

**Software Engineering**

**CPS2002**

**Assignment Report**

Martin Bartolo - 0218300L

Mikhail Cassar - 0319599M

BSc (Hons) Computing Science and Mathematics

Assignment due 27<sup>th</sup> May 2019

# Contents

<b>Diagram Example</b>	<b>3</b>
<b>Code Snippet</b>	<b>3</b>
<b>Introduction</b>	<b>3</b>
<b>Game Design</b>	<b>4</b>
<b>Enhancements</b>	<b>4</b>
Different Map Types . . . . .	4
<b>Enhancements</b>	<b>4</b>
Different Map Types . . . . .	4
Class Diagram . . . . .	12
<b>Game Enhancements</b>	<b>12</b>
<b>Code Coverage Statistics</b>	<b>12</b>
<b>Running The Game</b>	<b>12</b>

## Diagram Example

```
~~~Machine Learning 1 Assignment~~~
~~~Created by Martin Bartolo~~~
Enter 1 to see the answer for Question 1, 2 for the answer to Question 2 and 3 for the answer to Question 3
1
How many features would you like? (Please enter a number between 1 and 3)
3
Enter 1 for Sepal Length, 2 for Sepal Width, 3 for Petal Length, 4 for Petal Width
1
2
3
|
```

## Code Snippet

```
public class Main{
    public static void main (String args[]){
        System.out.println("Hello World!");
    }
}
```

## Introduction

The aim of this assignment was to collaboratively work on a software project, with the main focus being on rigorous software testing and the use of Git. Our first task was to set up our environments, namely our Git repository on Github and our Jenkins environment on the University Jenkins server. First, we initialised our Git repository and ensured that each team member could commit changes and push and pull them from Github. When this was ensured, we set up our Jenkins environment to work with Maven and scan for changes from Github every few minutes. Whenever changes are found, they are built and run with a detailed code coverage report and test results being displayed using the Emma plug-in. Our progress at this point can be seen by viewing the "Part1" tag on our Github repository. After completing our set-up, we were ready to start working on the two remaining tasks which will be discussed in detail throughout the remainder of this report.

# Game Design

The design of the game consists of sectioning the game into the classes, here we will talk about the contents of those classes while also explain how they all work together.

The information provided will be regarding the basic version of the game i.e. before any enhancements to the game were made. The basic version of the game can be played by going to the tag "Part2".

## The Initial Game Design

This is the code for the basic version of the game, thus this is the game before any enhancements were made.

### Game Class

The Game class contains the main game loop and most of the logic regarding the game. The most important functions of the Game class include initialising the tiles and setting up the players, compute the directions of each player, generate the HTML map files for each player (This changes in the future generations of the game) and also exits the program.

Listing 1: The initial code of the Game class

```
import java.io.*;
import java.util.ArrayList;
import java.util.InputMismatchException;
import java.util.Scanner;

public class Game {

    // Amount of turns which have been played
    private int turns = 0;
    // Amount of players
    int playerNum;

    // ArrayList of players
    ArrayList<Player> players = new ArrayList<Player>();

    Map map = new Map();// map object

    final private int minPlayers = 2;
    final private int maxPlayersFirstRange = 4;
    final private int minPlayersSecondRange = 5;
    final private int maxPlayers = 8;

    final private int minMapSizeFirstRange = 5;
    final private int minMapSizeSecondRange = 8;
    final private int maxMapSize = 50;
```

```

private Scanner scanner;// to be used throughout class

//Constructor for the game object
Game() {
}

public static void main(String[] args) {
    //This variable is used to hold the
    //previous directions taken by a given player
    String directions;

    System.out.print("Welcome to the Treasure Map Game by Martin")
    Game game = new Game();

    //Run startGame method to initialise players and map
    game.startGame(game);

    // will be set to true when the treasure
    //is found by one of the players
    boolean foundTreasure = false;

    //An array which holds all the players
    //who found the treasure on a given turn
    //This is just in case more than one
    //player finds it on the same turn
    boolean[] winners = new boolean[game.players.size()];

    //Generating the initial html files here before
    //there are any moves
    //Generating an html file for each player in the game
    for(int i = 0; i < game.players.size(); i++){

        if(game.generateHtmlFile(i, game.map.mapSize, " ") == 0){
            System.err.println("Could not generate HTML files");
        }
    }

    //Main game loop
    while (true) {

        //Increment amount of turns
        //which have been played
        game.turns++;

        System.out.println("-----");

        //Get each players desired direction
        //of movement for the current turn
        game.directionsLoop();

        //Generating an html file for each
        //player's current state
        for(int i = 0; i < game.players.size(); i++){

```

```

        //Obtaining the last 4 directions of each player
        directions = game.getPreviousDirections(i);

        if(game.generateHtmlFile(i, game.map.mapSize,directions) == 0){
            System.err.println("Could not generate HTML files");
        }
    }

    //Go through each player in the
    //game and check if they found the treasure
    //Mark the players who have found the treasure
    int i = 0;
    for(Player player: game.players){

        if(player.foundTreasure){
            foundTreasure = true;
            winners[i] = true;
        }
        i++;
    }

    //If the treasure has been found by one of the players
    if (foundTreasure) {

        for(i = 0; i < winners.length; i++){

            if (winners[i]){
                System.out.println("Congratualtions player " + (i+1) + ",
            }
        }
        break;
    }
}

System.out.println("-----\n");

//After a player wins the game the user is able to prompted to exit the g
game.exitGame(game);
}

//Method to initialise map along with players and
// their starting positions
private void startGame(Game game) {
    game.playerNum = getPlayerNum();

    map.mapSize = getMapSize();
    map.generate();// Generate map

    //In this loop all the Player objects are created
    //along with their starting position in the map
    for (int i = 0; i < game.playerNum; i++) {

        //A new player object is made

