Treasure Game

Dylan Frendo (138497M)
Miguel Dingli (049997M)

CPS 2002 - Software Engineering

Department of Computer Science
University of Malta

May 16, 2017

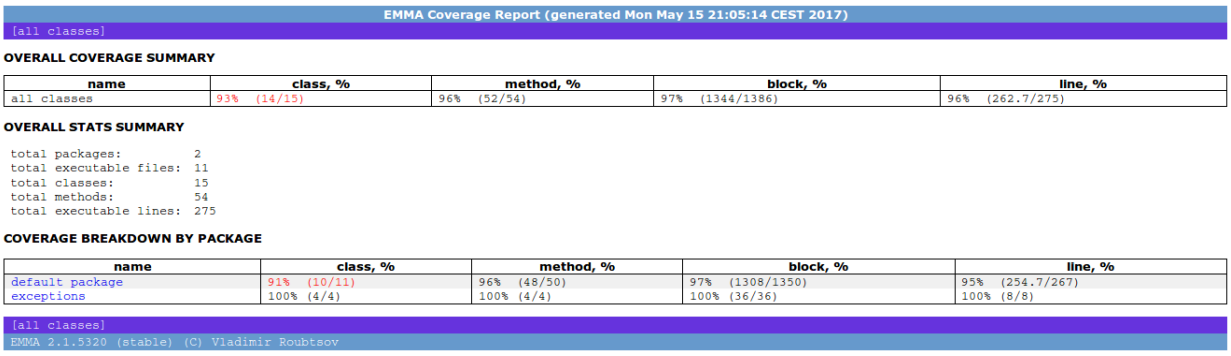
Contents

1 Code Coverage

In this section, code coverage analysis will be provided for both part 2 and part 3.

1.1 Part 2

Figure 1 shows the coverage of all the packages that make up the game (up to Part 2). The exceptions package was fully covered and will thus not be mentioned. Figure 2 shows the default package code coverage summary.



EMMA Coverage Report (generated Mon May 15 21:05:14 CEST 2017)

[all classes]

OVERALL COVERAGE SUMMARY

name	class, %	method, %	block, %	line, %
all classes	93% (14/15)	96% (52/54)	97% (1344/1386)	96% (262.7/275)

OVERALL STATS SUMMARY

total packages:	2
total executable files:	11
total classes:	15
total methods:	54
total executable lines:	275

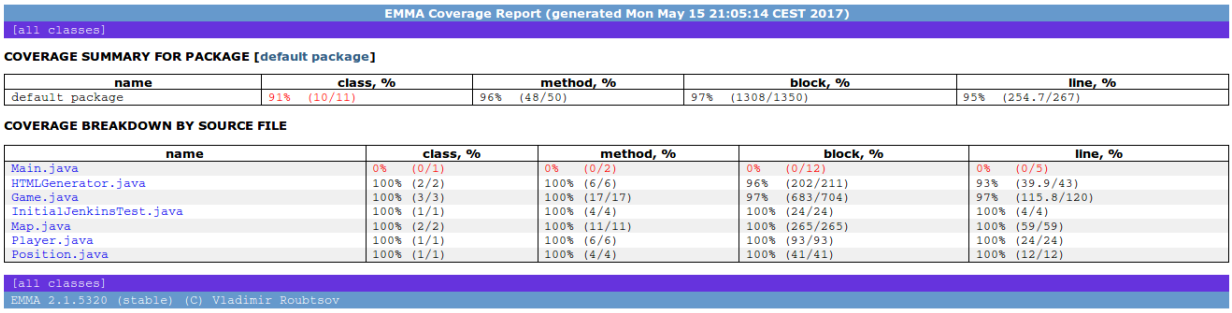
COVERAGE BREAKDOWN BY PACKAGE

name	class, %	method, %	block, %	line, %
default package	91% (10/11)	96% (48/50)	97% (1308/1350)	95% (254.7/267)
exceptions	100% (4/4)	100% (4/4)	100% (36/36)	100% (8/8)

[all classes]

EMMA 2.1.5320 (stable) (C) Vladimir Roubtsov

Figure 1: Package Coverage Summary for Part 2.



