited** : This is a variable of type Set<Tile>and it holds all the tiles visited by the player.
4. **moves** : This is a variable of type ArrayList<Direction>and it holds all the moves made by the player.

These following is a list of methods found in this class:

1. **Player(Random)**: This is a constructor for the player with the random number generator parameter used to generate the starting position of the player.
2. **setPosition(Position)**: This is a method used to set the position of the player
3. **generateStarting(Random)**: This is a method used to generate the starting position of the player with the random number generator being used to generate coordinates.
4. **getTilesVisited()**: This is a method used to get the tiles visited by the player.
5. **setStart(Position)**: This is a method used to set the starting position of the player
6. **addVisitedTile(Position)**: This is a method used add the tile at the passed position to the set of visited tiles.

7. **getMoves()**: This is a method used to get the moves performed by the player.
8. **move(Direction)**: This is a method used to move the player in the passed Direction. This returns false if the move cannot be performed, in case of a movement leading the player outside the map.
9. **getPosition()**: This is a getter for the current position of the player.
10. **getStart()**: This is a getter for the starting position of the player.

Figure 23: Player Class diagram

2.2.17 Game

This is a static class found in the game package and is used to execute and run the game.

This contains the following member variables:

1. **map** : This is a variable of type Map and it holds the only instance of the Map class in the program.
2. **random** : This is a variable of type Random and it holds the random number generator used to generate numbers for maps and starting positions.
3. **players** : This is a variable of type Player[] and it holds all the tiles visited by the player.
4. **winners** : This is a variable of type ArrayList<Player>and it holds all the winners.
5. **mapResultBuilder** : This is a variable of type MapResultBuilder and it holds the builder for the result pages.
6. **director** : This is a variable of type Director and it holds the director used to construct the HTML pages.

These following is a list of methods found in this class:

1. **setNumPlayers(int)**: This is a method used to set the number of players in the game.
2. **getPlayers()**: This is a getter method for the players.
3. **setWinner(Player)**: This is a method to set a winner.
4. **generateHTMLfiles()**: This is a method used to generate the HTML files for the players.

5. **isAWinner(Player)**: This is a method used to check if the passed player is a winner.
6. **getMap()**: This is a getter method for the map.
7. **setDirector(Director)**: This is a method used to set the director.
8. **setRandom(Random)**: This is a method used to set the random number generator.
9. **reset()**: This is a method used to reset the game variables for testing purposes.
10. **main(String[])**: This is the main method of the class and is the method used to run and execute the game.

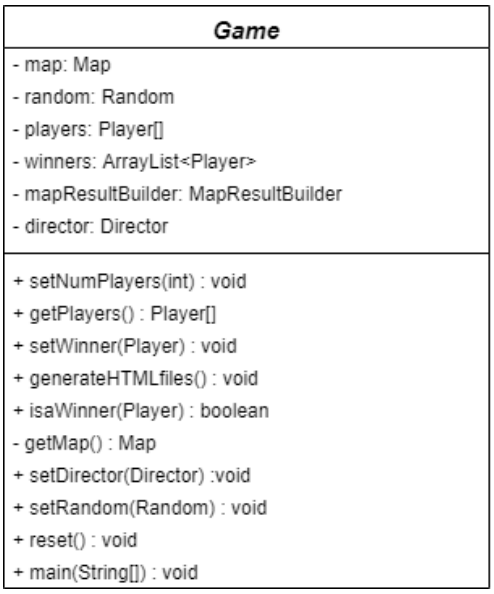

Figure 24: Game Class diagram

2.2.18 UML Class Diagram

In the figure below, one can see how the aforementioned classes are linked together to produce the working implementation of the game.

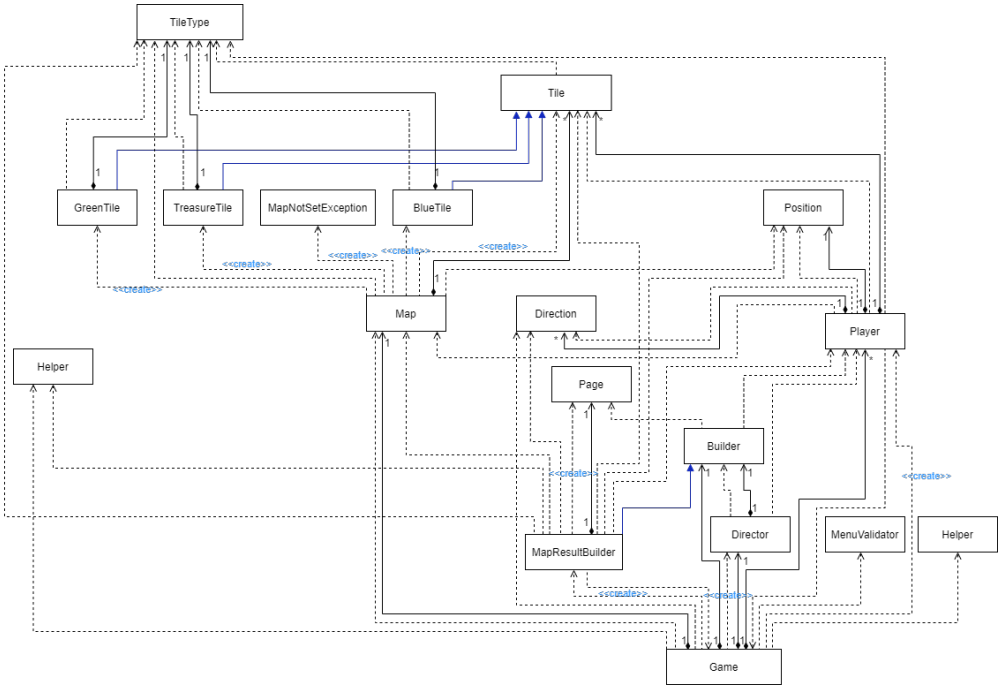

Figure 25: UML Class diagram

2.3 Gameplay

In this section, there will be an overview of the game user interface as it is being played. It is worth mentioning that each player contains his progress HTML file and that the **game was developed in a way such that users cannot see or 'hack' the treasure tile from developer or debugger tools such as inspect element**.