```

```

        Player player = new Player();

        //The random position of the player is
        //set to a grass tile
        player.position = player.setStartingPosition(map.getGrassTiles());

        //The created player is added to the ArrayList of players
        players.add(player);
    }
}

// Method to get the amount of players from the user
private int getPlayerNum() {
    int num;
    System.out.println("How many players will be playing? (Pick a number between 2 and 8)");

    while (true) {
        //Get user input
        scanner = new Scanner(System.in);

        //Validate user input
        num = validatePlayerNum(scanner);

        //Return value if it is valid (not an error value)
        if (num > 1) {
            return num;
        }
    }
}

// Method to validate the user's input for the player number
int validatePlayerNum(Scanner scanner) {
    int num;
    try {
        //Set to user input from getPlayerNum
        num = scanner.nextInt();
    }
    //If input is not an integer
    catch (InputMismatchException e) {
        System.err.println("Invalid input. Please enter a number between 2 and 8");
        return 0; //Return error value of 0
    }
    //If input is correct
    if (num >= minPlayers && num <= maxPlayers) {
        return num; //Return value entered by the user
    }
    //If input is not within required range
    else {
        System.err.println("Please enter a number between 2 and 8");
        return 1; //Return error value of 1
    }
}

// Method to get the map size from the user

```

```

private int getMapSize() {
    int size;
    System.out.println("How large would you like the map to be? (Map will be

while (true) {
    //Get user input
    scanner = new Scanner(System.in);

    //Validate user input
    size = validateMapSize(scanner);

    //Return value if it is valid (not an error value)
    if (size > 4) {
        return size;
    }
}

}

// Method to validate the user's input for the map size
int validateMapSize(Scanner scanner) {
    int size;
    try {
        //Set to user input from getMapSize
        size = scanner.nextInt();
    }

    //If input is not an integer
    catch (InputMismatchException e) {
        System.err.println("Invalid input. Please enter a number");
        return 0; //Return error value of 0
    }

    //If input is above largest allowed map size
    if (size > maxMapSize) {
        System.err.println("Map too big. Please enter a size below 50");
        return 1; //Return error value of 1
    }

    //If map is too small for 2-4 players
    else if (playerNum <= maxPlayersFirstRange && size <
        minMapSizeFirstRange) {
        System.err.println("Map too small. Please enter a size of 5 or more");
        return 2; //Return error value of 2
    }

    //If map is too small for 5-8 players
    else if (playerNum >= minPlayersSecondRange && size <
        minMapSizeSecondRange) {
        System.err.println("Map too small. Please enter a size of 8 or more");
        return 3; //Return error value of 3
    }

    //If input is correct
    else {

```



```

        return size;//Return value entered by the user
    }
}

// Method to get direction which each player
//would like to move in for the current turn
private void directionsLoop() {
    //(u, d, l or r) depending on user's desired
    //direction of movement
    char direction;
    //Condition to break out of while
    //loop when a valid direction is entered
    boolean validMove;

    //Loop through each player in ArrayList
    for (Player player : players) {
        System.out.println("Player " + (players.indexOf(player) +
            1) + ", please choose a direction (u, d, l or r).");

        validMove = false;
        while (!validMove) {
            direction = 'x';
            //Make sure that user input is
            //valid (i.e. one of u, d, l or r)
            while(direction == 'x' || direction == 'y') {
                scanner = new Scanner(System.in);
                direction = validateDirectionInput(scanner);
            }

            //Check if move is within map and execute if it is
            if (checkOutOfBounds(direction, player, map.mapSize)
                == 1) {
                validMove = true;

                //Change player's position variables
                //to new position
                player.move(direction);

                //Triggers event for corresponding tile type
                map.evaluateCurrentPlayerTile(player);
            }
        }
    }
}

// Method to check whether a move is within the map boundaries
int checkOutOfBounds(char direction, Player player,
    int mapSize) {
    switch (direction) {
        case 'l':
            //If map limit is exceeded
            if (player.position.x - 1 < 0) {

```

```

        System.err.println("Cannot move left. Please
        try another direction.");
        return 0; //Return error value of 0
    }

    //If move is within map
    else {
        System.out.println("Player moved to the left");
        return 1; //Return correct value 1
                //to indicate that move is valid
    }

case 'r':
    //If map limit is exceeded
    if (player.position.x + 1 >= mapSize) {
        System.err.println("Cannot move right.
        Please try another direction.");
        return 0; //Return error value of 0
    }
    //If move is within map
    else {
        System.out.println("Player moved to the right");
        return 1; //Return correct value 1 to
                //indicate that move is valid
    }

case 'u':
    //If map limit is exceeded
    if (player.position.y - 1 < 0) {
        System.err.println("Cannot move up.
        Please try another direction.");
        return 0; //Return error value of 0
    }
    //If move is within map
    else {
        System.out.println("Player moved up");
        return 1; //Return correct value 1 to
                //indicate that move is valid
    }

case 'd':
    //If map limit is exceeded
    if (player.position.y + 1 >= mapSize) {
        System.err.println("Cannot move down. Please
        try another direction.");
        return 0; //Return error value of 0
    }
    //If move is within map
    else {
        System.out.println("Player moved down");
        return 1; //Return correct value 1
                //to indicate that move is valid
    }
}