EMMA Coverage Report (generated Mon May 15 21:05:14 CEST 2017)

[all classes]

COVERAGE SUMMARY FOR PACKAGE [default package]

name	class, %	method, %	block, %	line, %
default package	91% (10/11)	96% (48/50)	97% (1308/1350)	95% (254.7/267)

COVERAGE BREAKDOWN BY SOURCE FILE

name	class, %	method, %	block, %	line, %
Main.java	0% (0/1)	0% (0/2)	0% (0/12)	0% (0/5)
HTMLGenerator.java	100% (2/2)	100% (6/6)	96% (202/211)	93% (39.9/43)
Game.java	100% (3/3)	100% (17/17)	97% (683/704)	97% (115.8/120)
InitialJenkinsTest.java	100% (1/1)	100% (4/4)	100% (24/24)	100% (4/4)
Map.java	100% (2/2)	100% (11/11)	100% (265/265)	100% (59/59)
Player.java	100% (1/1)	100% (6/6)	100% (93/93)	100% (24/24)
Position.java	100% (1/1)	100% (4/4)	100% (41/41)	100% (12/12)

[all classes]

EMMA 2.1.5320 (stable) (C) Vladimir Roubtsov

Figure 2: Default Package Coverage Summary for Part 2.

1.1.1 Main.java

In this file, none of the lines were tested. However, since this class simply creates a new game instance and calls two public methods which are tested elsewhere, not testing these lines is not actually a problem. Refer to Figure 3 for Main.java.

```
 4 public class Main {
 5
 6     public static void main(String args[]) throws Exception {
 7
 8         final Game game = new Game();
 9         game.setup();
10         game.startGame();
11     }
12 }
```

Figure 3: Main.java Coverage.

1.1.2 HTMLGenerator.java

In this file, three lines of code were not tested.

Figure 4 shows a switch statement where its conditions are enum values. Even though all of the possible enum cases available were stated and tested, the default is still required in Java. Testing the default case is not possible since all of the enum values were covered.

```
131     private String determineCellType(final Map.TILE_TYPE cellType, final boolean playerIsOnTile) {
132         switch (cellType) {
133             case GRASS:
134                 return playerIsOnTile ? GRASS_CELL_WITH_PLAYER : GRASS_CELL;
135             case WATER:
136                 return playerIsOnTile ? WATER_CELL_WITH_PLAYER : WATER_CELL;
137             case TREASURE:
138                 return playerIsOnTile ? TREASURE_CELL_WITH_PLAYER : TREASURE_CELL;
139             default:
140                 return "";
141         }
142     }
```

Figure 4: HTMLGenerator.java Coverage (1/2).

Figure 5 shows an untested catch statement. The catch statement is not being tested because the exception PositionIsOutOfRange thrown by wasVisited(int, int) will never be thrown for this case. As can be seen in the two *for* loops, the loop will never go beyond the map size and thus the position (x, y) will never be out of range. The method is called in different locations, which requires the exception to be used, and thus it cannot be removed.

```
81      private String createTable(final Map map, final Player player) {
82          final int mapSize = map.getMapSize();
83          int x, y;
84          final StringBuilder table = new StringBuilder();
85          table.append(createCaption(player));
86
87          for (int i = 0; i < mapSize; i++) {
88              table.append(TAB_FOR_ROW);
89              table.append("<tr>\n");
90              for (int j = 0; j < mapSize; j++) {
91                  x = player.getPosition().getX();
92                  y = player.getPosition().getY();
93
94                  try {
95                      if (x == j && y == i) {
96                          table.append(determineCellType(map.getTileType(j, i), true));
97                      } else if (player.wasVisited(j, i)) {
98                          table.append(determineCellType(map.getTileType(j, i), false));
99                      } else {
100                         table.append(IDLE_CELL);
101                     }
102                 } catch (PositionIsOutOfRange positionIsOutOfRange) {
103                     // Create Table loop is incorrect.
104                     positionIsOutOfRange.getMessage();
105                 }
106             }
107             table.append(TAB_FOR_ROW);
108             table.append("</tr>\n");
109         }
110         return table.toString();
111     }
```

Figure 5: HTMLGenerator.java Coverage (2/2).

1.1.3 Game.java

In this file, four lines and two switch statement default cases were not tested. Since the reason behind the two switch statements not being covered is the same as that of the switch statement previously discussed, these will not be presented in this subsection.

Figure 6 shows the first untested line found in startGame(). This line is not covered since the only way that the players array can be uninitialized is if setup() is not called. However, if setup() is not called, then map is also uninitialized, and so the first condition is satisfied instead.

```
111         // Check if either map or players array were not initialized
112         if (map == null) {
113             throw new GameWasNotInitialized("Map");
114         } else if (players == null) {
115             throw new GameWasNotInitialized("Players array");
116         }
```

Figure 6: Game.java Coverage (1/3).

Figure 7 shows another untested line found in setMapSize(). The reason behind this also has to do with the setup() method. Since setMapSize() is only called from setup(), in which the players array is initialized, then the "players == null" condition should never actually be satisfied.

```
208         // Check if players array was initialized
209         if (players == null) {
210             throw new GameWasNotInitialized("Players array");
211         }
```

Figure 7: Game.java Coverage (2/3).

