anOtherGame

System Design Document

Version 1.0
2017-05-28

Miranda Bånnsgård
Amanda Jonsson
Maja Nyberg
Allex Nordgren

This version overrides all previous versions.

Table of contents

1 Introduction

This is a document that describes the construction of The Lost Kitten application specified in the documents and analysis document.

1.1 Design goals

We have been developed the application towards two main design goals.

- The design must be testable, which means that it should be possible to isolate parts of the model for testing.

- The controller and graphical user interface should not be dependent on the model. They should be exchangeable and open for future development.

1.2 Definitions, acronyms and abbreviation

- MVC - Model-View-Controller pattern. This design pattern is used to separate concerns of the classes. The model contains the logic, the view represent the visualization of the application and the controllers controls the data flow and translates the user's interaction with the view into actions which the model then will operate.[1]

- JavaFX - A software platform for developing desktop applications.[2] It is graphics packages and media packages that enable the developer to design, create, test and debug.[3]

- Gradle - Gradle is an open source build automation system.[4]

- Open-Closed-principle - A design pattern that means that system design should be open for extensions and closed for modification.

- Eventbus - Simplifies the communication between components in the application by allowing publish-subscribe-style communication between the components.

See the RAD for definitions.

[1] Design Patterns-MVC, https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm, 2017-05-16
Java SE Application Design With MVC,
http://www.oracle.com/technetwork/articles/javase/index-142890.html, 2017-05-16

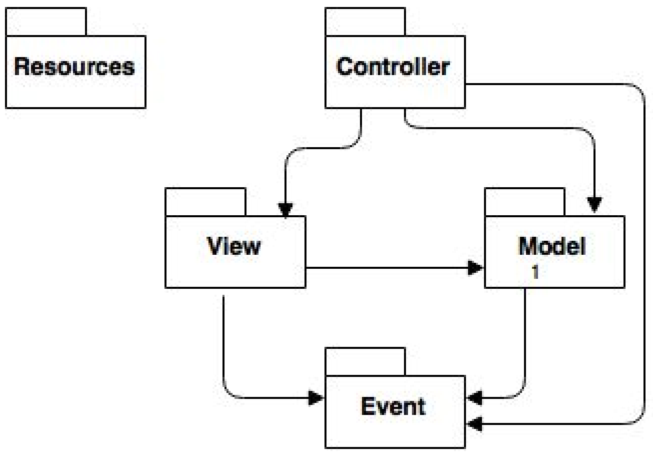
[2] JavaFX, https://en.wikipedia.org/wiki/JavaFX, 2017-05-15

[3] What is JavaFX?, http://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm, 2017-05-16

[4] Gradle, https://en.wikipedia.org/wiki/Gradle, 2017-05-22

2 System architecture

The application is a desktop application and it runs on a single computer.
The application is divided into four top level packages. The arrows represent the dependencies.



Resources - contains mostly pictures and FXML files which are used in the application.
Controller - contains all the controllers.
View - contains some view classes representing the GUI.
Model - contains the object-oriented model, the logic.
Event - contains classes that handles the events and the eventbus.

The controller and view package has references to both View, Model and Controller. The Model package has no references to either of the Controller or View packages. The interfaces and abstract classes should be open for extensions and closed for modifications, and thereby implemented according to the open-closed principle.

2.1 General observations

2.1.1 The aggregate class

We have an aggregate class which is the TheLostKitten class. This class is responsible for the game logic and handles a player's turn. All calls from the controllers will pass through the TheLostKitten class.

2.1.2 Interfaces

We have implemented against interfaces. We have seven interfaces, almost one per class in the model. This is a way to make the code open for extension but closed for modification. This is the design pattern open/closed-principle.

2.1.3 Unique JavaFX names

All labels, textfields and buttons, for example, need to have unique JavaFX id’s, because they are referred to in many classes.

2.1.4 Spaces

There are three different types of spaces in the application. The ones that are called the stations, which a player can be positioned at and stations also have a marker. Then there is the ones that are called spaces. These are not stations but a player can be positioned at a space but they do not have a marker. There is a type of space which is just a visualization to mark how the stations and spaces are connected. A player can not be positioned at these ones. The stations and spaces are stored in two lists: List<ISpace> **spaces** and List<IStation> **stations**. All stations are also contained in the list of spaces. A station has its name as the main identity and a space has its coordinates as main identity. Since a station is also a space, the coordinates can be used as identifiers for these as well.