```

```

        default:
            return 0; //Return error value of 0
    }
}

// Method to check whether an inputted direction is
//valid (i.e. either u, d, l or r)
char validateDirectionInput(Scanner scanner) {
    String direction; //User input
    char c; //User input after it being converted
        //into a character
    String directions = "udlr"; //String containing
        //each accepted direction

    try {
        //Set to user input from getDirections
        direction = scanner.next();

        //Ensure that inputted string is of length 1
        if (direction.length() > 1) {
            throw new RuntimeException("Input too long.
                Please enter a character (u, d, l or r)");
        }

        //Convert input string to char
        c = direction.charAt(0);

        //Ensure that character is a letter
        if (!Character.isLetter(c)) {
            throw new RuntimeException("Input is not a character.
                Please enter a character (u, d, l or r)");
        }
    }

    //If an error is thrown in the try block
    catch (RuntimeException e) {
        System.err.println(e.getMessage());
        return 'y'; //Return an error value which we
            //will associate with an exception
            //when testing
    }

    //Change char input to lowercase to allow U, D, L and R
    c = Character.toLowerCase(c);

    //If input is a char but not one of the directions
    if (!directions.contains(Character.toString(c))) {
        System.err.println("Invalid input. Please enter a
            direction (u, d, l or r)");
        return 'x'; //Return an error value which
            // we will associate with an invalid character when testing
    }

    //If input is correct

```

```

        else {
            return c; //Return valid user inputted character
        }
    }

// This method is used to generate the
//HTML files so that they can be opened in browser
int generateHtmlFile(int playerIndex, int mapSize, String direction) {
    //Value to return to mark if
    //method has run successfully or not
    //Set to 1 by default. This will
    //change to 0 if an error is encountered
    int returnValue = 1;

    //This variable is used to hold
    //the type of tile which the player has landed on
    int tileType;

    //This variable checks if the player
    //is currently on this tile
    boolean playerHere;

    //A file object is being created
    //where the name is given depending
    //on the number of the player
    File file = new File("map_player_" + (playerIndex + 1) + ".html");

    //The actual file is created here
    try {
        //If file already exists set return
        // value to 2 to mark that it is being overwritten
        if(!file.createNewFile()){
            returnValue = 2;
        }
    } catch (IOException e) {
        e.printStackTrace();
        returnValue = 0; //Set return value to error
    }

    //This object is used to be able to add
    //to the string easily
    StringBuilder htmlText = new StringBuilder();

    //This is the html code which is going
    //to be placed in each file
    htmlText.append( "<!doctype html>\n" );
    htmlText.append( "<html>\n" );

    htmlText.append( "<head>\n" );
    htmlText.append( "<style>\n" );

    htmlText.append("div {\n" +
        //The width of the grid is

```

```

// set depending on the inputted map size
//The height is larger than
//the width since we are also goign to
//have to count the header which is above the grid
"    width: ").append(mapSize).append("00px;\n")
.append("    height: ").append(mapSize + 1).append("00px;\n")
.append("}\n")
.append("\n")
.append(".header {\n").append(
//The width of the header is changed
//depending on the size of the map
"    width: ").append(mapSize).append("00px;\n")
.append("    height: 100px;\n")
.append("    outline: 1px solid;\n")
.append("    float: left;\n")
.append("    text-align: center;\n")
.append("    background-color: #1f599a;\n")
.append("    font-family: Arial, sans-serif;\n")
.append("    font-size: 20px;\n")
.append("    color: white;\n")
.append("}\n")
.append("\n")
.append(".cellGray {\n")
.append("    width: 100px;\n")
.append("    height: 100px;\n")
.append("    outline: 1px solid;\n")
.append("    float: left;\n")
.append("    background-color: Gray;\n")
.append("}\n")
.append("\n")
.append(".cellGreen {\n")
.append("    width: 100px;\n")
.append("    height: 100px;\n")
.append("    outline: 1px solid;\n")
.append("    float: left;\n")
.append("    background-color: Green;\n")
.append("}\n")
.append("\n")
.append(".cellBlue {\n")
.append("    width: 100px;\n")
.append("    height: 100px;\n")
.append("    outline: 1px solid;\n")
.append("    float: left;\n")
.append("    background-color: Blue;\n")
.append("}\n")
.append("\n")
.append(".cellYellow {\n")
.append("    width: 100px;\n")
.append("    height: 100px;\n")
.append("    outline: 1px solid;\n")
.append("    float: left;\n")
.append("    background-color: Yellow;\n")
.append("}\n");

```

```

htmlText.append( "</style>\n" );
htmlText.append( "</head>\n\n" );

htmlText.append( "<body>\n" );

htmlText.append("<div>\n" + "        <div class=\"header\"> \n" + "
\n" +
        //First we need to set a header
        //for each game map which each player sees
        "        <p> Player ").append(playerIndex + 1)
        .append("</p>\n")
        .append("        <p> Moves: ").append(direction).append(" </p> \n")
        .append("        </div>\n")
        .append("        \n");

//Now we will build the current
//map depending on the players current position
//We will change colours of new tiles that have
//been stepped on and mark the player's current position
//For loop used to loop through each grid
for (int j = 0; j < mapSize; j++) {
    for (int i = 0; i < mapSize; i++) {

        //playerHere is set to false at each iteration
        playerHere = false;

        //Check if the player went on this tile already
        if(players.get(playerIndex).ifTileExists(i, j)){

            //If the tile exists then the player must
            be on one of these tiles
            //Checking if the current tile is the
            players current position on the map
            if(players.get(playerIndex).position.x == i &&
                players.get(playerIndex).position.y == j){
                playerHere = true;
            }

            //Obtain the tile type of the current tile
            tileType = map.getTileType(i,j);
        }

        else{
            //If not the tile has a default tile type
            tileType = 3;
        }

        switch(tileType){
            //Grass tile
            case 0:

                if(playerHere){

                    htmlText.append("<div class=\"cellGreen\">" +

