TIPLPA

Robot Floor Control

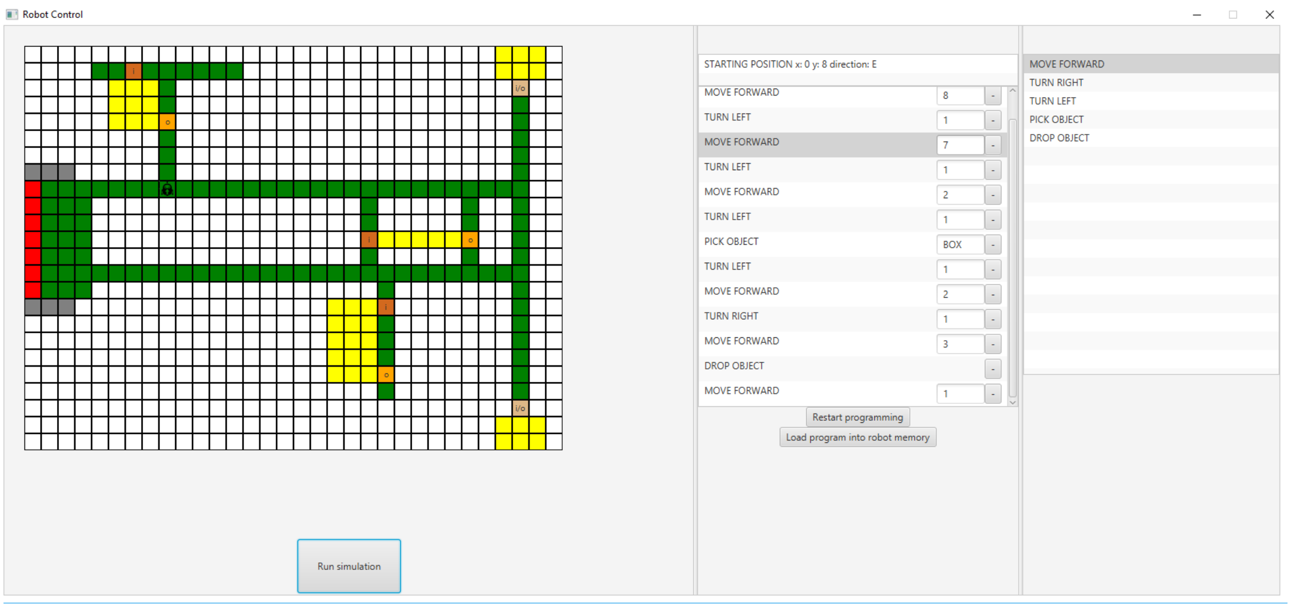
May 24th 2017

Group 20

Michał Kapiczyński - 201700096

Benjamin Schlie - 201370902

Lasse Have Nielsen - 201370981



```
STARTING POSITION x: 0 y: 8 direction: E
MOVE FORWARD    8
TURN LEFT       1
MOVE FORWARD    7
TURN LEFT       1
MOVE FORWARD    2
TURN LEFT       1
PICK OBJECT     BOX
TURN LEFT       1
MOVE FORWARD    2
TURN RIGHT      1
MOVE FORWARD    3
DROP OBJECT
MOVE FORWARD    1
```

TIPLPA, Q4 2017

Department of Engineering, Aarhus University

Version 1

Contents

1 Introduction

This paper describes a robot control system made for a factory application. Its main purpose is to provide the interface for programming robot behaviour. The initial description of the assignment stated needs and requirements. From that the requirements for this project of scope were created.

2 Requirements

The first thing done was identifying the requirements for the project, so that all group members were aware of the desired project result.

2.1 Functional Requirements

FR-1 The factory floor shall be organised in tiles.
FR-2 The workstations can be placed on the tiles.
FR-3 Each workstation shall be able to have a drop-off or a pick-up point or both of them.
FR-4 The labels of the tiles for the Scheme part shall follow table 1 on the following page.
FR-5 A robot shall be able to carry at most one work piece at a time.
FR-6 An error should be displayed if the robot leaves the area it is allowed to operate on.
FR-7 A user shall be able to define the order in which workstations are operated.
FR-8 Next to the parking strip security barriers shall be placed so that robots can only enter or leave from the same direction.
FR-9 A robot attempting to leave the area of movement-allowed tiles must be immediately switched off and its alarm shall be activated.

2.2 Non-Functional Requirements

NFR-1 All functionality of the robot shall be written in Scheme.
NFR-2 The IDE shall be implemented in Java.
NFR-3 The Java part shall not contain state of:

- **NFR-3.1** Work stations
- **NFR-3.2** Position in the space
- **NFR-3.3** Tile representation meaning

NFR-4 The user should be allowed to choose the starting point and initial direction of the robot.

