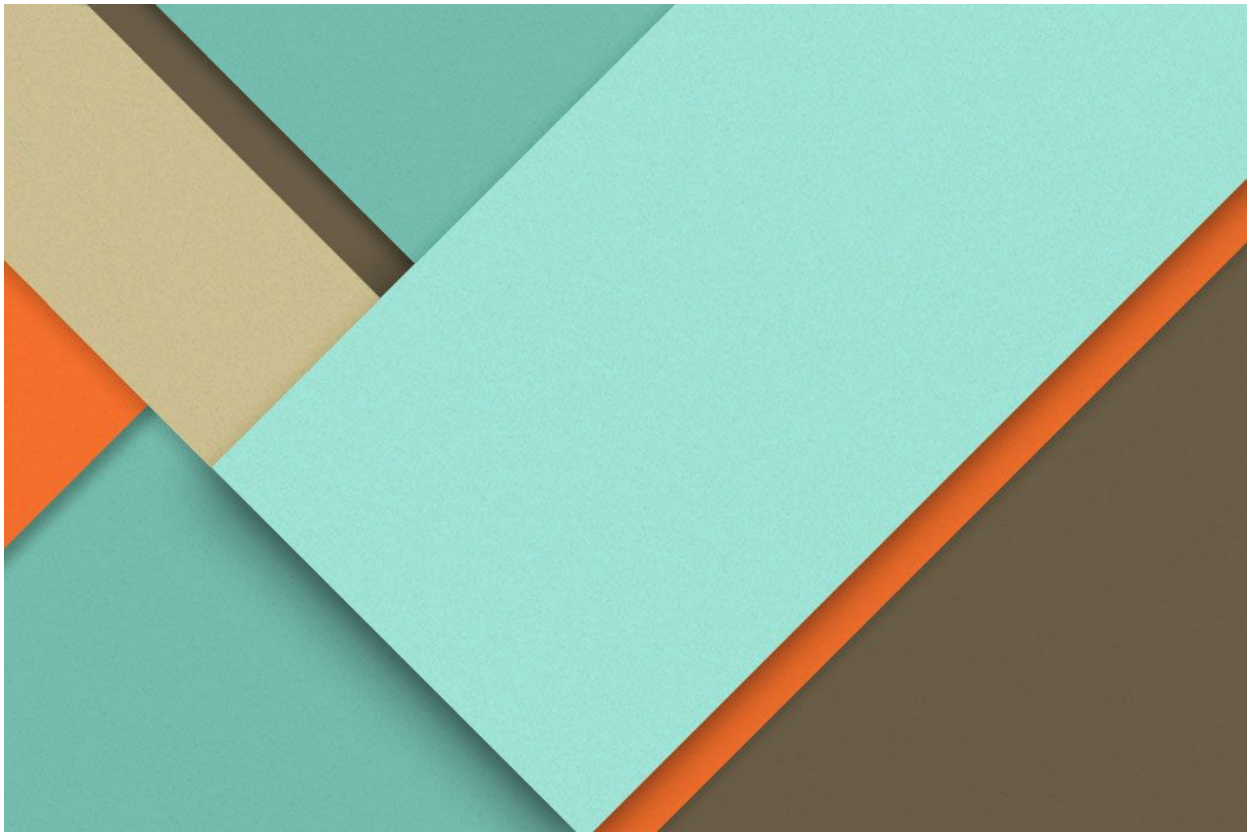

CPS2002- Software Engineering Assignment

29.04.2018

Dylan Galea
BSc. in Mathematics & Computer Science
Year 3
84296M

Gabriel Camilleri
BSc. in Computer Science
Year 2
484997M

Table Of Contents

Introduction

The aim of this group work assignment was to create a multiplayer treasure finder game using Git Flow, test driven development (TDD) and design patterns. Per [1], Gitflow makes parallel development easy by separating the development of new parts of the system from already finished features. This ease of parallel development is due to the creation of feature branches in order to work on a particular feature of the system. This makes it easier for a multiple of developers to even work on the same feature in collaboration. In fact, in this assignment a number of branches where created were almost in all of them parallel development was used. Per [2], test driven development is the technique to create 1 test that would test some function of the program. Then write enough code to the fail the test. Then enough code is written in order to pass the test and check that all previous tests pass. When all of these steps are ready, the process continues from step 1. In this assignment, test driven development was used through the production of the game as mentioned in this paragraph. Per [3], design patterns show a general solution to a design problem that occurs a lot in programming. These design patterns are split under 3 categories, Behavioural, Structural and Creational. In this assignment design patterns were used in part 3 of the assignment, were after the code in part 2 was released, the user requests some improvements on the released software.