2.1.5 Concurrency issues

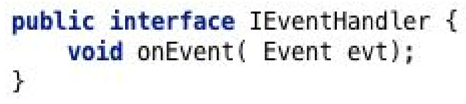
The prestanda of the game is slowed down since the map with all stations, spaces and markers is created when the the user pushes the “Starta”-button. This is something that could be fixed at a later time.

2.1.6 JavaFX Controllers

It is a bit tricky to apply the design pattern MVC when you use JavaFX. The view is represented by fxml-files which are connected to a specific controller. Since the fxml-files are not java-classes you can not add more java code to the view that are represented by the fxml-files. Due to this we have methods and java-code in our controllers that actually is view. We are aware of this but found that this was the most applicable way to to it. However, we have the most important aspects of the MVC-pattern as the Model package has no references to either of the Controller or View packages.

2.1.7 EventBus

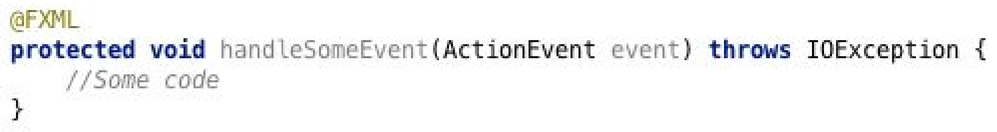
The application uses an eventbus. Updates of variables will be handled by setters. Setters will trigger the state change via the eventbus. The application are using this interface.



```
public interface IEventHandler {
    void onEvent( Event evt);
}
```

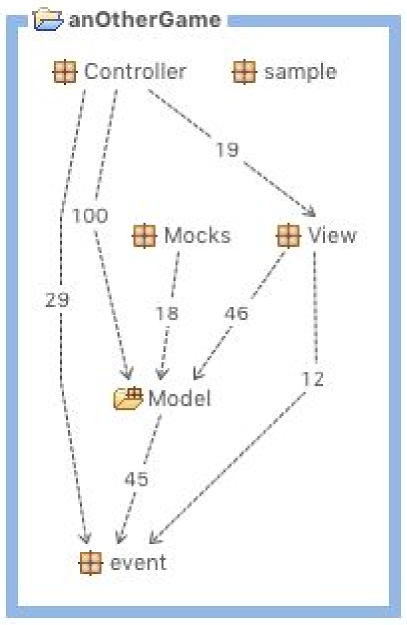
[5]

User input will be handled by JavaFX handle methods, which looks like this:



```
@FXML
protected void handleSomeEvent(ActionEvent event) throws IOException {
    //Some code
}
```

2.2 Dependency analysis



Using the tool Stan4j we could create this dependency schema. The nodes represent the packages in our application and the edges represent the dependencies. We are using the design pattern MVC which can be seen in the diagram. Model has no dependencies to any other packages besides the event package. This dependency