Figure 8 shows two untested lines found in generateHTMLFiles(). These lines were not covered since it requires an IOException to occur, which cannot be simulated in the tests. If the IOException is thrown, the tests cannot detect that an exception was thrown since the method handles the exception without changing any variables and the method is private. Also the variables used to indicate a file location are fixed and thus cannot be changed.

```
240         try {
241             // .gitignore is still needed in the directory, thus re-write it after
242             // cleaning the directory
243             gitIgnore = FileUtils.readFileToString(GitIgnoreLocation);
244             FileUtils.cleanDirectory(new File(playersMapLocation));
245             FileUtils.writeStringToFile(GitIgnoreLocation, gitIgnore);
246             for (final Player player : players) {
247                 playerFile = new File(playersMapLocation + "map_player_" + player.getID() + ".html");
248                 FileUtils.copyFile(HTMLTemplateLocation, playerFile);
249                 new HTMLGenerator(playerFile, map, player);
250             }
251         } catch (IOException e) {
252             e.getMessage();
253         }
```

Figure 8: Game.java Coverage (3/3).

1.2 Part 3

Figure 9 shows the coverage of all the packages that make up the game (up to Part 3). Once again, the exceptions package was fully covered and will thus not be mentioned. Figure 10 shows the default package code coverage summary.

EMMA Coverage Report (generated Tue May 16 21:01:41 CEST 2017)

[all classes]

OVERALL COVERAGE SUMMARY

name	class, %	method, %	block, %	line, %
all classes	95% (21/22)	97% (73/75)	98% (1873/1921)	97% (382.6/396)

OVERALL STATS SUMMARY

total packages: 2
total executable files: 16
total classes: 22
total methods: 75
total executable lines: 396

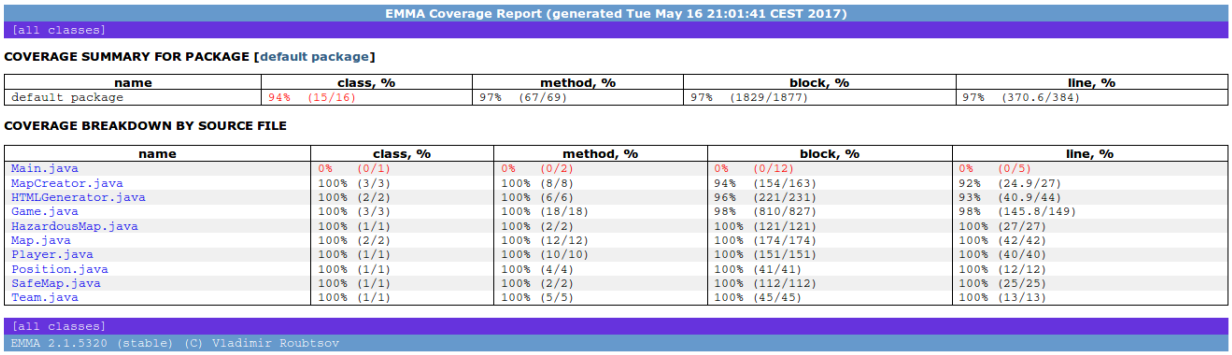
COVERAGE BREAKDOWN BY PACKAGE

name	class, %	method, %	block, %	line, %
default package	94% (15/16)	97% (67/69)	97% (1829/1877)	97% (370.6/384)
exceptions	100% (6/6)	100% (6/6)	100% (44/44)	100% (12/12)

[all classes]

EMMA 2.1.5320 (stable) (C) Vladimir Roubtsov

Figure 9: Package Coverage Summary for Part 3.



EMMA Coverage Report (generated Tue May 16 21:01:41 CEST 2017)

[all classes]

COVERAGE SUMMARY FOR PACKAGE [default package]

name	class, %	method, %	block, %	line, %
default package	94% (15/16)	97% (67/69)	97% (1829/1877)	97% (370.6/384)

COVERAGE BREAKDOWN BY SOURCE FILE

name	class, %	method, %	block, %	line, %
Main.java	0% (0/1)	0% (0/2)	0% (0/12)	0% (0/5)
MapCreator.java	100% (3/3)	100% (8/8)	94% (154/163)	92% (24.9/27)
HTMLGenerator.java	100% (2/2)	100% (6/6)	96% (221/231)	93% (40.9/44)
Game.java	100% (3/3)	100% (18/18)	98% (810/827)	98% (145.8/149)
HazardousMap.java	100% (1/1)	100% (2/2)	100% (121/121)	100% (27/27)
Map.java	100% (2/2)	100% (12/12)	100% (174/174)	100% (42/42)
Player.java	100% (1/1)	100% (10/10)	100% (151/151)	100% (40/40)
Position.java	100% (1/1)	100% (4/4)	100% (41/41)	100% (12/12)
SafeMap.java	100% (1/1)	100% (2/2)	100% (112/112)	100% (25/25)
Team.java	100% (1/1)	100% (5/5)	100% (45/45)	100% (13/13)

[all classes]

EMMA 2.1.5320 (stable) (C) Vladimir Roubtsov

Figure 10: Default Package Coverage Summary for Part 3.