```

```

        "<img src=\"player.png\" alt=\"player\">" +
        "</div>\n");
    }

    else{

        htmlText.append("<div class=\"cellGreen\"></div>\n");

    }
    break;

//Water tile
case 1:

    if(playerHere){

        htmlText.append("<div class=\"cellBlue\">" +
            "<img src=\"player.png\" alt=\"player\">" +
            "</div>\n");

    }

    else{

        htmlText.append("<div class=\"cellBlue\"></div>\n");

    }
    break;

//Treasure Tile
case 2:

    if(playerHere){

        htmlText.append("<div class=\"cellYellow\">" +
            "<img src=\"player.png\" alt=\"player\">" +
            "</div>\n");

    }

    else{

        htmlText.append("<div class=\"cellYellow\"></div>\n")

    }
    break;

default:
    //No need to check for player here
    //as a player can never be on a gray tile
    htmlText.append("<div class=\"cellGray\"></div>\n");
    break;
    }
}
}

```

```

htmlText.append("\n</div>");
htmlText.append( "</body>\n" );
htmlText.append( "</html>\n" );

try{
    BufferedWriter bw = new BufferedWriter(new FileWriter(file));
    bw.write(htmlText.toString());
    bw.close();
}catch(IOException io){
    io.printStackTrace();
    returnValue = 0;
}

return returnValue;
}

//Method used to get the last n directions
String getPreviousDirections(int playerIndex){
    String directions;
    StringBuilder stringBuilder = new StringBuilder();

    int directionSize = players.get(playerIndex).directions.size();

    //Loop for the last 6 directions the player has moved
    for(int i = 1; i <= 6; i++){
        //If only one direction has been entered
        if(directionSize == 1){
            stringBuilder.append(" ").append(players.get(playerIndex).
                directions.get(directionSize - 1));
            break;
        }
        //If more than 1 directions have been entered
        else if (directionSize >1){
            //Add direction unless there are
            //less than 6 total directions
            if(directionSize - i <0){
                break;
            }
            stringBuilder.append(" ").append(players.get(playerIndex).
                directions.get(directionSize - i));
        }
    }

    directions = stringBuilder.toString();

    return directions;
}

//Method to delete Html files
void deleteHtmlFiles(int playerNum){

    //Loops through all player files
    for(int i = 1; i <= playerNum ; i++){

```



```

        //Delete each file iteratively
        try{
            File file = new File("map_player_" + i + ".html");

            if(!file.delete()) {
                System.out.println("File does not exist");
            }
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}

//Method to exit the game
private void exitGame(Game game){
    if(getExitChar() == 'e'){
        //Delete all Html Files
        deleteHtmlFiles(game.playerNum);
        System.out.println("Thank you for playing!");
    }
}

//Method to get the user to input e to exit the program
private char getExitChar(){
    char exit = 'x';

    System.out.println("Press e if you would like to exit the program");

    //Make sure that user input is valid
    while(exit == 'x' || exit == 'y') {
        scanner = new Scanner(System.in);
        exit = validateExitChar(scanner);
    }

    return exit;
}

//Method to validate user input from getExitChar()
char validateExitChar(Scanner scanner){
    String input;//User input
    char c;//User input after it being converted into a character

    try {
        //Set to user input from getDirections
        input = scanner.next();

        //Ensure that inputted string is of length 1
        if (input.length() > 1) {
            throw new RuntimeException("Input too long.
            Please enter e to exit");
        }

        //Convert input string to char
        c = input.charAt(0);
    }
}

```

```

        //Ensure that character is a letter
        if (!Character.isLetter(c)) {
            throw new RuntimeException("Input is not a character.
            Please enter e to exit");
        }
    }

    //If an error is thrown in the try block
    catch (RuntimeException e) {
        System.err.println(e.getMessage());
        return 'y'; //Return an error value which we will associate with an ex
    }

    //Change char input to lowercase to allow U, D, L and R
    c = Character.toLowerCase(c);

    //If input is a char but not one of the directions
    if (c != 'e') {
        System.err.println("Invalid input. Please enter e to exit");
        return 'x'; //Return an error value which
        //we will associate with an invalid character when testing
    }

    //If input is correct
    else {
        return c; //Return valid user inputted character
    }
}
}

```

## Map Class

The Map class is used to generate the tile map which the players use to play the game on. Also the tile events which occur are also taken care of by the Map class.

Listing 2: The initial code of the Map class

```

import java.util.Random;

class Map {

    //Size of the map. Total squares will be mapSize x mapSize
    int mapSize;

    //The map will consist of 2d array of tiles along with each tile's type
    int[][][] tiles;

    //Below are counters used to hold the current number of tiles with the corres
    //The use of these is so that the program does not have more than one treasure
    int grassCount = 0;
    private int waterCount = 0;
}