This report contains 5 main sections, the first section is about code coverage analysis, where data about the area the tests cover in the code will be displayed. The second section provides the design details of the basic version of the game. The third section contains details about the design patterns used. The fourth section contains some screenshots about the basic version of the game and the enhancements .Finally , the fifth section contains instructions for running and configuring the game. The following section is about code coverage.

Code Coverage Analysis

Part 2:

The following images are of the overall coverages of the project, then by the package and then all the classes inside the packages (only the ones where at least a single line is not covered).

Figure A1- Code Coverage of Project & Packages

Part2_Muliplayer_Game:

Figure A2- Code Coverage of Part2_Muliplayer_Game's class

HTML_File_Gen:

Figure A3- Code Coverage of HTML_File_Gen's class

Tressure_Finder_Game:

Figure A4- Code Coverage of Tressure_Finder_Game's classes

In the class: GameLauncher.java we find that there are no tests, and no coverage is found. The reason behind this is that the code in Game Launcher has to do with user input which cannot be simulated and exceptions which are already addressed in the appropriate exception tests derived from their own class. This class only mainly saves user inputted

values, prints out statements and calls the game constructor of the game engine where the real logic is found.

Moving on, we find the HTML_Gen.java class. In this class we see that there is a method unused. This method is called displayFile(), its job is to output the html files onto a web browser. It is unknown how exactly to test that this is working unless you run it during the game. Thus it was decided to be skipped. Other than this there were two lines not tested. One of which, was the output of the title to the HTML player files when there is a game running in collaborative mode. There were so many tests regarding the single vs collaborative mode as well as HTML generation that it was felt there did not need to be another for a single line. The final line, was found in the method getTileColour, were the returning values where HTML code with the RGB hexadecimal numbers in string form. The cases would be filled in 4 colours, when the tile is not visited, when the tile is visited and: is a grass tile, a water tile and a treasure tile. If somehow it would not return any of these which would be impossible, it would return a blank string, which is this untested line. It is said to be impossible as in map tests we have tests which show maps adding together grass, water and treasure would make the whole map, thus there cannot be another tile property, and of course one cannot have something other than visited or not visited, under the latter property.

FInally in the class GameEngine.java we find 3 methods and 60 lines not covered by the tests. Starting from the method getMoveFromUser uses user input thus it is quite difficult to test and all this method was not covered. The moving functions however have been tested in their according test classes. Another method is generateFiles, here the use of this method is to create and put player HTML files in the directory, this also cannot be thought of as to how to test this method. Finally we have startGame method, which is split into 2 parts, the initialisation and the playing of the game inside the winning conditions. The problem with this is the method is made up of printing statements and user input both of which are harder to test also some tests have been done on similar code, such as a test on if the winning condition is met. The reason why this is not shown to be covered is that since we cannot set the map size as users, we instead create our own map and test if the winning conditions are met in that way. However it would not use this method but lines of this method are copied to be tested.

As one can see most of the reasons of above include either cases of user input, print statements and displaying or creating files, thus are hard or impossible to test under JUnit tests, or there happen to exist tests on identical code found in other classes, thus would only be used to cover a single line or two.

Part 3:

The following images are of the overall coverages of the project, then by the package and then all the classes inside the packages (only the ones where at least a single line is not covered).

Figure A5- Code Coverage of Project & packages

Part2_Muliplayer_Game:

Figure A6- Code Coverage of Part2_Muliplayer_Game's classes

HTML_File_Gen:

Figure A7- Code Coverage of HTML_File_Gen's classes

Tressure_Finder_Game:

Figure A8- Code Coverage of Tressure_Finder_Game's classes

Tressure_Finder_Game_Maps:

Figure A9- Code Coverage of Tressure_Finder_Game_Maps' classes

In the class: GameLauncher.java we find that there are no tests, and no coverage is found. The reason behind this is that the code in Game Launcher has to do with user input which cannot be simulated and exceptions which are already addressed in the appropriate exception tests derived from their own class. This class only mainly saves user inputted values, prints out statements and calls the game constructor of the game engine where the real logic is found.