3 Design

The requirements has been turned into a design of the system. All the examples from now on will be presented with the example Floor Plan showed at fig. 1 on the next page, which can be changed by the user at any point with no effect on actual system implementation.

3.1 Floor Plan

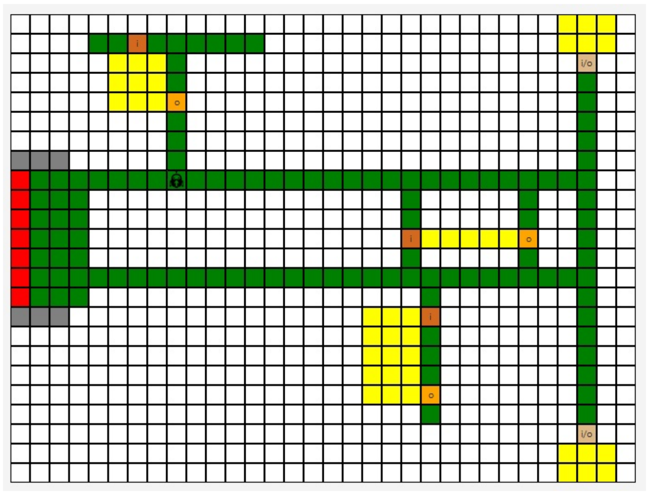


Figure 1: Example of the Floor Plan

The Floor Plan was designed to be represented by two dimensional array of characters. This approach was adopted due to the fact that character representation of tiles is easy in future implementation and independent of the actual technology using the Floor Plan later on. The overview of colours, and their equivalent Scheme character representation is shown in table 1.

Tile Color	Description	Scheme Character
'Grey'	The robot is NOT allowed to access the tile	'-'
'Green'	The robot is allowed to access the tile	'A'
'Red'	Tile for parking the robot	'P'
'Dark Grey'	Safety barrier	'S'
'Brown' (Marked with 'i')	Tile stating that the robot can deliver an object to the workstation	'i'
'Orange' (Marked with 'o')	Tile stating that the robot can retrieve an object from the workstation	'o'
'Light brown' (Marked with 'i/o')	Tile stating that the robot can both retrieve and deliver an object.	'*'
'Yellow'	Tile stating a workstation represented by a number: 1, 2, 3...	'1, 2, 3...'

Table 1
Characters for designing the Scheme floor

3.2 Robot state management

From the requirements, it was already concluded that all logic of the robot, should be written in Scheme. Initially a sequence diagram was made to make a clear separation of what logic the Scheme part should contain, and what logic the Java part should contain. An example of this is shown in fig. 2 on the next page.

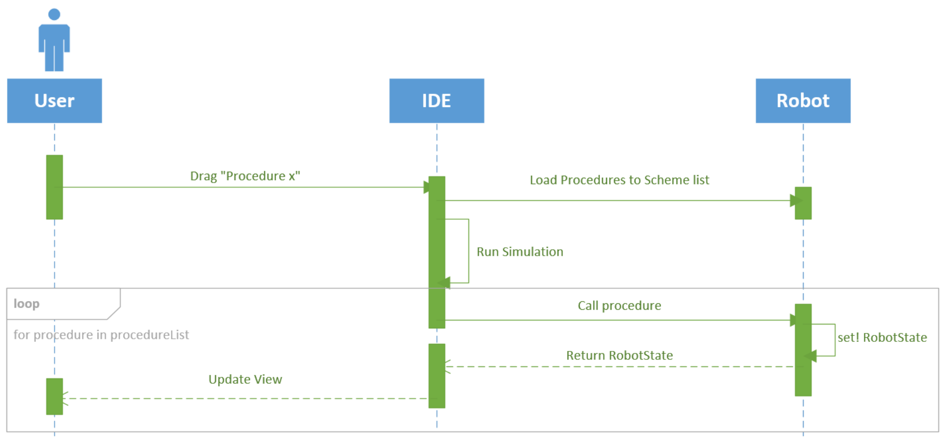


Figure 2: Example of sequence diagram: Move Robot

4 Implementation

This chapter describes the implementation of the Factory Floor Robot Control. The chapter consist of two sections describing the different programming paradigms used for implementation: the imperative part (Java) and the functional part (Scheme).

4.1 Java

The imperative part is written in Java, and is built as a JavaFX application [4]. This part contains all the GUI logic and uses a Kawa [2] framework for communicating with the functional part. The application is split into several packages in order to separate logic: **GUI** - containing all user interface components and functionality such as animations. **Robot** - holding a model of the robot state which is mapped when communicating with Scheme. **Utils** - different utilities such as file reader for loading scheme files and mapping used when displaying the floor. **Scheme** - containing Scheme communication logic.