```

```

private int treasureCount = 0;

//Constructor for the map object
Map(){
}

//This method is used on start up to create the map
//All tiles are randomly assigned a tile type
void generate(){

    tiles = new int[mapSize][mapSize][1];

    Random random = new Random();

    //Holds the current random tile obtained
    int[] randomPair = new int[2];

    //Holds a boolean value which determines if the [i][j] tile has already been
    boolean[][][] generatedTiles = new boolean[mapSize][mapSize][1];

    //This value holds the tile type (grass, water, treasure) which corresponds to
    int tileType;

    //The maximum number of water tiles in a map is set to one less than mapSize
    double waterMaxTiles = Math.ceil(mapSize-1);

    //The maximum number of treasure tiles is one
    int treasureMaxTiles = 1;

    //Initialising the generatedTiles array
    for(int i = 0; i < mapSize; i++) {
        for (int j = 0; j < mapSize; j++) {
            generatedTiles[i][j][0] = false;
        }
    }

    //Keep on looping until all tiles are randomly generated
    while(generatedTilesNum(generatedTiles) != mapSize*mapSize){

        //Randomly generate a pair of tiles
        //Numbers generated will be from 0 to mapSize-1
        randomPair[0] = random.nextInt(mapSize);
        randomPair[1] = random.nextInt(mapSize);

        //Now checking if the tile has already been obtained in the generated tiles
        //This is done by checking if the boolean value of the tile is true
        if(generatedTiles[randomPair[0]][randomPair[1]][0]){
            //Tile has already been generated
            //So the user must get another tile which has not already been generated

            //Keeps on looping until a new tile is generated
            while(generatedTiles[randomPair[0]][randomPair[1]][0]){

```

```

        randomPair[0] = random.nextInt(mapSize);
        randomPair[1] = random.nextInt(mapSize);
    }
}

//Keep on looping until the current tile is given a tile type
do {

    //Set the tile type for the newly generated tile
    //A random number from 0 to 2 is obtained which correspond to a tile type
    tileType = random.nextInt(3);

    //A switch statement is used to go through each of the possible tile types
    switch (tileType) {

        //Grass tile
        case 0:
            tiles[randomPair[0]][randomPair[1]][0] = 0;

            //The counter is updated since another grass tile has been placed
            grassCount += 1;

            break;

        //Water tile
        case 1:
            if (waterCount == waterMaxTiles) {
                //When the maximum number of water tiles is reached the check is
                //This is so that the while loop will continue until the map
                //is not full
                continue;
            } else {
                tiles[randomPair[0]][randomPair[1]][0] = 1;

                //The counter is updated since another water tile has been placed
                waterCount += 1;
            }
            break;

        //Treasure tile
        case 2:
            if (treasureCount == treasureMaxTiles) {
                //When a treasure tile is already placed the check is
                continue;
            } else {
                tiles[randomPair[0]][randomPair[1]][0] = 2;
                //The counter is updated since a treasure tile has been placed
                treasureCount += 1;
            }
            break;
    }
}

```

```

        default:
            //This case is accessed only when a random number which i
            System.err.println("Invalid random number obtained");
            break;

        }

        //add the the tile to the generatedTiles array
        generatedTiles[randomPair[0]][randomPair[1]][0] = true;
    }while( tiles[randomPair[0]][randomPair[1]][0] != 0 && tiles[randomPa
}
}

//Method to obtain the current number of elements in the generatedTiles array
int generatedTilesNum(boolean[][][] array){
    int count = 0;

    for(int i = 0; i < mapSize; i++){
        for(int j = 0; j < mapSize; j++){
            if(array[i][j][0]){
                count++;
            }
        }
    }

    return count;
}

//Method used to show the map
void showMap(){
    for (int x = 0; x < mapSize; x++) {
        for (int y = 0; y < mapSize; y++) {
            System.out.println("(" + x + "," + y + ") -> " + tiles[x][y][0])
        }
    }
}

//This method is used to create a 2d array which holds the map location of al
int[][] getGrassTiles(){

    //This array holds the number of grass elements each with 2 spaces for th
    int[][] grassTiles = new int[grassCount][2];

    //Counter used to keep track of the element in grassTiles
    int i = 0;

    //Loop through all the tiles
    for (int x = 0; x < mapSize; x++) {
        for (int y = 0; y < mapSize; y++) {

            //To obtain which tiles are grass just check the tile type indica

            //If the current tile has a grass tile type
            if(tiles[x][y][0] == 0){

```

```

        //Save the x and y coordinates in the grassTiles array
        grassTiles[i][0] = x;
        grassTiles[i][1] = y;

        //The counter is incremented so as to be able to go to the next tile
        i++;
    }
}

return grassTiles;
}

//This method is used to get the tile type of the current tile
int getTileType(int x, int y){
    return tiles[x][y][0];
}

//This method handles the events of when a player lands on each specific tile
void evaluateCurrentPlayerTile(Player player){

    int tileType = tiles[player.position.x][player.position.y][0];

    switch(tileType){

        //When grass tile
        case 0:
            //Nothing happens since a player is able to walk on grass
            break;

        //When water tile
        case 1:
            //Get the start position of the current player
            int startPositionx = player.positions.get(0).x;
            int startPositiony = player.positions.get(0).y;

            //The start position of the current player is added again to the
            player.addToPositions(startPositionx, startPositiony);

            //The current position of the current player is reset to the start position
            player.position.x = startPositionx;
            player.position.y = startPositiony;

            //Display message
            System.out.println("Player stepped on a blue tile, moving back to start");
            break;

        //When treasure tile
        case 2:
            //The foundTreasure element is set to true
            player.foundTreasure = true;
            break;
    }
}

```

```

        default:
            break;
    }
}
}

```

## Player Class

The Player class manages the individual properties of a player, such as the tile positions the player has moved on and the directions in which the player has moved.