Moving on, we find the HTML_Gen.java class. In this class we see that there is a method unused. This method is called displayFile(), its job is to output the html files onto a web browser. It is unknown how exactly to test that this is working unless you run it during the game. Thus it was decided to be skipped. Other than this there were two lines not tested. One of which, line 128, was the output of the title to the HTML player files when there is a game running in collaborative mode. There were so many tests regarding the single vs collaborative mode as well as HTML generation that it was felt there did not need to be another for a single line. The final line, Line 183, was found in the method getTileColour, were the returning values where HTML code with the RGB hexadecimal numbers in string form. The cases would be filled in 4 colours, when the tile is not visited, when the tile is visited and: is a grass tile, a water tile and a treasure tile. If somehow it would not return any of these which would be impossible, it would return a blank string, which is this untested line. It is said to be impossible as in map tests we have tests which show maps adding together grass, water and treasure would make the whole map, thus there cannot be another tile property, and of course one cannot have something other than visited or not visited, under the latter property.

In the class MapCreator.java we find there is a single line not covered. Similar to the previous untested line, this one in a default state. A type is either “Safe” or “Hazardous” and is handled accordingly, if somehow an input not between these two is given it calls the default which will create a new map under the “Safe” Type. Again a test for a single line which would normally not be passed through felt useless.

In the class SafeMap.java we find 2 consecutive lines not covered. The reason behind these is that another class HazardousMap.java contains similar code and identical at these parts, and have already tested on these 2 lines, thus it would have been lacking substance to test for the Safe version given the same would be expected.

FInally in the class GameEngine.java we find 3 methods and 60 lines not covered by the tests. Starting from the method getMoveFromUser uses user input thus it is quite difficult to test and all this method was not covered. The moving functions however have been tested in their according test classes. Another method is generateFiles, here the use of this method is to create and put player HTML files in the directory, this also cannot be thought of as to how to test this method. Finally we have startGame method, which is split into 2

parts, the initialisation and the playing of the game inside the winning conditions. The problem with this is the method is made up of printing statements and user input both of which are harder to test also some tests have been done on similar code, such as a test on if the winning condition is met. The reason why this is not shown to be covered is that since we cannot set the map size as users, we instead create our own map and test if the winning conditions are met in that way. However it would not use this method but lines of this method are copied to be tested.

As one can see most of the reasons of above include either cases of user input, print statements and displaying or creating files, thus are hard or impossible to test under JUnit tests, or there happen to exist tests on identical code found in other classes, thus would only be used to cover a single line or two.

Design of Basic Version of the Game

The UML diagram of the Basic prototype of the game (i.e. part 2 of the assignment) can be seen in the figure below:

The diagram was split in 4 due to the size of the diagram

Figure B1-Part 1 of UML Diagram

Figure B2-Part 2 of UML diagram

Figure B3-Part 3 of UML diagram

Figure B4-Part 4 of UML diagram

As it can be seen from figure B1 above , this implementation was designed to have a number of exception classes . These exception classes indicate the type of error that caused the exception . In fact , 3 types of exceptions were created for the basic version of the game. These were InvalidCharacterInputMoveException , InvalidMapSizeException and InvalidNumberOfPlayersException. These classes generate an exception whenever the user enters an invalid move character in console, map size not according to the specified range and the number of players is not between 2 and 8 respectively.

In addition to the exception classes , the TreasureFinderPlayer class was created. This class encodes the player playing the treasure game . As seen in figure B4 above the player has an isVisited boolean 2D array the same size of the treasure game map . This array stores true in position (x,y) if the tile at position (x,y) is visited in the map , and false otherwise. This would be easy for the html generator to check which tiles are visited or not (i.e for colouring the map).