1.2.1 Main.java

The untested code in this class is the same as that of the previous section.

1.2.2 HTMLGenerator.java

The untested code in this class is also the same as that of the previous section.

1.2.3 Game.java

The untested code in this class is also the same as that of the previous section.

1.2.4 MapCreator.java

There are two lines that are not tested in this file.

Figure 11 shows another switch statement where its condition is an enum. Even though all possible enums cases available were stated and tested, the default is still required in Java. Testing the default case is not possible since that enum does not exist.

```
37          switch (type) {
38              case SAFE_MAP:
39                  map = new SafeMap();
40                  // Set map size and generate map
41                  setMapSize(game);
42                  map.generate();
43                  return map;
44              case HAZARDOUS_MAP:
45                  map = new HazardousMap();
46                  // Set map size and generate map
47                  setMapSize(game);
48                  map.generate();
49                  return map;
50              default:
51                  return Map.getInstance();
52          }
```

Figure 11: MapCreator.java Coverage (1/2).

Figure 12 shows the setMapSize() method which was previously in the Game class but is now in the MapCreator class. Essentially, the reason why the line was not originally tested

still holds since this method is still called only from the setup() method in the Game class thus players array will always be declared beforehand.

```
65      private void setMapSize(Game game) throws GameWasNotInitialized {
66          Map map;
67
68          // Check if players array was initialized
69          if (game.getPlayers() == null) {
70              throw new GameWasNotInitialized("Players array");
71          }
72
73          // Minimum (based on players) and maximum map size and a range in string form
74          final int MIN_MAP_SIZE = (game.getPlayers().length <= 4 ? 5 : 8), MAX_MAP_SIZE = 50;
75          final String MAP_SIZE_RANGE = "(" + MIN_MAP_SIZE + "-" + MAX_MAP_SIZE + ")";
76
77          map = Map.getInstance();
78
79          // Loop until a valid map size is obtained
80          while (true) {
81              System.out.println("What will be the size of the map? " + MAP_SIZE_RANGE);
82              final int mapSize = game.getValidInt();
83
84              if (!map.setMapSize(mapSize, mapSize, game.getPlayers().length)) {
85                  System.out.println("The input value was out of the range " + MAP_SIZE_RANGE + ".");
86              } else {
87                  break;
88              }
89          }
90      }
```

Figure 12: MapCreator.java Coverage (2/2).

2 Design Details

The design for the basic version of the game (Figure 13) was based on the given initial class design in the second part of the assignment. Additional classes and new members in existing classes were added according to needs. In this section, some important deviations from the given initial design will be discussed.