2.3.1 Menu

This section highlights the CLI game menu from where players play their moves and set up the game to be played. As can be seen in the figure below, the game starts by asking the user how many players will be playing and for the map size. After this, the game starts immediately by showing the player in the current turn and which options he can choose to move.

```
Enter amount of players [2-8]
2
Enter length of map size
5

Player 1
Enter the next direction
1. UP
2. DOWN
3. LEFT
4. RIGHT
```

Figure 26: Game CLI menu

2.3.2 Moving out of the map

This section shows a graphical view of what happens when a player tries to move out of the map. As the image below clearly shows, a player is re-asked until a valid direction is entered.

```
Player 2
Enter the next direction
1. UP
2. DOWN
3. LEFT
4. RIGHT
3
You are on the edge, you cannot move left
Enter the next direction
1. UP
2. DOWN
3. LEFT
4. RIGHT
3
You are on the edge, you cannot move left
Enter the next direction
1. UP
2. DOWN
3. LEFT
4. RIGHT
4
```

Figure 27: Moving out of map

2.3.3 Moves

This section shows how the players' moves are recorded and shown in their respective views.

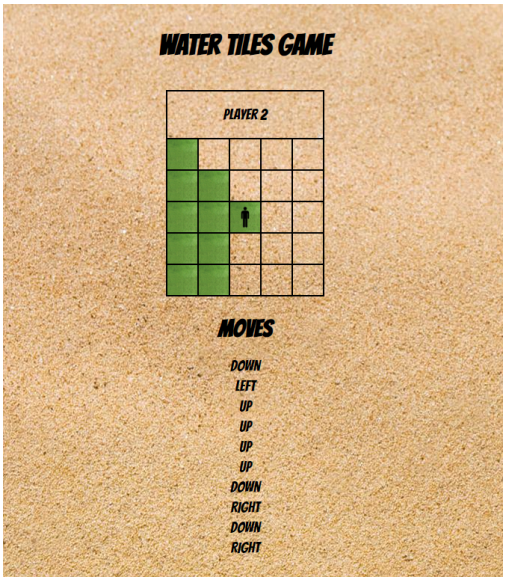

Figure 28: Recorded moves in players' screens

2.3.4 Winning

This section shows what happens when a user wins the match. The first diagram shows how the CLI advises the players that there is a winner while the second figure shows what the winning player sees in his map result.

```
Player 1
Enter the next direction
1. UP
2. DOWN
3. LEFT
4. RIGHT
3

Player 2
Enter the next direction
1. UP
2. DOWN
3. LEFT
4. RIGHT
4
Player 1 is a winner!
```

Figure 29: Winner shoutout in CLI

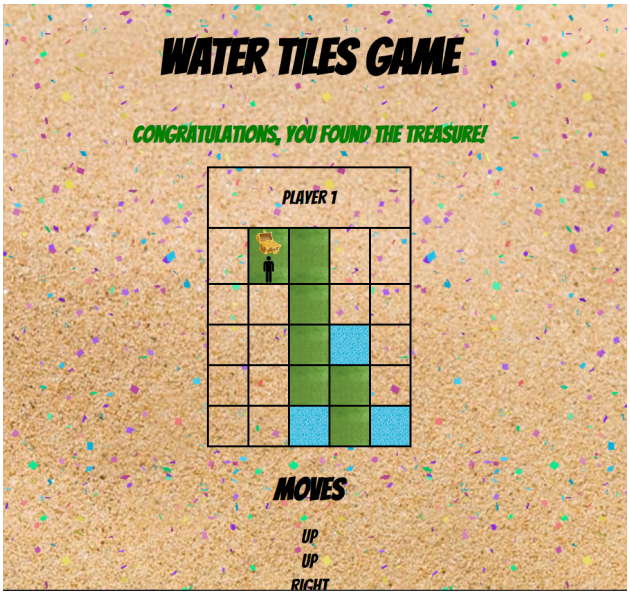

Figure 30: Winner's screen

3 Task 2: Game Improvements

For this part of the assignment, we were required to adjust our game in order to accept the following three changes.

1. Different map types
2. Store one map in memory
3. Team Exploration

After the game was implemented, a tag had to be created to be able to pinpoint the end of this part of development for this game. The tag can be found in the following link: `https://github.com/jacquesvcschool/CPS2002/releases/tag/v2.0`.

3.1 Code Coverage Metrics

One of the requirements at the end of the implementation of part 2 of this assignment was to take note and record the code coverage metrics at this point. As can be seen in the image below, all the code was covered by testing classes, however the same classes omitted in part 1 were also omitted in this part. A full report of the coverage can be found in the HTML file called index in the path '\target\site\jacoco'. Moreover, if one would want to regenerate the metrics, this can be done by running **mvn clean install** in the source directory.

Assignment

Assignment

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
map		100%		100%	0	74	0	125	0	31	0	7
team.player		100%		100%	0	37	0	97	0	23	0	4
files		100%		100%	0	23	0	68	0	13	0	4
menu		100%		100%	0	26	0	29	0	8	0	2
map.tile		100%		n/a	0	13	0	27	0	13	0	5
team		100%		100%	0	14	0	25	0	10	0	2
game		100%		n/a	0	1	0	3	0	1	0	1
exceptions		100%		n/a	0	1	0	2	0	1	0	1
Total	0 of 1,635	100%	0 of 175	100%	0	189	0	376	0	100	0	26

Figure 31: Code coverage metrics for part 2.

The following is similar to the explanation in part 1 showing which two classes are omitted from the testing coverage count and why.

1. **Game.java**: This is the class which contains the methods used int the **main()** method, also found in this class, which is used to execute the game. Despite not being in the code coverage count, one can still find a GameTest test class. This contains tests which test the main method and some of the class' methods. In fact, there is also a simulation of the game. However, It was omitted from the coverage class because since some of our methods, for instance, the main method, are static, PowerMockito had to be used and Jacoco's code coverage does not count these tests and hence, was excluded from the test classes to count just for the coverage numbers to look better.
2. **Helper.java**: This is a class which contains methods to copy, read and write from and to files. Since these are general methods used and tested by the programming community, tests for this class were not implemented. For this reason, this class was omitted from the tests to check for coverage so that the coverage numbers actually show what really needed to be tested.