Listing 3: The initial code of the Player class

```

import java.util.ArrayList;
import java.util.Random;

class Player {

    //Player's current position
    Position position;

    //This array list is used to hold the previous positions the player
    ArrayList<Position> positions = new ArrayList<Position>();

    //This array list is used to hold the previous directions of the player
    ArrayList<String> directions = new ArrayList<String>();

    //Check to see if a player has found the treasure
    boolean foundTreasure;

    //Constructor for the player object
    Player(){
    }

    //Constructor for the player object when the player's position is given
    Player(Position position){
        this.position = position;
    }

    //Method used to add a position to the positions ArrayList using the x and y
    void addToPositions(int posx, int posy){
        Position position = new Position(posx, posy);
        positions.add(position);
    }

    //Method to move the player's position according to a given direction
    void move(char direction){

        //A switch statement is used to represent all possible directions
        switch(direction){
            case 'l':
                // change player's position

```

```

        this.position.x --;
        // add position to list of previous positions
        addToPositions(position.x, position.y);
        // add direction to list of previous directions
        directions.add("left");
        break;

    case 'r':
        // change player's position
        this.position.x ++;
        // add position to list of previous positions
        addToPositions(position.x, position.y);
        // add direction to list of previous directions
        directions.add("right");
        break;

    case 'u':
        // change player's position
        this.position.y --;
        // add position to list of previous positions
        addToPositions(position.x, position.y);
        // add direction to list of previous directions
        directions.add("up");
        break;

    case 'd':
        // change player's position
        this.position.y ++;
        // add position to list of previous positions
        addToPositions(position.x, position.y);
        // add direction to list of previous directions
        directions.add("down");
        break;

    default:
        break;
}

//This method sets the starting position of a player
Position setStartingPosition(int[][] grassTiles){

    Random random = new Random();

    Position position = new Position(0, 0);

    //Obtaining the length of grassTiles so as to be able to know from which
    int grassCount = grassTiles.length;

    //random number from 0 to length of grassTiles is obtained
    int grassTilesIndex = random.nextInt(grassCount);

    //The start position is set
    position.x = grassTiles[grassTilesIndex][0];

```



```

        position.y = grassTiles[grassTilesIndex][1];

        //The current position of the player is set to the start position
        this.position = position;

        //The start positions is added to the created player
        addToPositions(position.x, position.y);

        return position;
    }

    //This method is used in the game class to check if a player has already been
    boolean ifTileExists(int x, int y){

        //Create the Position object to use to compare
        Position positionUse = new Position(x,y);

        //Looping through element in the positions array list
        for (Position position : positions) {

            //Comparing the x and y values of the current Position object in the
            if(positionUse.x == position.x && positionUse.y == position.y){

                //If one of the obejct in the array list matched then it exists i
                return true;
            }
        }
        return false;
    }
}

```

## Position Class

The Position class is used to simplify the way in which the program interprets the player's location within the tile map.

Listing 4: The initial code of the Position class

```

public class Position {

    //Value of each coordinate
    int x;
    int y;

    //Method to display a position object
    @Override
    public String toString() {
        return "Position{" +
            "x=" + x +
            ", y=" + y +
            '}';
    }
}

```

```
//Constructor for the player object
Position(){
}

//Constructor for the position object when both x and y values are given
Position(int px, int py) {
    x = px;
    y = py;
}
```

## Enhancements

### Different Map Types

The design pattern chosen for this enhancement was the **factory design pattern**. Since this enhancement was centred around the map's creation then it was clear that a creational design pattern had to be chosen. The main things which needed to be kept in mind were that 2 initial map types had to be implemented while more map types could easily be added to the future. Therefore, the Factory design pattern was the most suitable for the task.

The class diagram of the enhancement can be seen below.

\*\*\*\*Add diagram here\*\*\*\*

Our implementation is centred around a *Map* interface which contains each method which will be used by the different map types. Each type of map then has a class which implements this main *Map* interface as can be seen in the class diagram above. This design allows for the easy implementation of additional map types in the future.

A factory method is present in the *MapCreator* class which is passed the map's type and its size. This class has a creator subclass for every map type, allowing for easy creation of additional map types in the future. An instance of the correct creator subclass will then be made and used to create the map. This can be seen in the code snippet below.

```
//Factory Method
Map createMap(String type, int mapSize){

    MapCreator creator;

    if(type.equals("safe")){
        creator = new SafeMapCreator(mapSize);
    }
    else if(type.equals("hazardous")){
        creator = new HazardousMapCreator(mapSize);
    }
    else{
        creator = null;
        System.err.println("Invalid map type");
    }

    if (creator != null) {
        return creator.create();
    }
    else{
        return null;
    }
}
```

The map is then created in the appropriate creator subclass by setting the map size and calling the generate method in the map type's class (either the *SafeMap* or *HazardousMap* class in our implementation but more can easily be added). A code snippet of this create method for the "safe" map type is shown below

```
//Method to create and return a safe map
@Override
public Map create(){
    //getInstance method used here to obtain a
    //static instance of the hazardous map
    SafeMap map = SafeMap.getInstance();
    map.setMapSize(mapSize);
    map.generate();
    return map;
}
```

In the generate method for the safe map, the maximum amount of water tiles is set to 10% and then rounded down to the nearest integer. This code snippet is shown below.

```
//The maximum number of water tiles in a map is set to
//10% of the total tiles
int waterMaxTiles = (int) Math.floor((mapSize*mapSize) * 0.1);
```

In the generate method for the hazardous map, the maximum amount of water tiles is set to 25% and then rounded up to the nearest integer. This code snippet is shown below.