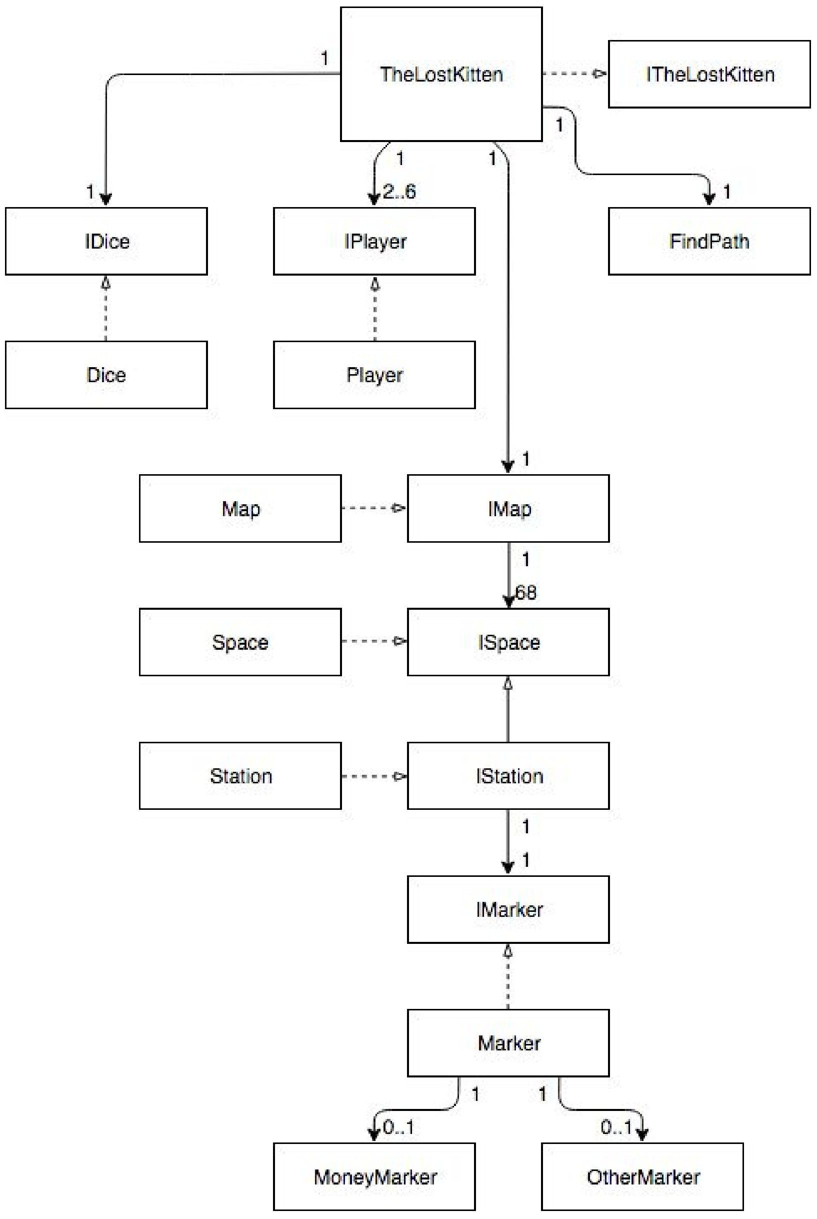
[5] http://www.cse.chalmers.se/edu/course/tda367/#lectures

analysis is very similar to the one we did ourselves further up, but in this one we can distinguish more details about the dependencies.

3. Subsystem decomposition

3.1 Model

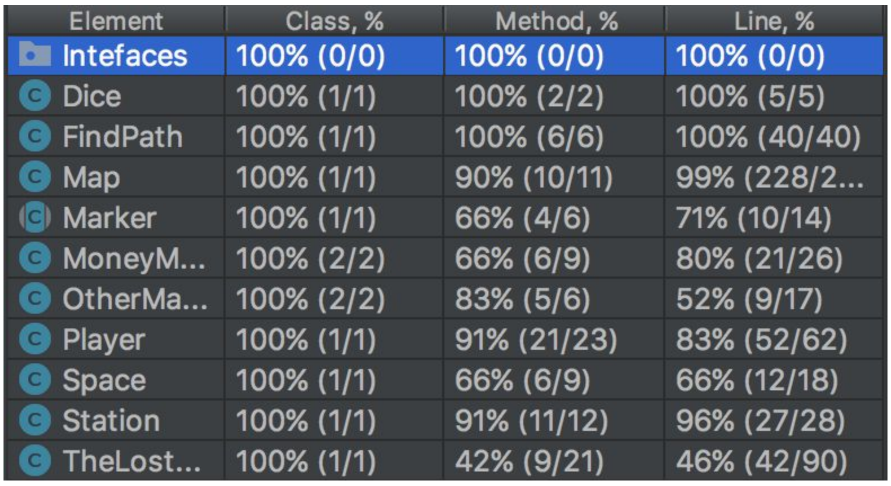
The Model package design model:



The Model, as mentioned further up, contains the object-oriented model which is the logic of the game. The model do not know about the view and the controllers, which means that there is no instances of the view classes or the controller classes in the model.

Since our design must be testable we have implemented a Test-class for each class in the model. This means that we have the tests: DiceTest, FindPathTest, MoneyMarkerTest, OtherMarkerTest, PlayerTest, SpaceTest, StationTest, Maptest and TheLostKittenTest. Each test is testing many methods that are implemented in the class that is tested. These tests can be found in the Test-package, anOtherGame/src/test/java/Model.