3.2 Solution to Enhancements

This section goes through and explains all the changes needed to change the system to meet the new requirements. In each section, an account of which classes changed, which classes were created and how these classes were made to interact with each other is given.

3.2.1 Different Map Types

This enhancement entailed that we make the game support different map types. A **Safe** map had to be created to be able to contain 10% blue tiles while a **Hazardous** map had to be created to contain between 25% and 35% blue tiles. Moreover, the system had to be implemented in such way to be able to create new map types in the future.

After reading about design patterns, we came to the solution that the design pattern to implement to fix this problem is the **Factory Design Pattern** since it makes construction of similar but different objects easier. Moreover, this pattern fits our problem since we need to create new objects in the future but we do not know their types as of now and since we want to localise the logic to create a complex object.

The first step involved changing our Map class to an abstract class which can be extended by other classes. From this change the Map class resulted in a class with the following class diagram. Moreover, some member variables were removed as they were implemented directly in the new classes extending it which will be discussed further below

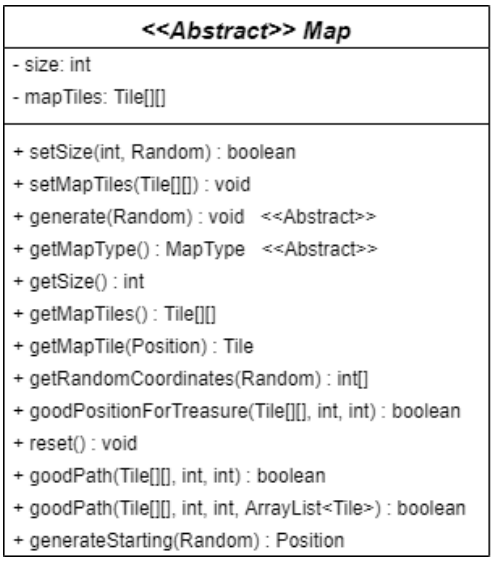

Figure 32: New Map Class diagram

The following are the added classes in order to be able to implement the Factory design pattern.

1. **MapType**

   This is an enumeration found in the map package and is used to hold the two types of maps, namely, SAFE and HAZARDOUS.

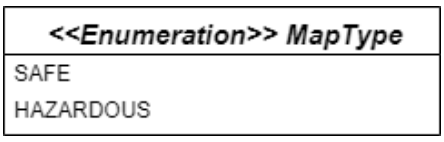

Figure 33: MapType Class diagram

2. **SafeMap**

   This is a class found in the map package which extends the abstract Map class and represents a safe map. It is exactly the same like the old map class however the probability of finding blue tiles is 10% and the constructor and getMap() methods were created to implement the singleton method for the next enhancement.

Figure 34: SafeMap Class diagram

3. **HazardousMap**

   This is a class found in the map package which extends the abstract Map class and represents a hazardous map. It is exactly the same like the old map class however the probability of finding blue tiles is 35% and the constructor and getMap() methods were created to implement the singleton method for the next enhancement.

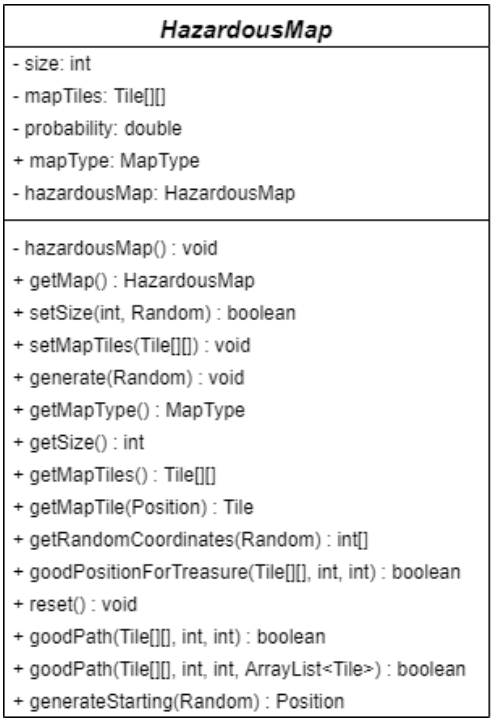

Figure 35: HazardousMap Class diagram

4. **MapFactory**

   This is an abstract class found in the map package and represents a map factory. This class contains two methods, one called getMap(MapType) which returns a map of the type passed and an abstract method named createMap() which creates a map. This can be seen in both the class diagram and code snippet below.

```java
/**
 * Class for Map factory
 */
public abstract class MapFactory {

    /**
     * Factory method
     * @param mapType type of map
     * @return a map
     */
    public static Map getMap(MapType mapType)
    {

        MapFactory creator;

        //get type of creator needed
        switch (mapType)
        {
            case SAFE: creator = new SafeMapFactory();break;
            default: creator = new HazardousMapFactory();break;
        }

        //create and return map
        return creator.createMap();
    }

    /**
     * Abstract method to create the map
     * @return new map
     */
    public abstract Map createMap();

}
```

Figure 36: MapFactory Code snippet

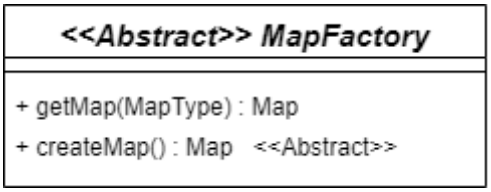

Figure 37: MapFactory Class diagram

5. **SafeMapFactory**

   This is a class found in the map package which extends the MapFactory class and represents a safe map factory. This class contains two methods, one called getMap(MapType) which returns a map of the type passed and a method named createMap() which creates and returns a safe map. This can be seen in both the class diagram and code snippet below.

```java
/**
 * Class for Safe Map creator
 */
public class SafeMapFactory extends MapFactory {

    @Override
    public Map createMap() { return SafeMap.getMap(); }

}
```

Figure 38: SafeMapFactory Code snippet

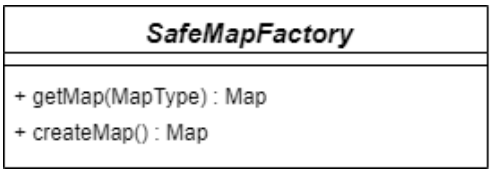

Figure 39: SafeMapFactory Class diagram

6. **HazardousMapFactory**

   This is a class found in the map package which extends the MapFactory class and represents a hazardous map factory. This class contains two methods, one called getMap(MapType) which returns a map of the type passed and a method named createMap() which creates and returns a hazardous map. This can be seen in both the class diagram and code snippet below.

```java
/**
 * Factory class for hazardous map
 */
public class HazardousMapFactory extends MapFactory {

    /**
     * Method to return a hazardous map
     * @return hazardous map
     */
    @Override
    public Map createMap() { return HazardousMap.getMap(); }

}
```