As can be seen in figure B2 , in this implementation the map is also a 2D array of characters. The characters permissible in this map are G,W and T which represent a green,water, treasure tile respectively . In this implementation the map is generated using the generateMap() function which generates the map randomly. In this implementation the generateMap() function works in the following way :

- Fill the whole array with G tiles
- Pick randomly a tile and set is as T (important not on the border, to not be surrounded by water tiles)
- Randomly put water tiles (not more than 7 , so as to not surround the treasure by water tiles) on green tiles.

As seen in figure 4 above , the game engine has a number of validation functions such as the validMapSize function . In this implementation these validations are important because the game does not start if the number of players and map size are not according to the assignment specification sheet . In fact the user is asked to enter the number of players and the map size in the game launcher and the game engine constructor throws an exception if it is initialized with incorrect values . This could have been done entirely in the game engine class , however it was not done in this way in order to have higher code coverage in the game engine class. After the game engine class is initialized correctly , the game would start after the game launcher invokes the StartGame method. This method would then initialize the players in the game and get moves from the user , along with outputting the map to an html file.

Design Patterns Details

Enhancement 1:

For the first enhancement , a design pattern had to be chosen in order to support different map types in the game . For the current implementation only 2 different map types had to be implemented , however the design pattern used had to be chosen carefully in order to allow for different map types in the future . As a result the Factory Method design pattern was used . The Factory Method design pattern was chosen because it is the most ideal design pattern to use when a class cannot anticipate the class of objects to be created and it must allow support for different types of classes of objects.

In order to implement the factory design pattern , first the map created in the basic implementation version of the game was modified to be an abstract class . The class diagram for this class can be seen in figure C1 below:

Figure C1-Map class Diagram

As seen in figure C1 above , the class map has 3 fields storing the size of the map , tiles array and the type of the map. Along with the setters and getters , the class Map has an abstract method called generateMap that will generate the map according to the type of the map . Thus this method has to be implemented by the concrete classes of the Map hierarchy , since these would contain the specific implementation of the type of the map.

After the class Map was created , the concrete classes SafeMap and HazardousMap that extend Map were created. These classes contain the specific implementation of the generateMap method. For example in figure C2 below , the implementation of the generateMap for the HazardousMap class can be seen . In this implementation there are 25%-35% of water tiles created which is a different implementation than that of a SafeMap.

Figure C2-Implementation for generateMap

After the Concrete Map classes were implemented the concrete MapCreator class was created. The implementation of the MapCreator class can be seen in figure C3 below:

Figure C3-Implementation for MapCreator class

Thus referring to figure C3 above , whenever the game engine needs a map , the MapCreator class will be called , where the method createMap will be called , taking the type of the map to be created and the size of the map . Then as seen in figure C3 line 12 above the map creator gets an instance of the specific map creator for that type passed by the user and calls the createMap method of that child class and returns the newly created map.

Finally , the last classes that were created were the specific creator classes , namely the SafeMapCreator and HazardousMapCreator classes . These classes create a product of type SafeMap or HazardousMap respectively. The implementation for the SafeMapCreator can be seen in figure 8 below (note that the HazardousMapCreator class has a similar implementation).

Figure C4-Implementation for SafeMapCreator class

As shown in figure C4 above , whenever the createMap for the SafeMapCreator is called , it calls safeMap's generate function which randomly generates the map and then returns the safeMap as shown in line 17 . Note that line 15 refers to the second enhancement ,i.e not this design pattern.

In order to create a map to be used in the game then it was required to call the mapCreator from the game engine class passing the type and size of the map as parameters to the constructor. This code can be seen in figure C5 below :

Figure C5-Creating a map in the game engine class

Enhancement 2:

In the next enhancement , a design pattern had to be chosen to store only one instance of the map in memory where each player still sees the parts that he has explored only . It must be noted that in the basic version of the game , the game engine was already designed to only have one map in memory , where using the isVisited array in each player , the player can only see his explored part of the map . However, what was not done was to ensure that inthe game engine no more than 1 map can be created in memory . Thus , the Singleton design pattern was chosen for this enhancement . The singleton design pattern was chosen because it is the ideal pattern to use when only one instance must be created for a particular object.