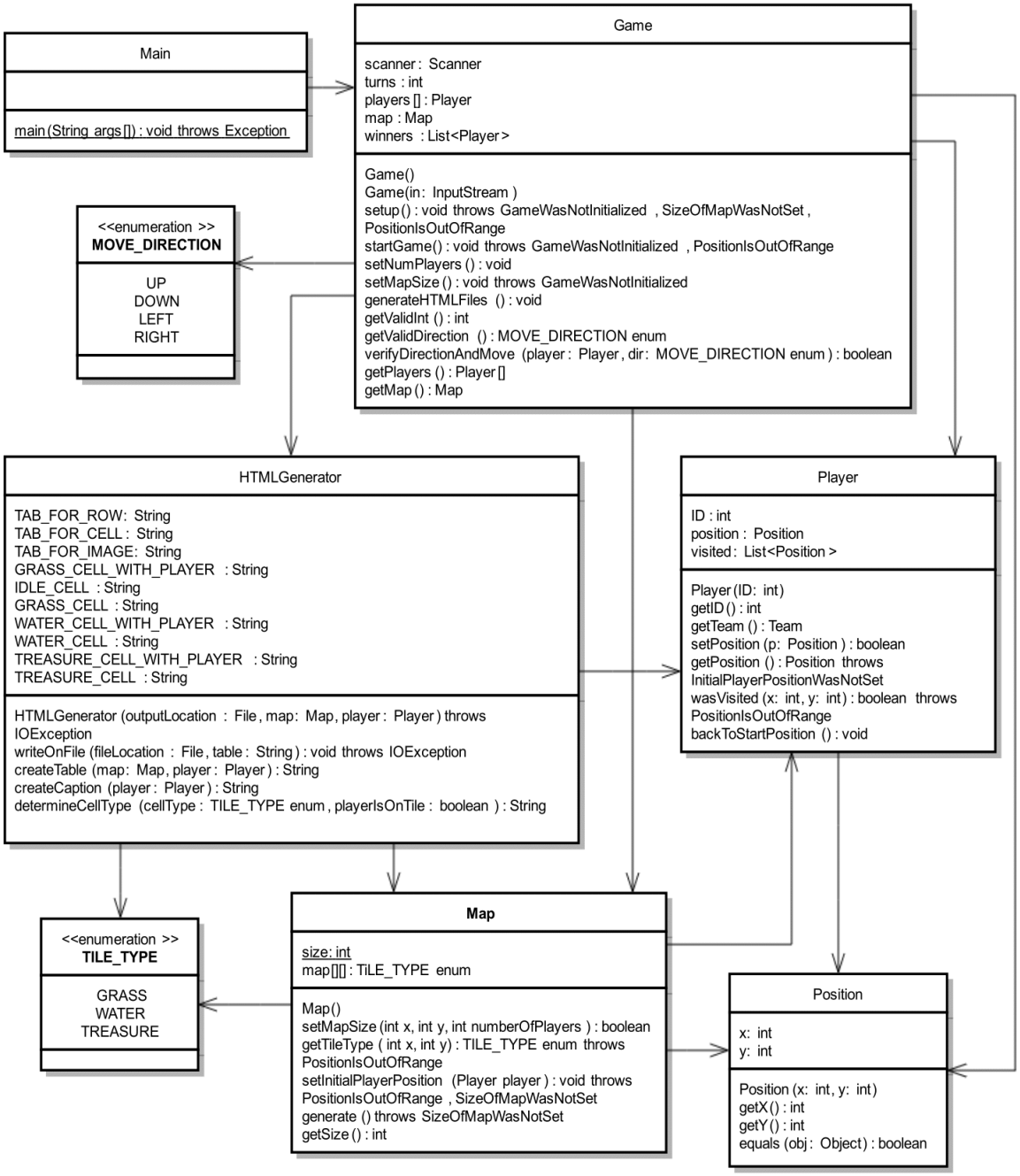


Figure 13: Class diagram of the Basic Version Game.

In the Game class, an optional input stream as a constructor argument was added, giving the ability to supply input to the game from a custom input stream. This is mostly important for the game tests. Additionally, various helper methods were added to distribute behavior across multiple methods, and getters were added to be used in the testing. In the Player class, a list of visited positions is used to indicate which map tiles should be uncovered for the player when the HTML file is generated. The wasVisited() method is used to check whether a specific position is in this list of positions. The first position in this list is assumed to be the players starting position. In fact, backToStartPosition() sends the player back to this position.

In the Map class, the setMapSize() is also given the number of players so that the minimum map size can be checked. The setInitialPlayerPosition() generates a random initial position for a given player, making sure that this is a grass tile. Some getters were also added. The getTileType() allows checking of the tile type at a particular position, for example to check whether a player landed on the treasure, whilst the getSize() getter is used to check coordinate limits, in the HTML generation, and in other situations.

The HTMLGenerator class is dedicated to generating HTML files for players. All methods in this class were made private, except for the constructor. This means that given an output location, the map, and a specific player, the constructor calls private methods to generate the HTML file for the specified player. This also means that an HTMLGenerator instance is created for each player in each round of the game.

Finally, note that two enumerations were created for convenience. MOVE_DIRECTION is mostly used by the Game and defines the directions that a player can move in, while the TILE_TYPE is used in the Map and HTMLGenerator and defines the types of tiles that can be present in a map.

3 Enhancements

3.1 Different Map Types

The main aim of this improvement was to add support for different map types in the game. Previously, there was only 1 map type that created a map with 1 treasure cell, 10% of the map size water cells and the remaining green cells. The design pattern used to handle this improvement is the factory method. The factory method design pattern was chosen since it allows support for the construction of different map types and can add more map types in the future by extending the Map class and overriding the methods. This design pattern allows code to be more elegant and can make efficient use of subclasses.