Figure 40: HazardousMapFactory Code snippet

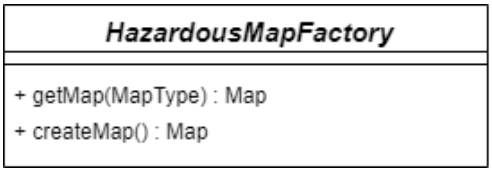

Figure 41: HazardousMapFactory Class diagram

The below figure shows how the aforementioned classes integrate and communicate with each other to create a Factory design pattern.

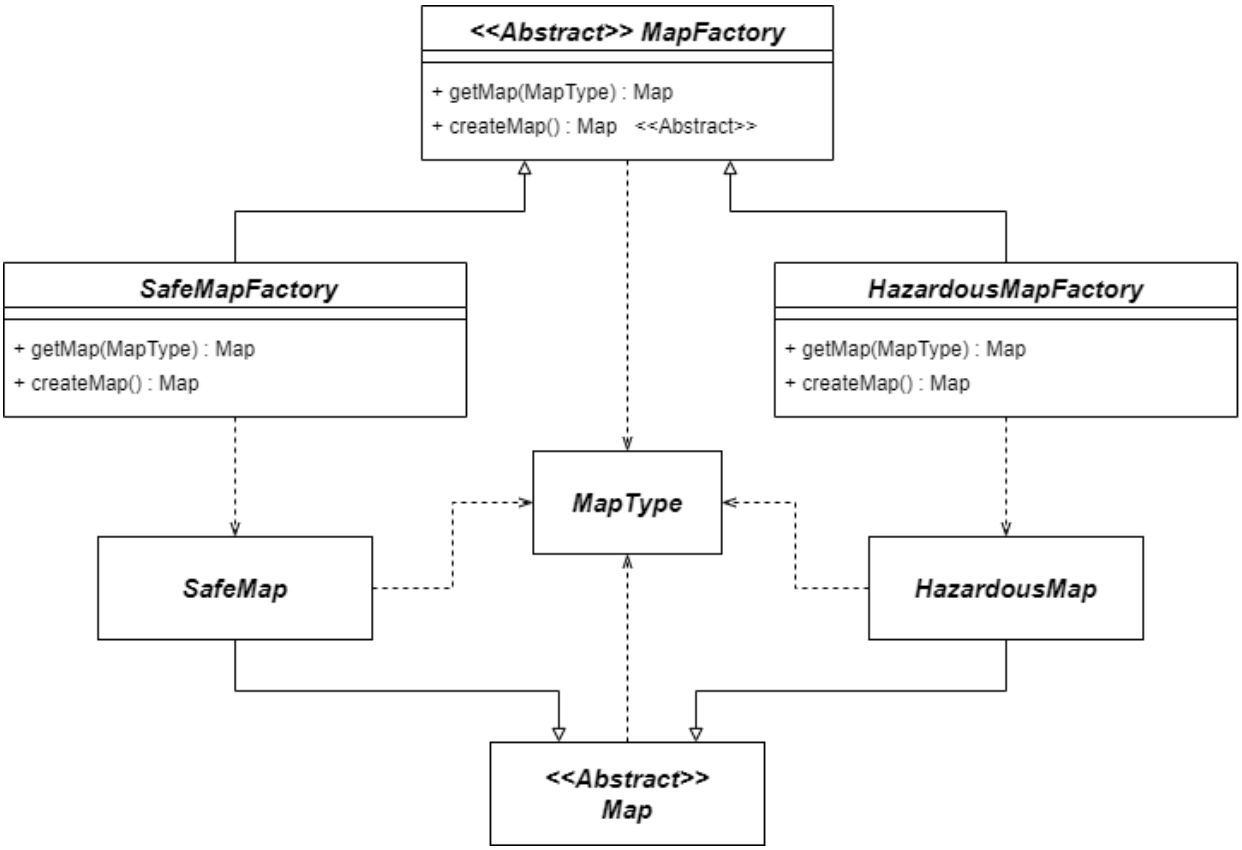

Figure 42: UML Class Diagram for Enhancement 1 - Factory Design Pattern

3.2.2 Store one map in memory

The second enhancement involved the requirement to only keep one map in memory at all times and the design pattern used to fulfill this requirement is the **Singleton** design pattern. The classes SafeMap and HazardousMap were implemented as singleton classes so that only one instance of them may exist in memory. As can be seen in the figure below, this was achieved by making the constructor for these classes private, hence, only accessible from the class itself and by adding a static getMap() method which returns the only instance of that class. This can be seen in both the class diagram and code snippet below.

```java
//for singleton
private static SafeMap safeMap = new SafeMap();

/**
 * private constructor
 */
private SafeMap(){}

/**
 * method to return map instance
 */
public static SafeMap getMap() { return safeMap; }
```

Figure 43: SafeMap code snippet for Enhancement 2 - Singleton Design Pattern

```java
//for singleton
private static HazardousMap hazardousMap = new HazardousMap();

/**
 * private constructor
 */
private HazardousMap(){}

/**
 * method to return map instance
 */
public static HazardousMap getMap() { return hazardousMap; }
```