Since the Map class was designed to be abstract in this implementation , the singleton design pattern could not be used on the map class. Thus it was used on the concrete classes SafeMap and HazardousMap . It must be noted that in this implementation 2 maps could still be created where one is hazardous and one is Safe , however since the map type is passed from the game launcher , the game engine can never create maps of different types , thus the implementation of the design pattern is correct.

The implementation of the singleton design pattern could be seen in figure C6 below , where the HazardousMap class is demonstrated .(Note the same is done in the SafeMap class).

Figure C6-HazardousMap class implementation

Thus as seen from figure C6 , a static-private instance of the Hazardous map was created and set to null .Also , as seen from line 19 the constructor is set to private , this enables no direct creation from outside the class . Thus no class can create multiple objects of type HazardousMap. However when a class needs to create a HazardousMap , it uses the getInstance method shown in line 13 where this method creates a new instance of the HazardousMap class if the map is not created , or it would return the already defined instance if it has already been created. Thus in this way only one instance of the class Hazardous map could be created . The SafeMap class has the same implementation so that no one can create multiple instances of the safe map .

The code used in order to create a safe map or hazardous map can be seen in figure C7 below :

Figure C7-HazardousMap use of getInstance method to create a hazardous map

Enhancement 3:

For this enhancement , a design pattern had to be used in order to make the game playable single mode or collaborative mode in teams . The team assignment had to be randomly generated and the members of the team see the tiles uncovered by all team members. In order to implement this enhancement the design pattern that was used was the Observer design pattern. The observer design pattern was decided to be used because in this problem there is a one to many dependency between the isVisited boolean 2D array and the members of the team . Thus whenever there is a change in the tiles uncovered by a member of the team all the members of the team must be notified . This makes the observer design pattern the ideal candidate to be used . Another reason why the observer design pattern was used because the observer design pattern is ideal when the size of the team is not known.

In this implementation the a Subject class was created , this Subject class was declared as the parent root of the class TeamSubject . The Subject class diagram can be seen in figure C8 below :

Figure C8-Subject class diagram

As seen in figure C8 above , the Subject class diagram contains a list of observers that will be observing some concrete subject . Whenever an observer needs to be added to the list of observers the attach method is used , and whenever an observer needs to be removed from the list of observers the detach method is used . The Notify method shown in figure C8 above is then used in order to notify the observers that a change in state has occured in the concrete child subject. In this implementation the concrete child subject was called the TeamSubject , where the state that it is observed on is the isVisited array where whenever a player modifies the isVisited array related to a team , the players in the same team are notified and their isVisited array is changed to that stored in the TeamSubject. Thus this makes the concrete observer ideal to be the player. Figure C8 below shown the Notify method to be implemented in the Subject class so that it would be present in all classes.

Figure C9-Notify Implementation

As it can be seen in figure C9 above the notify method is used to call the method update of each observer in the observer list . The update method would then update the state of the observer to that stored in the TeamSubject.

The figure C10 below shows the implementation of TeamSubject

Figure C10 - TeamSubject implementation

As shown in figure C10 above the TeamSubject class stores the mapState.This boolean 2D array is updated by the player whenever the player moves to a new tile. Whenever the

player moves to a new tile , the player also sets the team's state to uncover the tile that he explored using the method setState in figure C10 above . As seen in line 38 above , whenever this method is called , the notify method is then called to update the state of all the players in the team , so that the team players can see the player's move update.

The calling of the setMethod can be seen in figure C11 below :

Figure C11 - setState call

As seen in figure C11 above , this calling of setState could be possible because each player(which in this implementation is the observer) contains an instance of the team he is assigned to and thus calls his team's setState method. It must also be noted that if tSubject is null then the game mode is single , thus the observer design pattern is not in use as shown in line 65 above .

The observer implementation can be seen in figure C12 below :

Figure C12 - Observer implementation

As it can be seen above the abstract class Observer only has the abstract method update , that will be used in the concrete observer (Player) to update his state. The update method for the child TreassureFinderPlayer can be seen in figure C13 below:

Figure C13 - update implementation for TreassureFinderPlayer

Thus as seen in figure C13 above the TreassureFinderPlayer's update method takes the state stored in the team and stores it in the isVisited state , so that the tiles that were uncovered by the player's team mate can be viewed by the player.