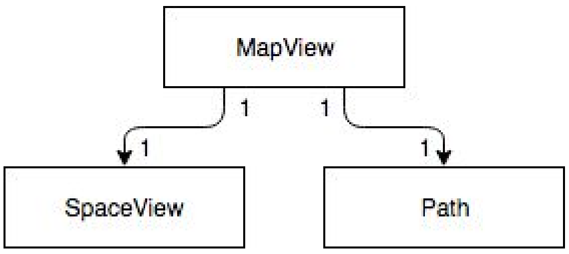
In the picture below, the test coverage of this project is visible. Some model classes are dependent on other models classes, for example TheLostKitten that uses the other classes like Player etc. Therefore we wrote tests on the more fundamental Models like Player, Dice, FindPath etc.



Element	Class, %	Method, %	Line, %
Intefaces	100% (0/0)	100% (0/0)	100% (0/0)
Dice	100% (1/1)	100% (2/2)	100% (5/5)
FindPath	100% (1/1)	100% (6/6)	100% (40/40)
Map	100% (1/1)	90% (10/11)	99% (228/2...
Marker	100% (1/1)	66% (4/6)	71% (10/14)
MoneyM...	100% (2/2)	66% (6/9)	80% (21/26)
OtherMa...	100% (2/2)	83% (5/6)	52% (9/17)
Player	100% (1/1)	91% (21/23)	83% (52/62)
Space	100% (1/1)	66% (6/9)	66% (12/18)
Station	100% (1/1)	91% (11/12)	96% (27/28)
TheLost...	100% (1/1)	42% (9/21)	46% (42/90)

3.2 View

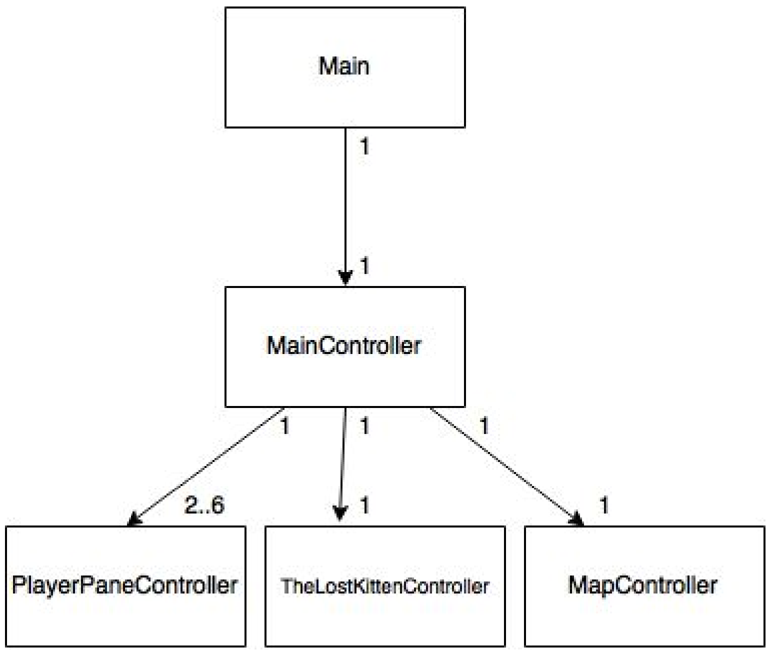
The View package design model:



The view recognizes actions on the gui, for example if a button is pushed. The view will then call a method in the controller that will handle the action. Since we are using JavaFX, most of our view consists of fxml-files, which are connected to a controller that handles the actions that the fxml-files recognizes.

