

TIPLPA

ROBOT FLOOR CONTROL

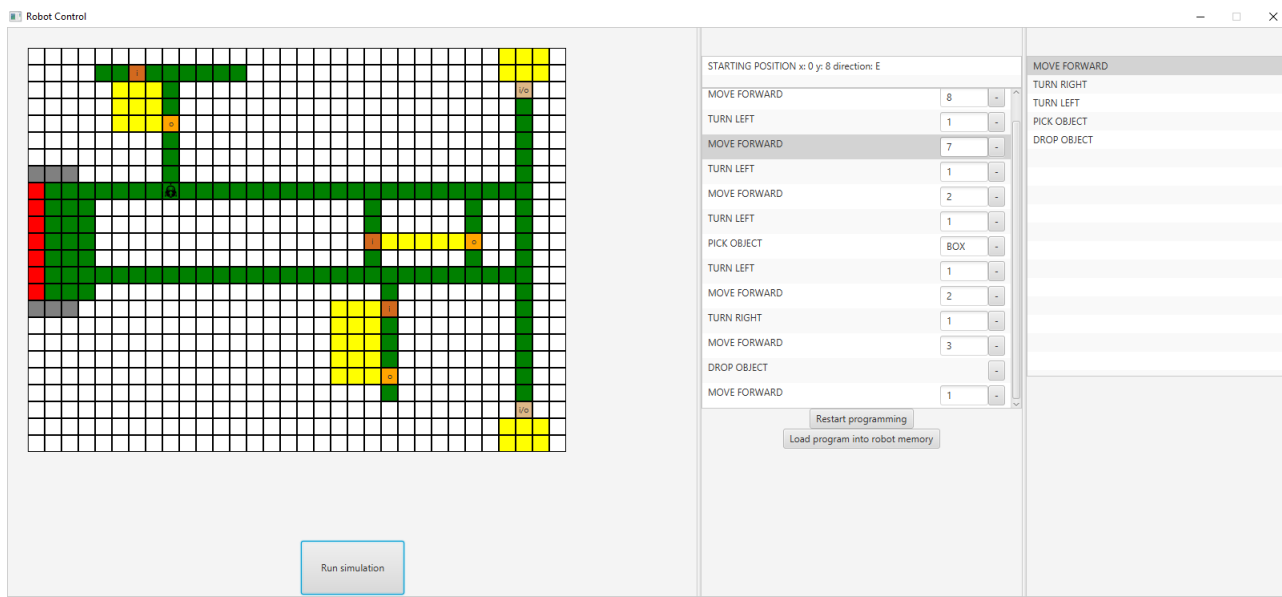
MAY 24TH 2017

GROUP 20

MICHAŁ KAPICZYŃSKI - 201700096

BENJAMIN SCHLIE - 201370902

LASSE HAVE NIELSEN - 201370981



TIPLPA, Q4 2017

DEPARTMENT OF ENGINEERING, AARHUS UNIVERSITY

VERSION 1

Contents

Contents	1
1 Introduction	2
2 Requirements	2
2.1 Functional Requirements	2
2.2 Non-Functional Requirements	2
3 Design	2
3.1 Floor Plan	3
3.2 Robot state management	3
4 Implementation	4
4.1 Java	4
4.2 Scheme	5
5 Tests	6
5.1 Java test	6
5.2 Scheme tests	6
5.3 Integration tests	7
6 Discussion	7
7 Conclusion	8
Bibliography	9

1 Introduction

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

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.

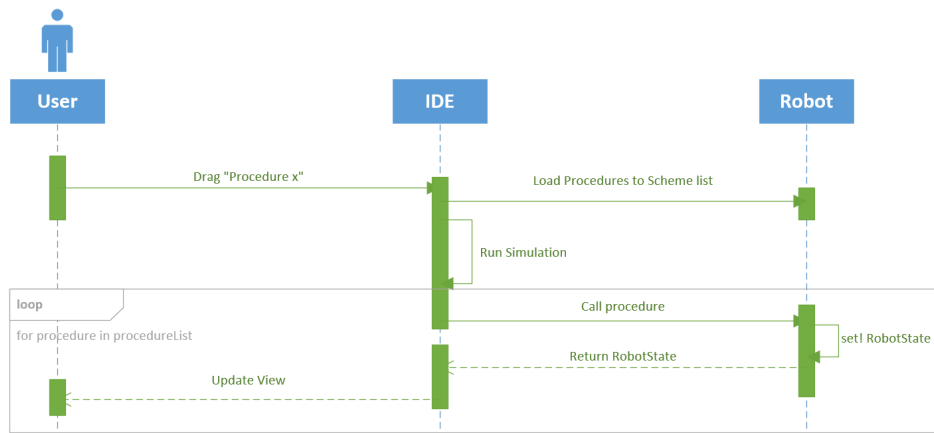


Figure 2: Example of sequence diagram: Move Robot

4 Implementation

This chapter describes the implementation of the Factory Floor Robot Control. The chapter consists 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

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 has 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.

```

1 (import
2   (rough-draft unit-test)
3   (rough-draft console-test-runner))
4
5 (include "../robot.scm")
6
7 (define-test-suite robotTestSuite
8
9   (define-test robotMoveForward-test
10    (let ((robot (robot 0 8 "E" 0 '())))
11      (let ((newRobot (send 'moveForward robot 8)))
12        (assert-eqv? (send 'getX newRobot) 8)
13        (assert-eqv? (send 'getY newRobot) 8)
14        (assert-eqv? (send 'getErrorCode newRobot) 0)
15      )
16    ))
17 )
18
19 (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 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 work to encapsulate 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 provided 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 to imperative programming, the functional paradigm was found especially challenging. Throughout the project it 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 Hallersted. 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].