After the observer design pattern was implemented , there was remaining to be used in the game engine .In the game engine a list of TeamSubjects was created in order to to represent different teams in the game . These subjects would then be randomly assigned the different players in the game . Note that in this implementation if 4 players are to play the game then 2-3 teams could be present since an empty team is not allowed . Also one player in each team is not allowed since this would essentially mean that single mode is played. Assigning of teams happens in the following way :

- Fill team i with player i
- If i>number of teams , randomly assign the remaining players in a team otherwise repeat from bullet point 1

Screenshots of the Basic Version of the Game & Enhancements

The next subsection contains screenshots about the basic version of the game

Screenshots of the Basic Version of the game

Figure D1 - Start of Game

Figure D2 - Output map for player 1

Figure D3 - Output map for player 2

Figure D4 - Output map for player 2

Figure D5 - Asking users for more input

Figure D6 : When player wins game

Figure D7 : Declaration of Winner on Screen

Screenshots of Enhancements

The screenshot in figure D8 below asks the user whether he wants a safe or hazardous map , this represents the first enhancement.

Figure D8: First enhancement , asking which type of map the user wants

The screenshot in figure D9 below asks the user whether he wants to play in single mode or in collaborative mode , this represent enhancement 3 . Note that enhancement 2 cannot be viewed in the game since this is more memory oriented.

Figure D9- Shows the choice given for the user to play in collaborative mode or single mode

The screenshots in figure D10,D11,D12,D13 below shows that the players assigned to the same team can view the same visited tiles.

Figure D10: Map for player 1 in team 1

Figure D11: Map for player 2 in team 2

Figure D12: Map for player 3 in team 2

Figure D13: Map for player 4 in team 2

The screenshots in figure D14, D15, D16 and D17 show the different maps when some team wins the game

Figure D14: Map for player 1 in team 1

Figure D15: Map for player 2 in team 2

Figure D16: Map for player 3 in team 2

Figure D17: Map for player 4 in team 2

The screenshot in figure D18 below shows the correct declaration of the winners of the game

Figure D18: Declaration of winners

Instructions for Configuring and Running the Game

1. Go to the Git Repository using the following link: ***github.com/dillu24/CPS2002_SE_GD***
2. Ensure you are in the proper branch, by pressing the “Branch” menu, and “Master”

Figure E1: Screenshot for finding the proper release

3. Now, go to the “Clone or Download” button and press “Download ZIP”

Figure E2: Screenshot for downloading the project from the github repository

4. Extract the zip folder, and open the Command Prompt.
5. Change to the director and run the Compile.bat file with the commands “cd <directory>” & “Compilation.bat”

Figure E3: Screenshot of the open Command Prompt with the command to change directory.

6. (If that doesn’t work ensure the PATH is set to the computer’s java jdk bin file thus locate it and use the command “set PATH = <Java JDK Bin Directory>;” and repeat step 5)
7. Now run the Run.bat file to begin the game, using the command “Run.bat”.

Conclusion

It could be concluded that the aims of the assignment were achieved . All enhancements required by the user were successfully implemented using a design pattern. Observing the github repository @CPS2002_SE_GD it could be also concluded that the software was also successfully implemented using test driven development and gitflow . Feature branches were created in order to implement different features of the basic game and these were all merged in the branch develop . After all feature branches were ready a release branch was created and some minor bug fixes along with documentation were prepared. For the third part of the assignment 1 hotfix named MapTypes was used , note that initially it was planned to use 3 hotfixes , one for each enhancement , however this was not allowed in git and thus the same hotfix was used for all enhancements. Hotfixes were used instead of feature branches because the software was already in live production.

References

[1] “Introducing GitFlow,” [Online]. Available: https://datasift.github.io/gitflow/IntroducingGitFlow.html.

[2] scottwambler, “Introduction to Test Driven Development (TDD),” [Online]. Available: http://agiledata.org/essays/tdd.html.

[3] O. E. Gabry, “Object-Oriented Analysis And Design — Design Patterns (Part 7),” 19 March 2017. [Online]. Available: https://medium.com/omarelgabrys-blog/object-oriented-analysis-and-design-design-patterns-part-7-bc9c003a0f29.