Figure 44: HazardousMap code snippet for Enhancement 2

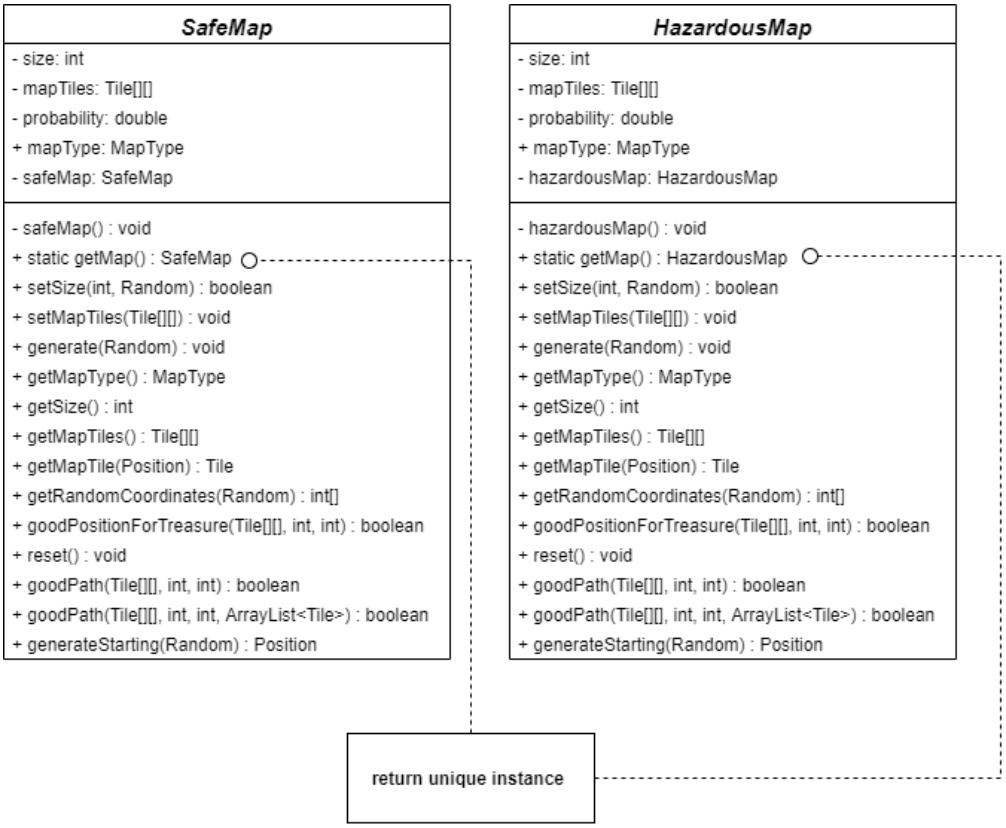

Figure 45: UML Class Diagram for Enhancement 2

3.2.3 Team Exploration

The final enhancement involved the requirement to implement a collaborative game mode apart from the normal solo mode. This means that now players should be able to play as a team, meaning team members move together. Therefore, since there is the need for a subject to update other objects to follow the subject's state, we decided to implement the **Observer** design pattern in order to be able to implement this enhancement. The idea is to have the team as the subject with the next direction being the state of the team while the players are the observers. Every time the state changes, meaning some team member played a move, all the observers are notified and they each get the recently changed state and each player moves in that direction.

In order to be able to make or implementation of the game support this enhancement, we had to make the following changes to the already existing classes.

1. **Builder and MapResultBuilder**: The abstract and concrete builder classes were changed slightly by changing the buildTitle() method to accept a player rather an index so that it could be checked if it is associated with a team and if so, the team's id is also printed in the HTML result file.

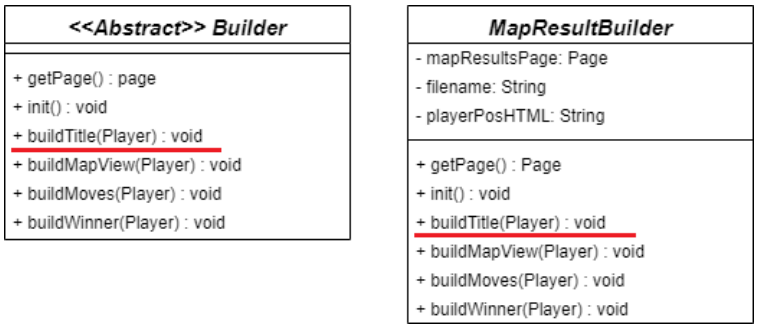

Figure 46: Changes to classes Builder and MapResultBuilder

2. **Map**: The introduction of teams means that a team should be able to generate a starting position. Therefore, for the design to make more sense, the genereateStarting() method previously found in the Player class was moved to the Map class.

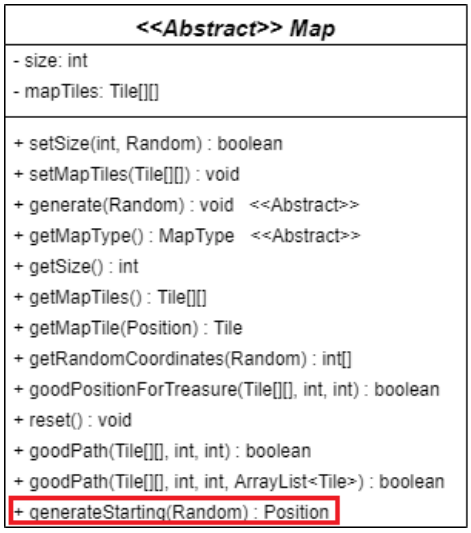

Figure 47: Addition to Map class

3. **Player**: One of the main changes to be able to implement the observer design pattern for this en-

hancement was the Player class. First of all, it was changed to extend the Observer class which will be explained further below. Moreover, two new member variables, id and team, were added together with their getter methods to give an id to each player and the possibility for a player to be attached to a team. In addition, this led to the creation to a new constructor without the Random class as parameter for the case of teams as the player would not need to call the method to generate a starting position. In fact, a new setup() method was created which is called when a player is attached to a team so that the player set to the team's starting position and the team passed as parameter is stored in the new member variable called team. One may also note the removal of the generateStarting() method which as explained above, was moved to the Map class.