To communicate with Scheme the Kawa framework is used. This framework allows Scheme to run in a Java environment.

The code snippet in table 2 from SchemeConfigurer class shows the initial configuration of the Scheme environment from java. Line 2 initialises the Scheme environment and line 3 sets a global scheme environment meaning that creating an environment for each Scheme interaction is not needed. Line 4 is a method call which loads all Scheme files into the scheme environment.

```
1     public void configureSchemeEnvironment(){
2          registerEnvironment();
3          Environment.setCurrent(new Scheme().getEnvironment());
4          configureSchemeFiles();
5      }
```

Table 2
Configuration of scheme environment from Java

Code snippet in table 3 is from SchemeProcedure class which is an extension of Procedure1 class from Kawa and wraps a Scheme call into an object. The SchemeProcedure class takes the Scheme procedure name as an argument in the constructor (this.name), and the function apply1 takes one argument. From this the evaluation string is created, and evaluated in the Scheme environment created in table 2 on page 4, and the result is returned which can then be used from Java, e.g. the current robot state.

```java
    public Object apply1(Object arg1) {
         Object result = Scheme.eval("(" + this.name + " " + arg1 + ")",
                 Environment.getCurrent());
         return result;
     }
```

Table 3
Evaluation of Scheme procedures from Java

4.2 Scheme

The Scheme code contains all the functionality of the program regarding robot behaviour, and is based on the functional programming paradigm. The implementation was split into separate files. The **FloorPlan** - which holds the state of the floor. **FloorUtil** - which is used as an accessor to the floor and tiles. **Robot** - holds information about the robot "object", and defines the rules imposed on its behaviour. **Simulation** - which runs the simulation and calls procedures of the robot.

As mentioned, Scheme is a functional programming language, and is very good at handling computations. However, the system requirements stated that Scheme should contain logic and state of the robot. Thereby, it was needed to use the Scheme function "set!", which sets a variable in the global state, and as a result contradicts with the understanding of the environment model [3]. Other than that, the loaded commands and command pointer were also saved with a "set!", so it was known what was the last procedure which has been called.

It gave a good separation of concerns, as all logic of the robot is contained in Scheme, and view logic in Java. The robot behaviour has been encapsulated in the class representation being a function containing other functions. The robot class function is immutable meaning that every call changing the robot state resulted in creation of a new robot instance. However, not all of the inside functions are pure functions because some of them depend on the Floor Plan representation, which is in contradiction to functional programming paradigm.

In table 4, a snippet of the factory floor is shown. It is a vector of vectors, which defines all the tiles.

```scheme
(define factoryFloor (vector
    (vector '- '- '- '- '- '- '- '- '- '- '- '- '- '- '- '- '- '- '- '-
        '- '- '- '- '- '- '- '- '0 '0 '0 '-)
    ...
```

Table 4
Scheme: Factory Floor

The implementation imposing the dependency between the robot and floor state complicates the design and makes it especially difficult to test the Scheme component, which is described in chapter 5 on the following page.

There is (moveRobot) function provided which is supposed to be called from Java and call the generic function "getNextRobotState" procedure, which does a "set!" on the robot state, with a function to be called as a parameter, so that no matter what function was passed as a parameter (moveForward, pickObject etc.) it would always update the robot in its global state. By doing it in a single generic function, it was ensured that set! on robot state was called only in this function, not all over the code, and that it was the same variable that was saved every time.

5 Tests

The system consists of two independent components: Java and Scheme, communicating with each other. As a result, it is prominent to guarantee the correctness of both components, as well as the system as a whole.

5.1 Java test

Java application is a graphical user interface application and it only contains logic regarding the user interface and communication with the Scheme application. As a result tests has been divided into two groups.

5.1.1 GUI tests

Graphical User Interface application requires proper functional testing approach due to the fact that the application has been implemented with use of the Java UI library JavaFX. JavaFX requires special actions in testing scenarios such as creating a JavaFX environment and UI mock components. Therefore it was decided to use TestFX framework. User interface tests if the interface components such as buttons or drag and drop elements work as expected, with no focus on robot behaviour.

5.1.2 Scheme communication

The Java component acting as a user interface for the Scheme application is highly dependent on the Scheme implementation. As a result unit tests validating the correct cooperation of Java - Scheme components have been implemented. The unit tests do not validate the scheme return value but only the operation of the communication between the components.

Covered test scenarios:

1. Test Scheme Environment configuration,
2. Test calling different scheme procedures from Java,
3. Test mapping of Java objects into Scheme procedure arguments,

5.2 Scheme tests

For testing of the Scheme component the Scheme unit testing framework named **rough-draft** [1] has been used. The framework allows to write test suites consisting of multiple test cases and use diverse assertions with detailed test result description.