3.3 Controller

The Controller package design model:



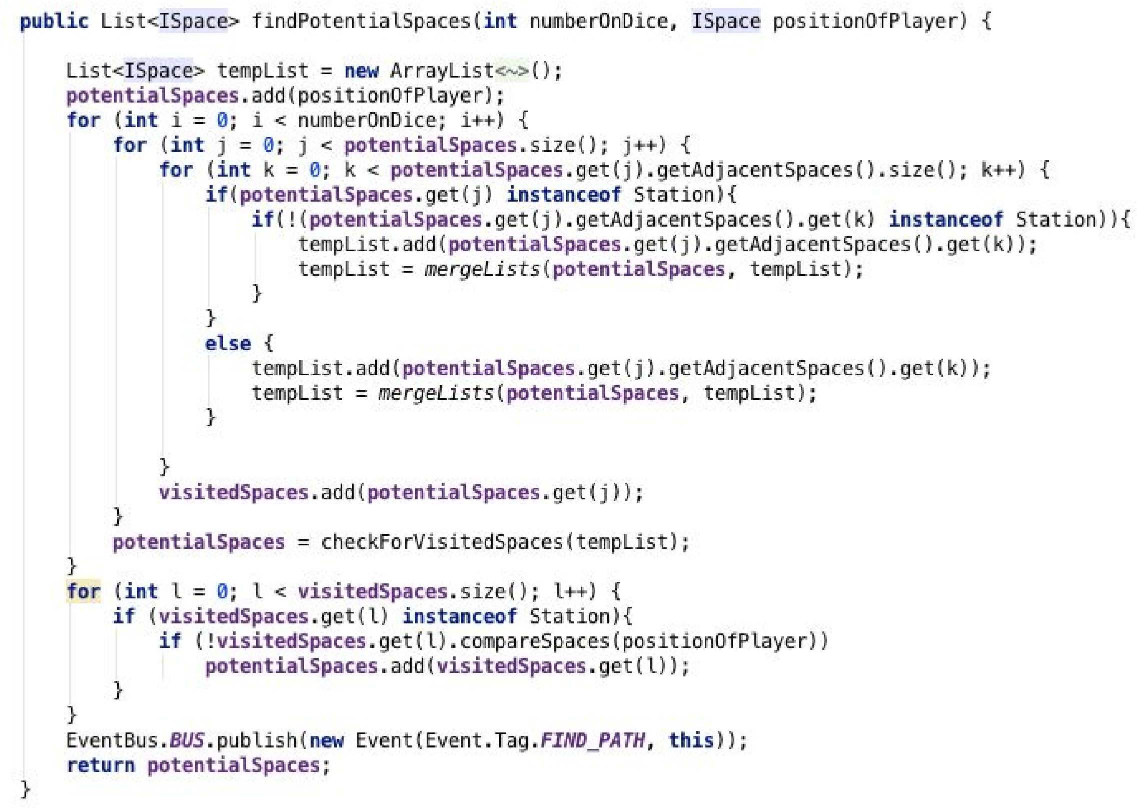
Main is the controller that starts the whole application. But it does not create an instance of the maincontroller but instead we are using the inbuilt method launch() that is included in the JavaFX library. You can use this to start a standalone application. We invoke the launch-method from the main-method and it is only allowed to invoke the launch-method once otherwise an exception will be thrown. The method does not return until the application is exited.

3.4 Model-View-Controller

This project uses the design pattern Model-View-Controller (MVC). The user of the application will activate, by for example pushing a button, a controller. Subsequently the controller will invoke a method in the model and by that change state. The model itself does not undertake actions, it only approves commands from the controllers and do what the method should do.

3.5 Find path algorithm

In the application there is a method that calculates which spaces on the map a player can go to after the player has rolled the dice. This method is rather complex. The method takes in two parameters, the result from the dice and the position of the player. By means of these two parameters the algorithm calculates which spaces the player can go to depending on what result the dice gave and where the player is positioned. What makes it even more complex is that a player can stop on a station and not go all steps that the dice gives. A part of the algorithm looks like this:



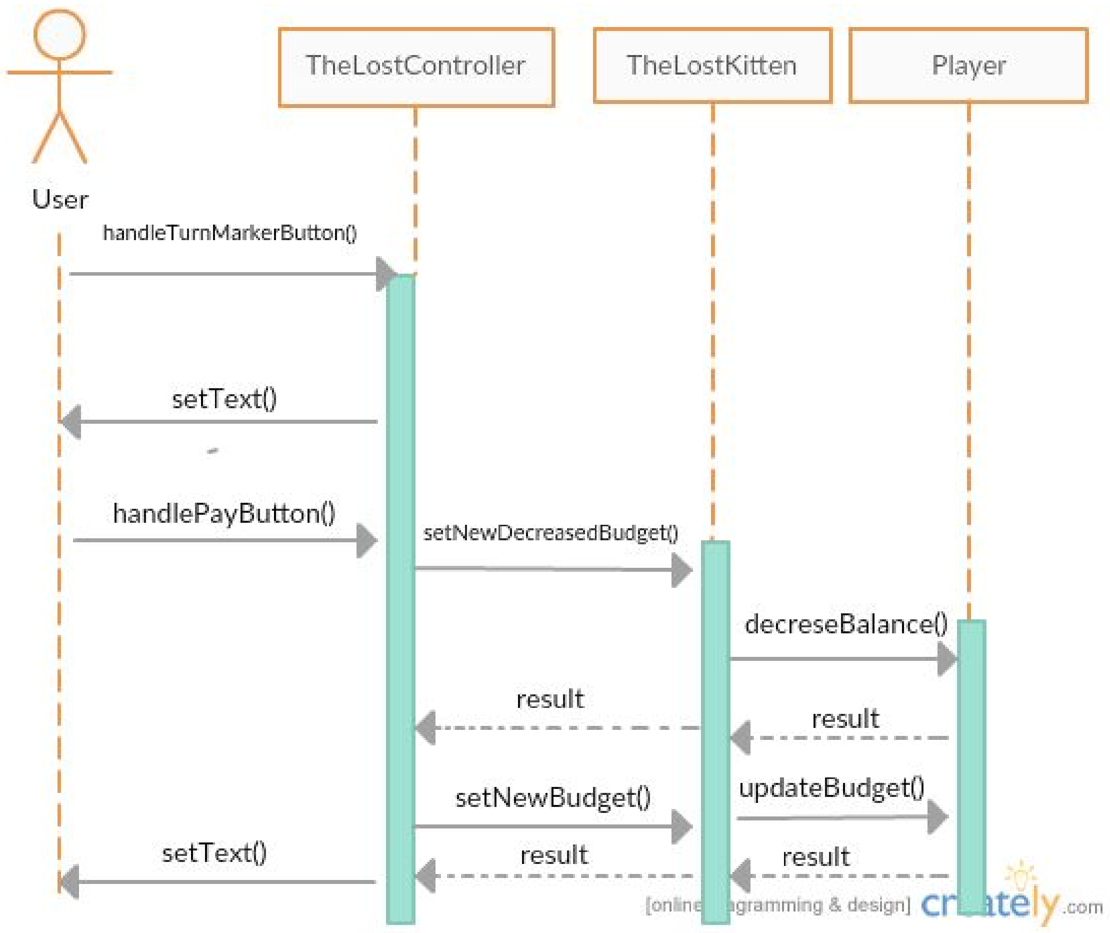
```java
public List<ISpace> findPotentialSpaces(int numberOnDice, ISpace positionOfPlayer) {

    List<ISpace> tempList = new ArrayList<~>();
    potentialSpaces.add(positionOfPlayer);
    for (int i = 0; i < numberOnDice; i++) {
        for (int j = 0; j < potentialSpaces.size(); j++) {
            for (int k = 0; k < potentialSpaces.get(j).getAdjacentSpaces().size(); k++) {
                if(potentialSpaces.get(j) instanceof Station){
                    if(!(potentialSpaces.get(j).getAdjacentSpaces().get(k) instanceof Station)){
                        tempList.add(potentialSpaces.get(j).getAdjacentSpaces().get(k));
                        tempList = mergeLists(potentialSpaces, tempList);
                    }
                }
                else {
                    tempList.add(potentialSpaces.get(j).getAdjacentSpaces().get(k));
                    tempList = mergeLists(potentialSpaces, tempList);
                }

            }
            visitedSpaces.add(potentialSpaces.get(j));
        }
        potentialSpaces = checkForVisitedSpaces(tempList);
    }
    for (int l = 0; l < visitedSpaces.size(); l++) {
        if (visitedSpaces.get(l) instanceof Station){
            if (!visitedSpaces.get(l).compareSpaces(positionOfPlayer))
                potentialSpaces.add(visitedSpaces.get(l));
        }
    }
    EventBus.BUS.publish(new Event(Event.Tag.FIND_PATH, this));
    return potentialSpaces;
}
```

3.6 Create Path

An other rather complex algorithm is the createPath method inside of the Map class. This method uses recursion to determine how the path between two stations should be drawn. There are actually two very similar instances of this method. The first one, in the Map, are used to create adjacent spaces between two stations. The second very similar method is used in the MapView where the graphic paths are being drawn. The reason why we did two very similar algorithms is that we wanted to make our application follow the MVC pattern, this means that we could not draw the visual path in the Map class and creating the adjacent spaces along the path shouldn't be the view's job.