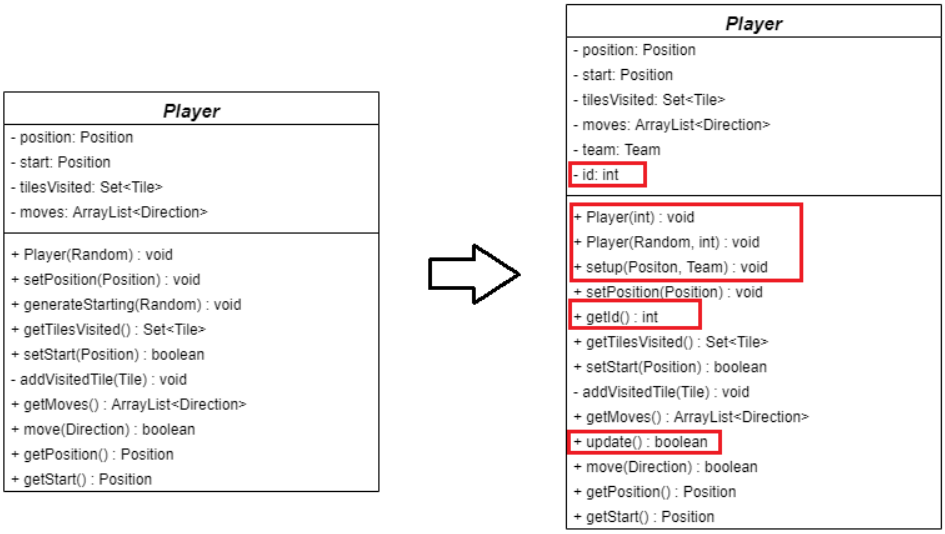

Figure 48: Changes to Player class

4. **Game**: Another class which had to be obviously changed is the Game class which runs and executes the game. Despite being in this section, this class also had to be changed to be able to implement the changes made for the first enhancement. Three member variables had to be added, teams, winnersTeams and gameMode which hold the teams in the games, the winning teams and the mode of the game chosen by the user.

   Moreover, as can be seen marked in red in the figure below, the following new methods had to be implemented:

   (a) **setWinningTeam(Team)**: This is a method used to add a team to the list of winning teams.

   (b) **setPlayerToTeam(Team, Player)**: This is a method used to add a player to a team.

   (c) **initTeams(int)**: This is a method used to initiate an amount of teams passed as parameter.

   (d) **printTeams()**: This is a method used to print the teams and its players before the match so players would know which team they make part of.

   (e) **getGameMode(int)**: This is used to get the game mode related with the input from the user which is passed as a parameter to the function.

   (f) **getMapType(int)**: Similarly, This is used to get the map type related with the input from the user which is passed as a parameter to the function.

   (g) **setMap(Map)**:This a setter for the map.

   (h) **printWinners()**:This a method used in the main class to print the winners.

(i) **playRounds(Scanner)**:This a method used to play the rounds. This was created to divide the code and make the code cleaner in the main function.

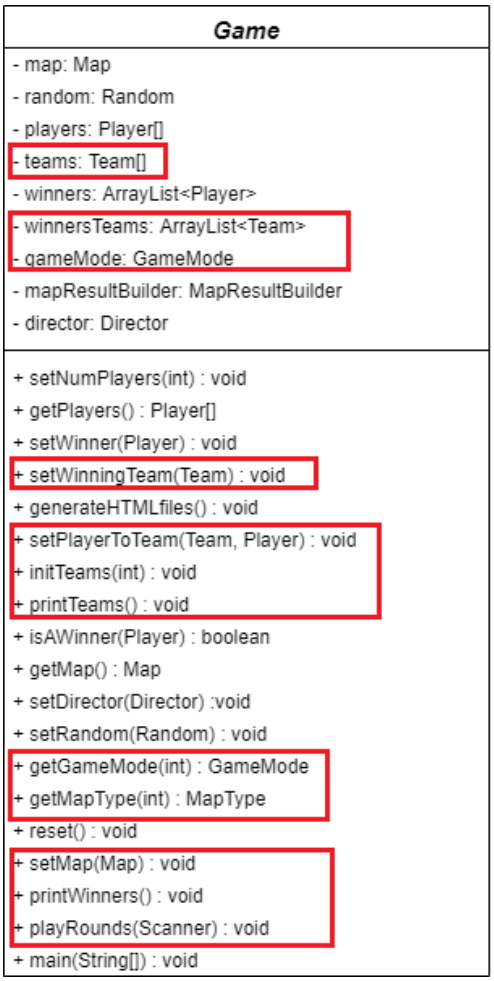

Figure 49: Changes to Game class

In addition, the following new classes had to be created:

1. **GameMode** This is an enumeration found in the game package and is used to hold the two types of game modes, namely, SOLO and COLLABORATIVE.

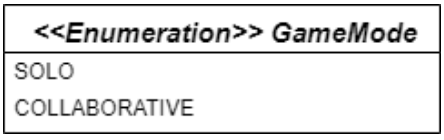

Figure 50: GameMode Class diagram

2. **GameMode** This is an enumeration found in the game package and is used to hold the two types of game modes, namely, SOLO and COLLABORATIVE.

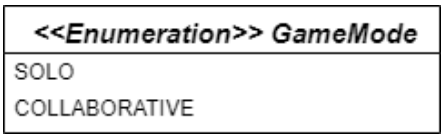

Figure 51: GameMode Class diagram

3. **Observer** This is an abstract class found in the player package and is extended by the Player class. This contains one member variable to hold the team, which is the subject, and an update() method used to update the state of the observer with the state from the subject.

```java
/**
 * Observer class
 */
public abstract class Observer {
    //subject
    protected Team team;

    /**
     * Method used to update all observers
     * @return if the update was successful
     */
    public abstract boolean update();
}
```

Figure 52: Observer Class code

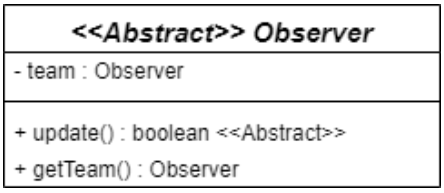

Figure 53: Observer Class diagram

4. **Subject** This is an abstract class found in the team package which represents the subject in the observer design pattern implemented. This contains one member variables, observer, which hold a list of players.

   Moreover, this class contains the following methods

   (a) **getState()**: This method gets the state of the team, that is the most recent direction for movement.

   (b) **getObservers()**: This method gets all the observers attached to the subject.