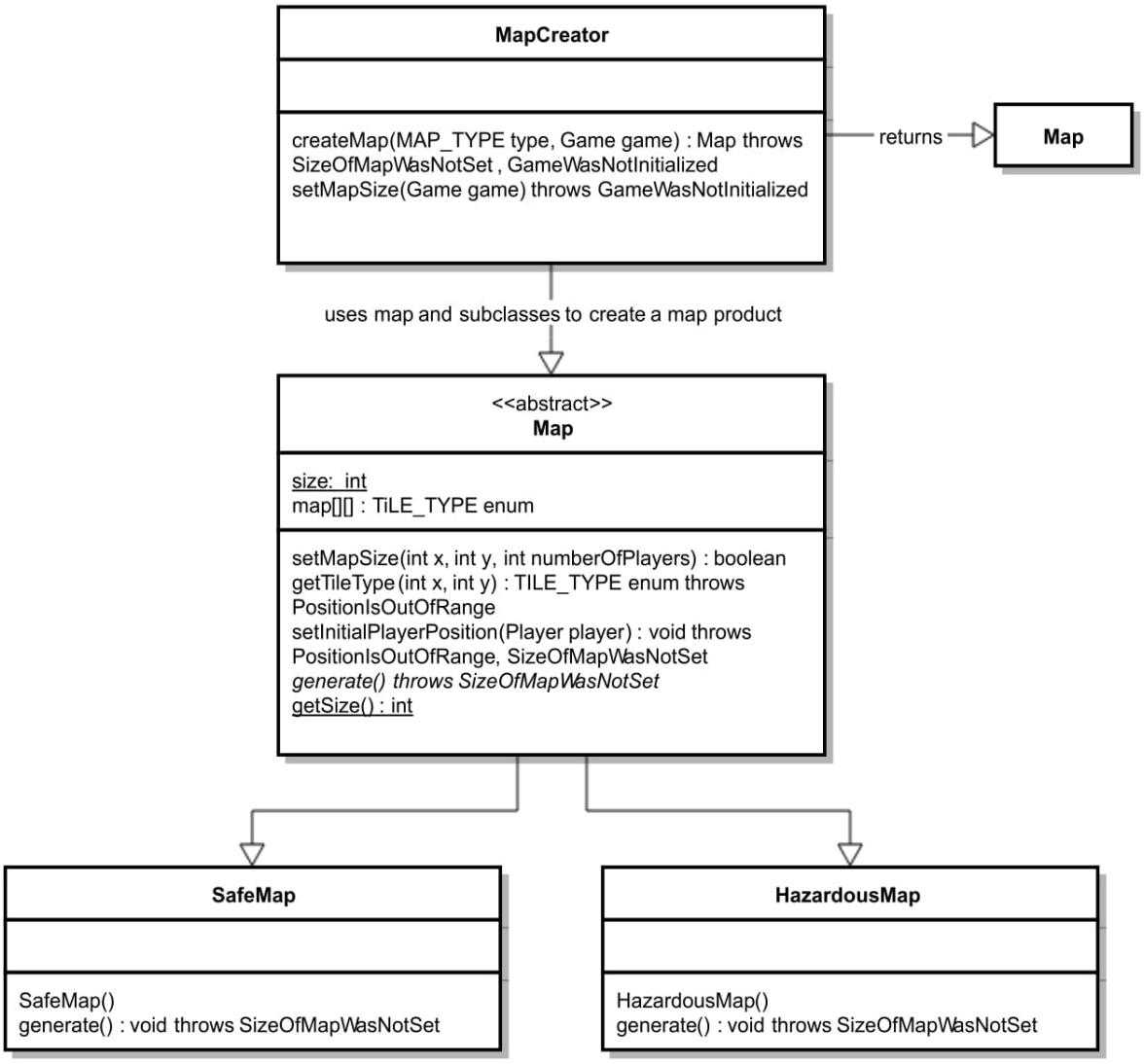


Figure 14: Factory method design pattern.

Figure 14 shows the class diagram used to implement the factory method design pattern. The map was declared an abstract class. Some of the methods used will be used by both subclasses and thus to remove code duplication they were put in the abstract class. The method generate() will be overridden by both methods. This method is used to differentiate the two maps. SafeMap will contain 10% water tiles while HazardousMap will contain between 25% and 35% water tiles. To create the Map, MapCreator is used. Depending on which map the user wants, the Game class will pass the type of the map that the user wants to MapCreator. MapCreator will then create an instance of the map needed, set the size of the map and generate the tiles in the map depending on the type. MapCreator will then return the instance of the map as a finish product. As it can be noted, MapCreator was made contrete since the creation of the maps were simple.

3.2 Store One Map in Memory

The next improvement is to allow only one instance of the map class to be created and be used by all the players currently playing. To achieve this, the Singleton design pattern is used. The reason behind using the Singleton design pattern was that with the Singleton design pattern it ensures that a class has only one instance at all time and provides a global method to access that instance.