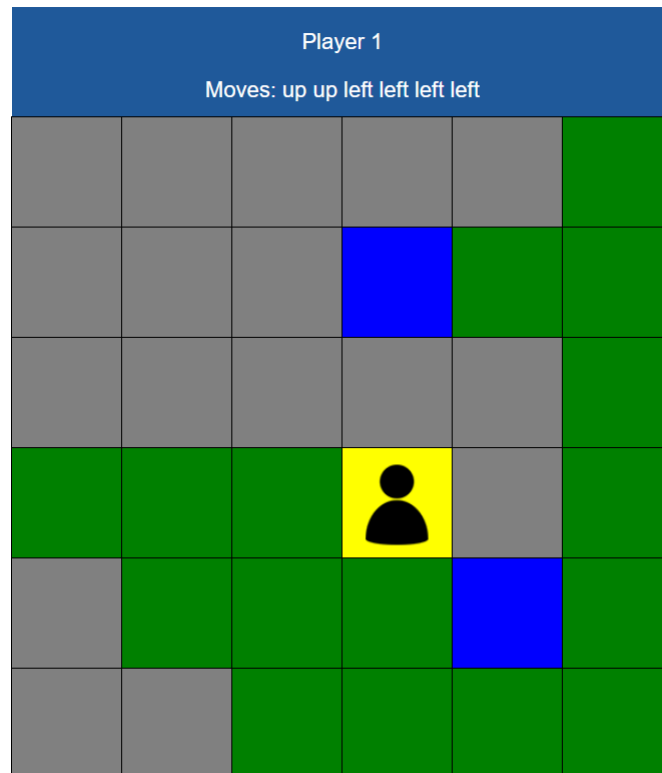
```
//The maximum number of water tiles in a map is set to
//25% of the total tiles
int waterMaxTiles = (int) Math.ceil((mapSize*mapSize) * 0.25);
```

Apart than these changes, the map is generated as it was in the generate method in the basic version of the program before the enhancements.

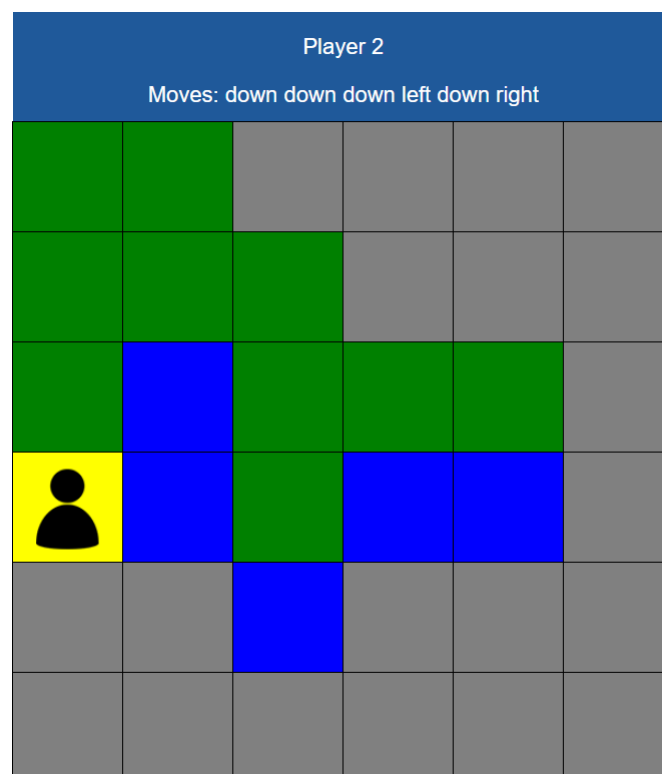
When running the program, the user is asked what map type they would like to play with. This can be seen below.

```
Welcome to the Treasure Map Game by Martin Bartolo and Mikhail Cassar
How many players will be playing? (Pick a number between 2 and 8)
4
How many teams should the players be split into? (Pick a number between 2 and the amount of players in the game)
2
How large would you like the map to be? (Map will be n x n)
6
Would you like to play in
1) a safe map with 10% water squares
2) a hazardous map with 25%-35% water squares
```

After the user chooses the map type, the map is created as discussed on the previous page and then the game can be played. On the next page is a comparison between the ending screen of a game played by 4 players in 2 teams on a 6x6 safe map and a game played by the same players on a 6x6 hazardous map. Notice the different amounts of water tiles encountered between the 2 map types.



Ending screen of game using a safe map



Ending screen of game using a hazardous map

The design pattern chosen for this enhancement was the **factory design pat-**

**tern.** Since this enhancement was centred around the map's creation then it was clear that a creational design pattern had to be chosen. The main things which needed to be kept in mind were that 2 initial map types had to be implemented while more map types could easily be added to the future. Therefore, the Factory design pattern was the most suitable for the task.

The class diagram of the enhancement can be seen below.

\*\*\*\*Add diagram here\*\*\*\*

Our implementation is centred around a *Map* interface which contains each method which will be used by the different map types. Each type of map then has a class which implements this main *Map* interface as can be seen in the class diagram above. This design allows for the easy implementation of additional map types in the future.

A factory method is present in the *MapCreator* class which is passed the map's type and its size. This class has a creator subclass for every map type, allowing for easy creation of additional map types in the future. An instance of the correct creator subclass will then be made and used to create the map. This can be seen in the code snippet below.

```
//Factory Method
Map createMap(String type, int mapSize){

    MapCreator creator;

    if(type.equals("safe")){
        creator = new SafeMapCreator(mapSize);
    }
    else if(type.equals("hazardous")){
        creator = new HazardousMapCreator(mapSize);
    }
    else{
        creator = null;
        System.err.println("Invalid map type");
    }

    if (creator != null) {
        return creator.create();
    }
    else{
        return null;
    }
}
```