   (c) **notifyAllObservers()**: This method is used to notify all players of the team

```java
/** Method to attach a player to the team ...*/
public void attach(Player observer) { players.add(observer); }

/** Method to notify all players and update them ...*/
public boolean notifyAllPlayers(){
    for (Player player : players) {
        //if the state is not correct, return false immediately
        if(!player.update())
            return false;
    }
    return true;
}
```

Figure 54: Subject Class code

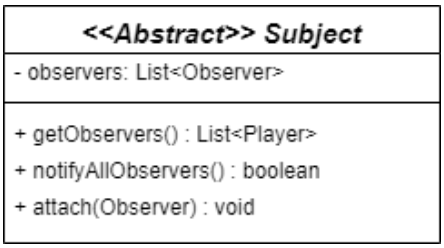

Figure 55: Subject Class diagram

5. **Team** This is a class found in the team package which extends the Subject class and it represents a team. Moreover, this class contains four member variables named start, nextPlayerTurn, id and state which hold the starting position of the team, the id of the next player who is next in turn and the id of the team respectively. The state of the subject represents the recent direction for the team movement, so if a player from the team moves, the direction is set and all the team players are notified to move in that direction.

   Moreover, this class contains the following methods

   (a) **Team(Random, int)**: This is a constructor to create a team with random number generator used to generate the team's starting position and the id to be set.

   (b) **getNextPlayerTurn()**: This method gets the id of the player who is next to play.

   (c) **getStart()**: This is a getter for the starting position of the team.

   (d) **getId()**: This is a getter for the id of the team.

   (e) **setState(Direction**: This is a setter for the direction.

   (f) **getState()**: This method gets the state of the team, that is the most recent direction for movement.

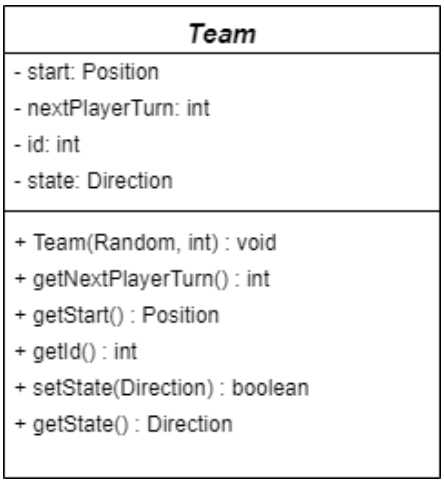

Figure 56: Team Class diagram

The below figure shows how the aforementioned classes integrate and communicate with each other to create a Factory design pattern.

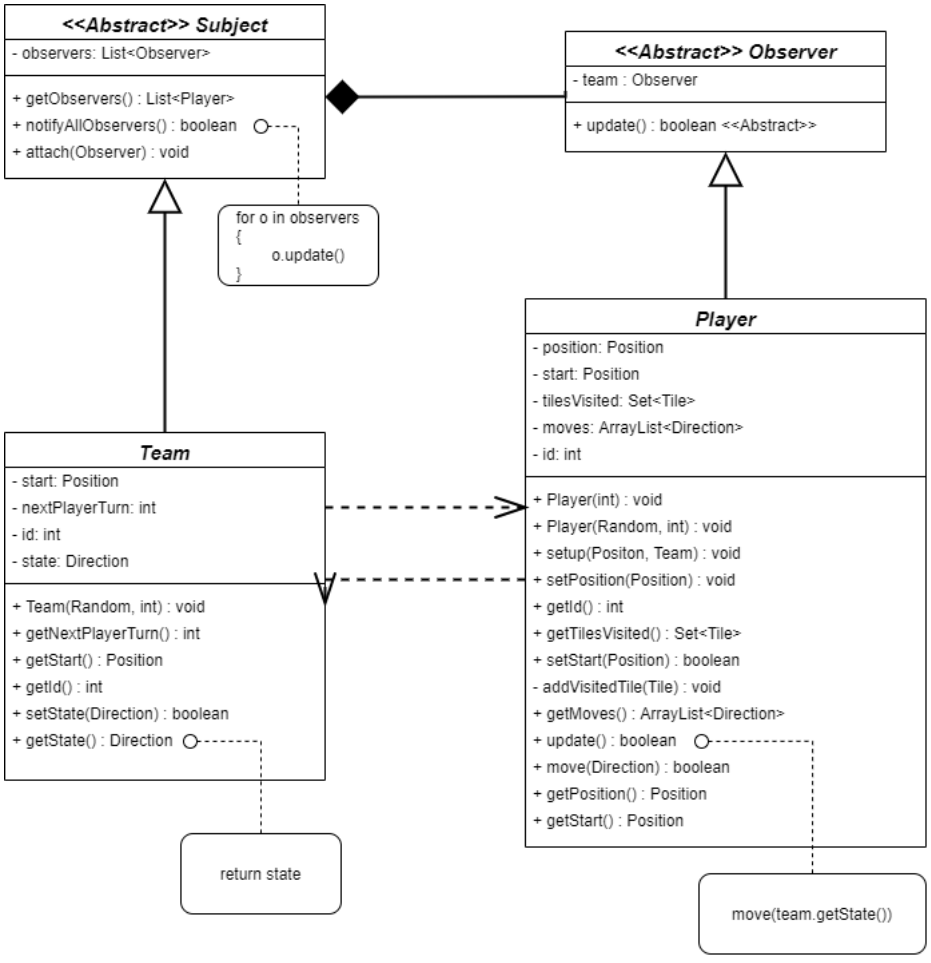

Figure 57: UML Class Diagram for Enhancement 3 - Observer Design Pattern

3.3 Gameplay

This section shows a graphical view of the new enhancements and how these change the look of the game. For the second enhancement, to keep one memory in map, not many can be seen and for the first enhancement, to have two types of maps, cannot be seen here very clearly as one map type just contains more blue tiles than the other, however, this can be easily noticed when playing. Below, one can view some the new features in screenshots.

3.3.1 Menu

This section highlights the CLI game menu. As can be seen in the figures below, the user is first asked for the game mode and then for the map type.

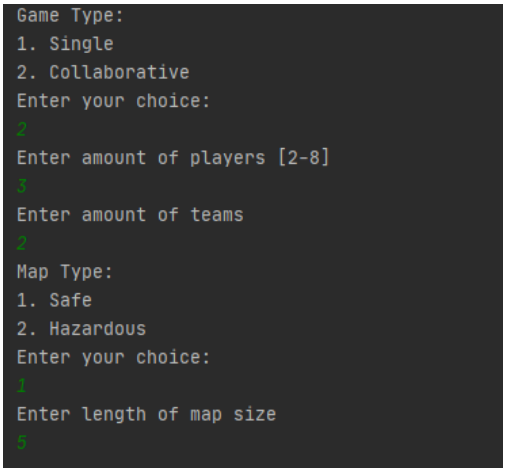

Figure 58: Game CLI menu asking for game mode, amount of teams and map type

3.3.2 Printing teams

This section shows a graphical view of how the game menu prints the random allocation of players to teams before the game starts.