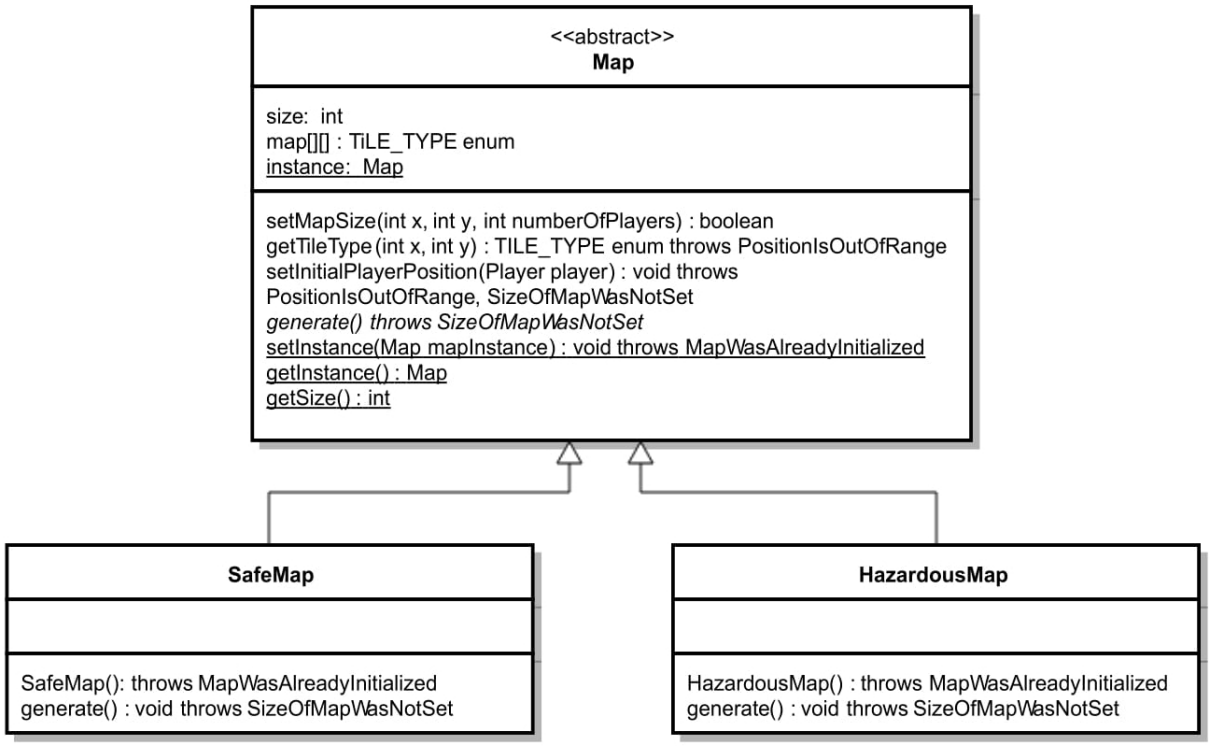


Figure 15: Singleton design pattern.

Figure 15 shows the updated class diagram. From the previous improvement, there are

now more than one map type. Thus to make the map singleton, an instance of type Map is held in the abstract class Map. No private constructor is required since the map is abstract. Two methods were added to Map to set the instance and to get the instance. SetInstance will throw an exception if there was already an instance beforehand, otherwise it will set the private static variable instance to the passed argument in setInstance. To create either SafeMap or HazardousMap, in the constructor the setInstance is called passing the object itself (this) to setInstance. This can cause either 2 things:

1. Map will accept the new instance and will set the instance to the passed argument.
2. Map will reject the new instance and will throw an exception cause the creation of a new instance to fail and deleting it. This way the code cannot create more than 1 instance of Map regardless of the map type and the code required to use getInstance to access the map.

To access the newly created instance, getInstance can be used from any class since its a static method. If no previous map instance was set beforehand, null is returned. GetInstance cannot create the map type since it cannot know which map type the user wants.

3.3 Team Exploration

The first observation that was made upon reading this required improvement was that in collaborative mode, the players will somehow need to share a list of visited positions. In order to avoid changing the Player class by too much and as a result having to change many other aspects of the game, it was decided that the player class should keep its list of visited positions.

Given this decision, another observation was now made. Since the players in a team will each keep a list of visited positions, they will somehow need to communicate positions between each other so that if a player discovers a new tile, all the other players in the team should know what this tile is.

After discussing various potential design patterns, the one that seemed to fit the criteria the best was the Mediator design pattern. This pattern is suitable since it encapsulates the communication between a set of objects (i.e. a set of players that make up a team) that are able to communicate without referring to each other explicitly. In this application of the Mediator pattern, teams will be Mediators, and players will be Colleagues that communicate only with one mediator.

Figure 16 shows the class diagrams used and it shows how the improvement was implemented, with respect to the Player class and the new Team class.

Among the changes, the Player class was given a team attribute, while the Team class maintains a list of players that are part of the team, playerList. In collaborative mode, an instantiated team is passed to the Player constructor, which calls the addPlayer() method in the team to subscribe to positions from the other team members.