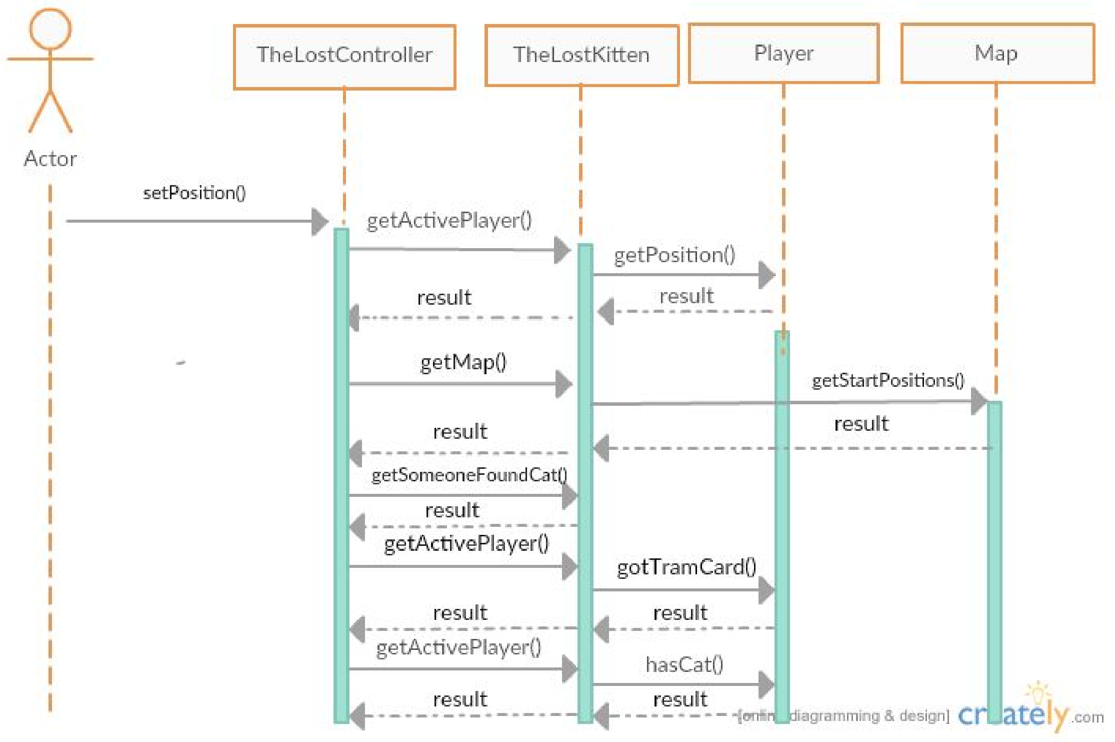
4. Sequence diagram

4.1 Use case 5: Flip Marker



setNewDecreasedBudget() - Updates the players budget by decreasing it with the cost of flipping a marker.
setNewBudget() - Updates the players budget with the amount specified by the marker. May be increased, decreased or stay the same, depending on what marker is turned.

4.2 Use case 9: Win game



5. Persistent data management

The resources i.e. images are stored in the folder *Resources* in the *Src* folder for the game.

6. Access control and security

NA. This application is handled in the same way if you are a user as if you are the developer. It is launched by running the application and exited by either clicking on the 'Close Game'- button or clicking on the red cross on the window.

7. References

Design Patterns-MVC,
https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm, 2017-05-16

Java SE Application Design With MVC,
http://www.oracle.com/technetwork/articles/javase/index-142890.html,
2017-05-16

JavaFX, https://en.wikipedia.org/wiki/JavaFX, 2017-05-15

What is JavaFX?, http://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm,
2017-05-16