All the implemented tests follow the ”arrange-act-assert” model. The example test implementation is shown in table 5 on the next page.

```
(import
 (rough-draft unit-test)
 (rough-draft console-test-runner))

(include "../robot.scm")

(define-test-suite robotTestSuite

  (define-test robotMoveForward-test
    (let ((robot (robot 0 8 "E" 0 '())))
      (let ((newRobot (send 'moveForward robot 8)))
        (assert-eqv? (send 'getX newRobot) 8)
        (assert-eqv? (send 'getY newRobot) 8)
        (assert-eqv? (send 'getErrorCode newRobot) 0)
        )
      ))
  )

(run-test-suite robotTestSuite)
```

Table 5
Unit testing in Scheme

There have been implemented three test-suites: ”FloorUtilTest”, ”robotTest”, ”simulationTest” with multiple test cases covering together all the ”public” functionality of the Scheme component. Tests use the test mock Floor Plans independent of an actual implementation of the application Floor Plan.

5.3 Integration tests

Testing system consisting of multiple components is always a challenge. It is crucial to verify that the state shared by this components is consistent for a whole period of system operation.

Taking into consideration that Java application provides the input data for the whole application (user decisions regarding the robot behaviour) it was decided that the test of the whole system would be implemented on Java side with use of functional UI testing, and use of Mock Scheme Environment.

Mock Scheme Environment would call the same methods as the real application but would use the mocked FloorPlan and state specific objects designed specifically for test scenarios. As a result, knowing the mock state on java side it would be possible to predict the desired results of scheme procedures and detect incorrect application behaviour such as robot moving wrong direction on the User Interface.

6 Discussion

Throughout this project two programming paradigms have been used. This approach has its advantages as well as some drawbacks which will be discussed in the following.

First of all, dividing the whole system into two separate components communicating with one another provided very clear separation of concerns between the robot behaviour, and the IDE GUI implementation. What is more, the functional paradigm was found to be very simple, and intuitive for operations modifying robot state such as moving the robot, turning it or picking, dropping an object. One of the biggest challenges was to try to keep to pure functional principles, and store the robot state on the Scheme part at the same time. It required violating some functional programming rules by using the "set!" procedure, but it was done with full conscience and was tried to be done only where necessary.

Even though, the scheme part is written with regard to functional paradigm an object oriented way of thinking encouraged the encapsulation of the robot behaviour in the functional class representation. Due to the dependency between the robot and floor state some of the robot state management functions are not pure functions, which probably could be avoided by providing the floor state as a function parameter. However, it was considered to be very unclear and troublesome, and it was decided to include the floor as a dependency for the robot. Moreover, the higher order functions were found useful for passing procedures as a parameter and executing it from another function.

The Java part has been implemented with use of the JavaFX GUI library, and it follows the imperative and object oriented way of thinking. The Java - Scheme communication implemented with use of Kawa framework has been encapsulated in separate package, and Java classes providing easy, and scheme-independent use from other Java components.

The system testing has been also implemented for both Java and Scheme parts. It was found difficult to test the java implementation due to the use of JavaFX GUI library which requires not only unit testing, but functional testing as well. The functional testing was not implemented, but the strategy was defined in the chapter 5 on page 6.

Scheme testing was found confusing, especially at the beginning as the Scheme language has been learned during implementation of the solution. At the end with use of the unit testing framework, and due to the fact that most of the functions were pure functions the tests implementation was not very troublesome. The only challenge was the management of the dependency between the robot and the floor for testing scenarios.

7 Conclusion

It was a challenge as well as a very interesting topic to use two different programming paradigms in the same project. From the already acquired knowledge of imperative programming, the functional paradigm was found especially challenging. Throughout the project is was possible to notice strengths: such as easy implementation of mathematical computations and the higher order functions allowing passing of functions as a parameters through the program flow. Furthermore it gave the ability to implement pure functions which were found very easy to test. On the other hand, it was found burdensome to keep to pure functional programming principles due to the need of storing persistent state at some point. It was a challenge to design the system so it was easy to test, when there was limited knowledge to the language and paradigm in the beginning. All in all, functional programming is considered to be very useful in specific applications, and is found to be a good extension to the standard imperative and OOP approaches.

Bibliography

[1] akeep. rough-draft. `https://github.com/akeep/rough-draft`, 2017. [Online; accessed 24th of May 2017].

[2] Kawa. Kawa. `https://www.gnu.org/software/kawa/`, 2017. [Online; accessed 23-May-2017].

[3] Klaus Kristensen and Stefan Hallerstede. Higher Order Programming. Week2-HigherOrderProrgamming.pdf, 2017. [Slides from Blackboard].

[4] Oracle. JavaFX. `http://www.oracle.com/technetwork/java/javafx/overview/index.html`, 2017. [Online; accessed 23-May-2017].