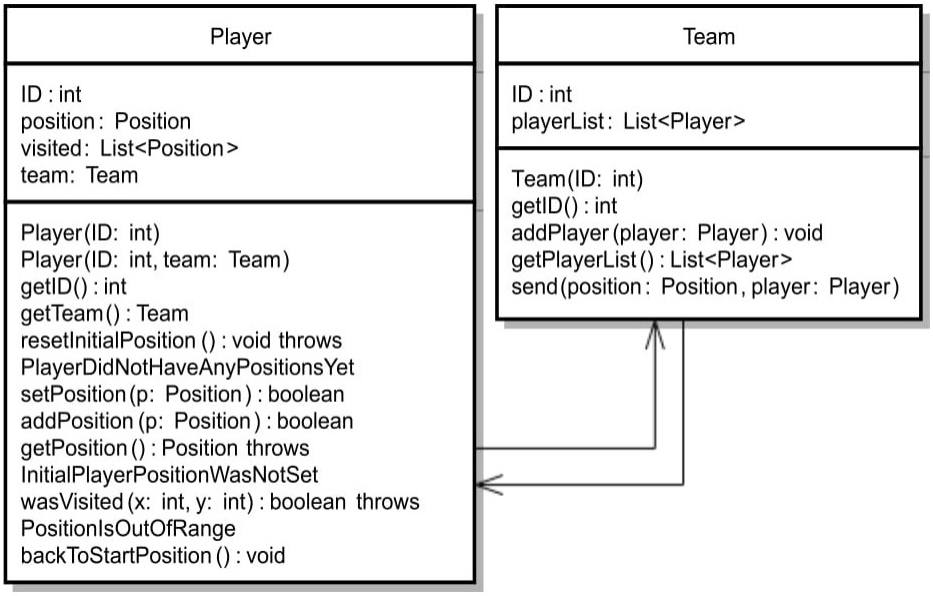


Figure 16: Mediator design pattern.

During the game, when a players setPosition(), is called, and if the player was assigned to a team, the new position is sent to the mediator using the send() method which broadcasts the position to all the team members using the addPosition() method which adds the position to the players individual list of visited positions without changing the current positions of the player.

Since the setPosition() method is used to set the initial position of a player, this means that a player object is likely to get at least one position from team members before its own initial position is set, which can disrupt the backToStartPosition() method. To fix this issue, the resetInitialPosition() is called for a player exactly after its initial position is set, so that this is placed as the first position in the visited list.

4 Game Visuals

Kindly note that for the "Store one map in memory" enhancement, this cannot exactly be shown using a screenshot. As was presented previously, since a singleton design pattern was used, it is guaranteed that there will only be one map instance.

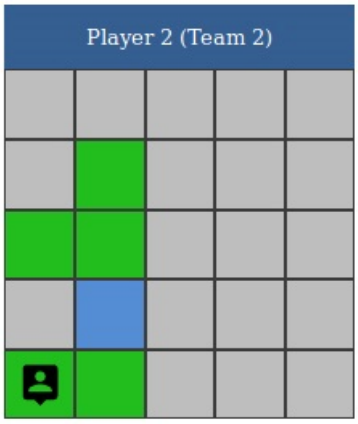
Figure 17: Basic Version of the Game.

(a) Safe Map. (b) Hazardous Map.

Figure 18: Different Map Types.

(a) 1st Player of Team 1.

(b) 2nd Player of Team 1.



(c) 1st Player of Team 2.

(d) 2nd Player of Team 2.

Figure 19: Team Exploration.

5 Configuring and Running the Game

Since maven is used for the project, the compilation and execution of the program were left to the project management and comprehension tool. For compilation, running the unit tests, taking the compiled code and package it into JAR file and installing the package into a local repository is done by executing the command:

$$\text{mvn clean emma:emma install} \tag{1}$$

Not that the coverage report is outputted and is found with the compiled code. To execute the compiled code, the following command is used:

$$\text{mvn exec:java -Dexec.mainClass=Main} \tag{2}$$

After the execute of the previous command, the settings on how the game will be set up will be asked to the user. The settings available are the following:

1. Play in collaborative mode or not collaborative mode (Required input 1 or 0 respectively). If the user wants to play as a team, an addition input will be asked for the amount of teams that will be playing (Required input 2 to 8).
2. The amount of players that will be playing (Required input from 2 to 8).
3. The type of the map which can be hazardous or safe (Required input 1 or 0 respectively).
4. The size of the map (Required input 5 to 50 if 2 to 4 players are playing otherwise 8 to 50).

When the configuration is completed, the players will required to every a direction (U, D, L, R) with every round until one of the player/s find the treasure cell. The visual maps for each player can be found in src/main/resources/players-map and they are updated with every round.