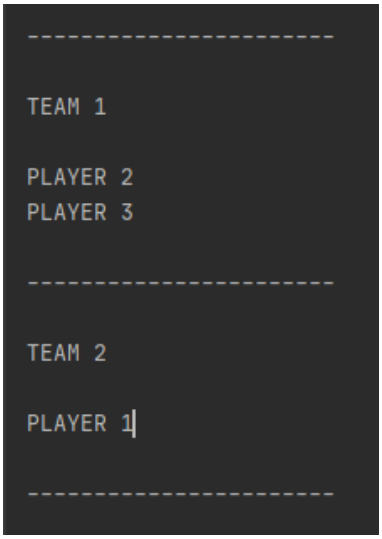

Figure 59: Printing allocation of players to teams

3.3.3 Turns

This section portrays how the the player turn is displayed in the menu.

```
Team 1: Player 2
Enter the next direction
1. UP
2. DOWN
3. LEFT
4. RIGHT
```

Figure 60: Player turn indication

3.3.4 HTML file view

This section shows how the HTML looks like in a collaborative game.

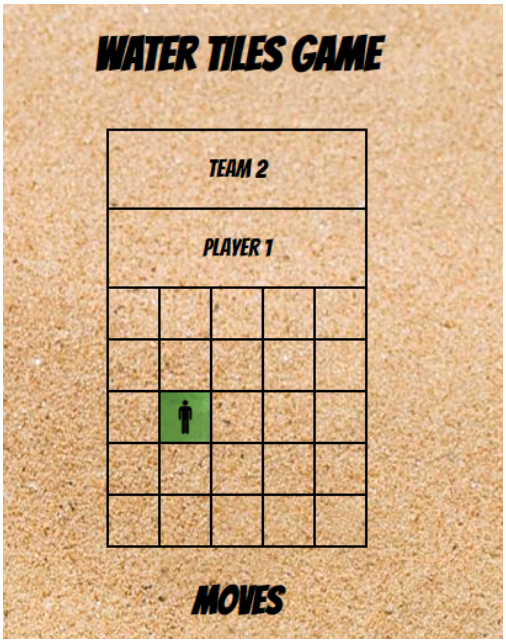

Figure 61: Player screen in collaborative mode

3.3.5 Winning

This section shows what happens when a user wins the match and is part of a team. The first diagram shows how the CLI advises the players that there is a winning team and shows all its players while the second figure shows what the winning player sees in his map result.

```
================================

Team 2 is a winner!
Player 1 is a winner!

================================
```

Figure 62: Winning team shoutout in CLI

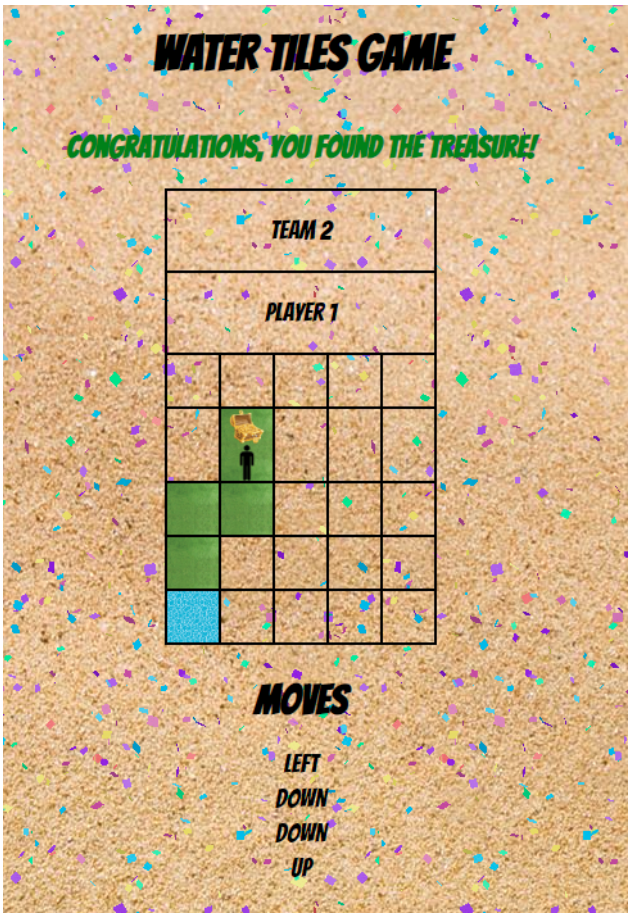

Figure 63: Winner's screen

4 How to run

To run the game, one can go to the root folder of the project and run the following command: **java -jar WaterTiles.jar**. Another option is to follow these steps:

1. Run **mvn clean install**
2. Go to target folder
3. Run **java -jar Assignment-2.0-jar-with-dependencies.jar**

5 Conclusion and evaluation of work

To conclude, we feel that we created a functioning and robust game which can be confirmed from the 100% test coverage obtained and we also feel that the game was implemented in a way which allows further enhancements in the future. However, just like any other thing, we feel that we could have done some things better. If we had more time on our hands, it would have been nice to make the menu look nicer in terms of menu user interface.

References

[1] "Design pattern - factory pattern." https://www.tutorialspoint.com/design_pattern/factory_pattern.htm.

[2] "Builder design pattern." https://www.geeksforgeeks.org/builder-design-pattern/.

[3] "Design patterns - observer pattern." https://www.tutorialspoint.com/design_pattern/observer_pattern.htm.

[4] "Design pattern - singleton pattern." https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm.

[5] "Mocking static methods with mockito." https://stackoverflow.com/questions/21105403/mocking-static-methods-with-mockito/.

[6] "git-flow cheatsheet." https://danielkummer.github.io/git-flow-cheatsheet/.

[7] "Find whether there is path between two cells in matrix." https://www.geeksforgeeks.org/find-whether-path-two-cells-matrix/.