The map is then created in the appropriate creator subclass by setting the map size and calling the generate method in the map type's class (either the *SafeMap* or *HazardousMap* class in our implementation but more can easily be added). A code snippet of this create method for the "safe" map type is shown below

```
//Method to create and return a safe map
@Override
public Map create(){
    //getInstance method used here to obtain a
    //static instance of the hazardous map
    SafeMap map = SafeMap.getInstance();
    map.setMapSize(mapSize);
    map.generate();
    return map;
}
```

In the generate method for the safe map, the maximum amount of water tiles is set to 10% and then rounded down to the nearest integer. This code snippet is shown below.

```
//The maximum number of water tiles in a map is set to
//10% of the total tiles
int waterMaxTiles = (int) Math.floor((mapSize*mapSize) * 0.1);
```

In the generate method for the hazardous map, the maximum amount of water tiles is set to 25% and then rounded up to the nearest integer. This code snippet is shown below.

```
//The maximum number of water tiles in a map is set to
//25% of the total tiles
int waterMaxTiles = (int) Math.ceil((mapSize*mapSize) * 0.25);
```

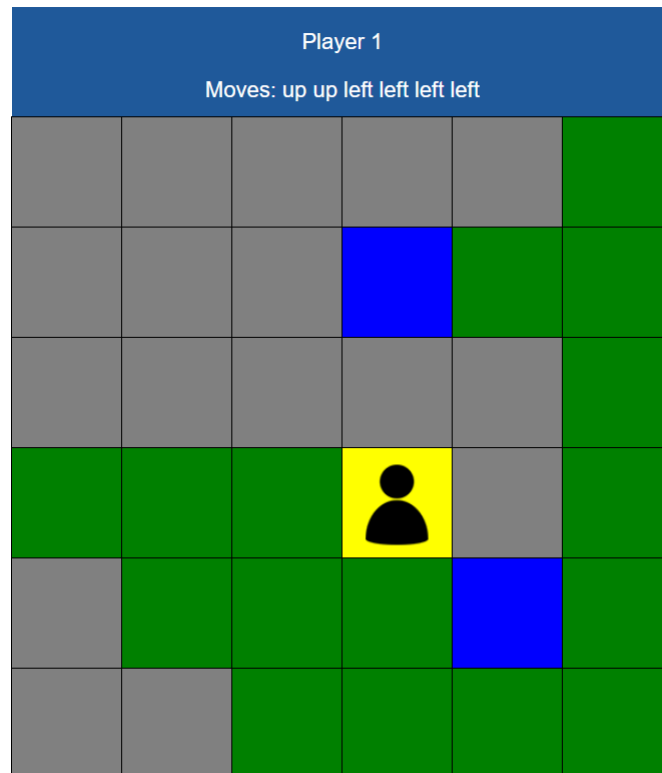
Apart than these changes, the map is generated as it was in the generate method in the basic version of the program before the enhancements.

When running the program, the user is asked what map type they would like to play with. This can be seen below.

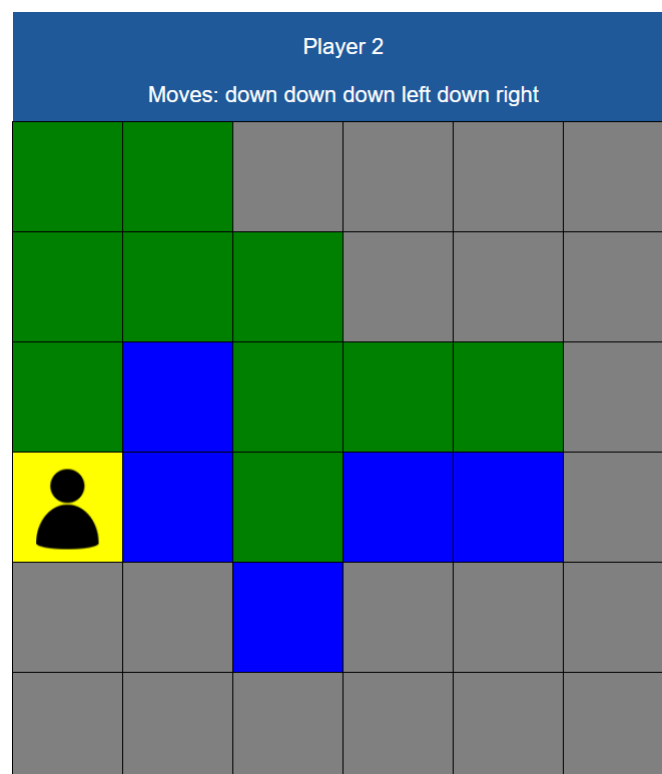
```
Welcome to the Treasure Map Game by Martin Bartolo and Mikhail Cassar
How many players will be playing? (Pick a number between 2 and 8)
4
How many teams should the players be split into? (Pick a number between 2 and the amount of players in the game)
2
How large would you like the map to be? (Map will be n x n)
6
Would you like to play in
1) a safe map with 10% water squares
2) a hazardous map with 25%-35% water squares
```

After the user chooses the map type, the map is created as discussed on the previous page and then the game can be played. On the next page is a comparison between the ending screen of a game played by 4 players in 2 teams on a 6x6 safe map and a game played by the same players on a 6x6 hazardous map. Notice the different amounts of water tiles encountered between the 2 map types.





Ending screen of game using a safe map



Ending screen of game using a hazardous map

The information provided will be regarding the basic version of the game i.e. be-

fore any enhancements to the game were made. The basic version of the game can be played by going to the tag "Part2".

## **Class Diagram**

This is the class diagram of the basic version of the game, here one will see how all the classes interact together to provide the user with a functioning game which meets the specified requirements.

## **Game Enhancements**

## **Code Coverage Statistics**

## **Running The Game**