MSO Lab Assignment 3:
Programming Learning App – Iteration 2

Version 1-2024

Introduction & description

In the previous assignment you created a design and basic implementation for an educational application that introduces people to the basics of programming.

In this assignment you will expand the application with new functionality. Adjust your design where necessary. You will also test the (new) functions and demonstrate that you can write code of good quality.

New features

UI for building programs

The first step is to build the user interface (UI) of the app, in which users should be able to load, build, and run programs. It does not have to look fancy, but of course you can make it as attractive as you want. You can make a simple Forms-application, but you can also choose a modern UI framework that doesn't require complex installs.

For the character, you can choose anything you like, such as a cat, a car, or a fantasy figure. You are also free to choose colors, and any other layout aspects.

The UI should support the following functions:

Loading programs

Users should be able to select a program file from a file chooser, or load a basic, advanced, or expert program, which randomly gives one of the hard-coded programs.

Editing programs

There are several ways to build and edit programs. You can choose a simpler, text-based solution ('satisfactory' score), or a more advanced block-based interface (for a 'good' – 'exceptional' score).

Text-based. The program is loaded as text in a text field, which can be easily edited by the user.

Block-based. The program is represented as a list of blocks, with indented blocks for the repeat command. There should be buttons to add a command, and you should provide input fields (possibly in a popup-window) to let the user enter parameters. You can choose how to place a command into the program:

- By clicking on a position, the command will be inserted.
- The command can be dragged and dropped into the desired location.

Think of a smart way to insert and show the indented blocks. You can also choose if you want to draw the program on a canvas or use UI elements.

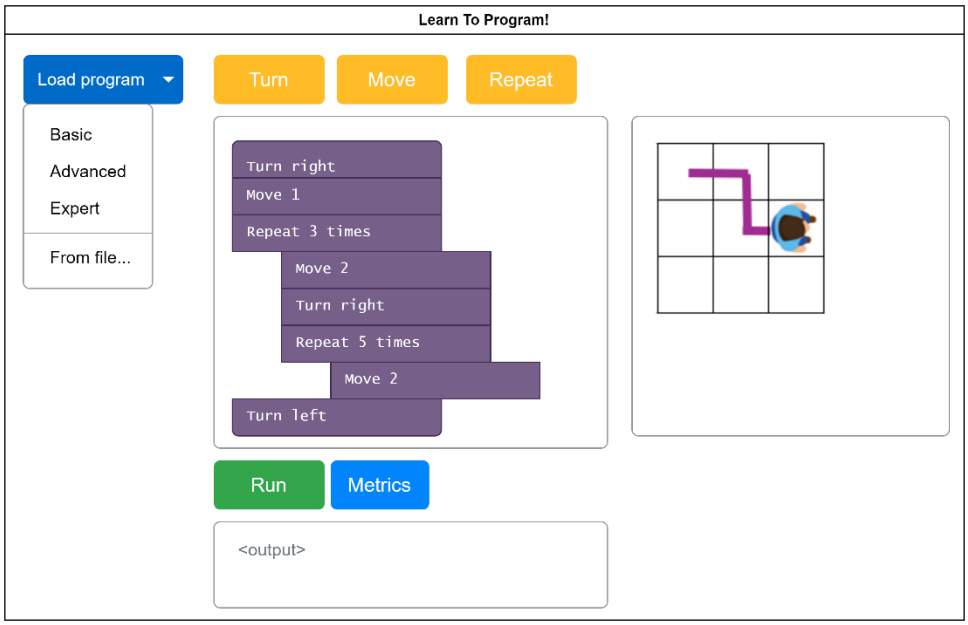
If a block is selected, you should be able to delete or move it.

Running programs and metrics

If the user presses a 'run' button, the program should be executed. The output trace is shown in a text field. If the user presses the 'metrics' button, the results are also displayed in a text field.

In addition, the path of the character is drawn in a separate area.

Below is a sketch of what the UI could look like, but you are free to make other choices. Please note that program and output do not match in the example. It does not matter (yet) that it's possible for a character to move to negative coordinates.

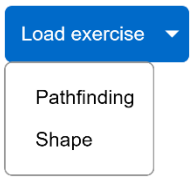


Make sure the UI is as separate as possible from the domain logic of the application, so it would not be too much trouble if you would replace the UI by, for example, a web interface.

The next features all require updates to the UI.

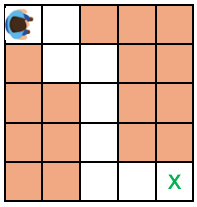
Exercises

We are going to offer exercises in our app, to really challenge our users to practice programming. There are two types of exercises: pathfinding and shape exercises. These exercises are loaded from files after clicking on a new menu or button.



Pathfinding exercises

In practical assignment 2, your grid could in theory be infinite. In pathfinding exercises, a grid is given with fixed dimensions, and the cells of that grid can be blocked. The character starts at a certain position (cell), and there is also an end position indicated. The goal of the exercise is to write a program that transports the character to the end position.



Grids should be loaded from text files, and may look like this (corresponding to the example above):

```
oo+++
+oo++
++o++
++o++
++oox
```

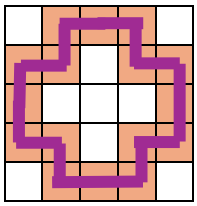
Note that a blocked cell is represented with a '+' sign, an open cell with an 'o' sign, and the end state with an 'x'. You may assume the character starts at the top-left position. You do not have to check for valid grids; you may assume there is always a valid path to the end state.

When loading an exercise, the grid should be shown on the screen, so the user can start writing a program that 'solves' it, by leading the character to the end state.

When **running** a program, the resulting path should be drawn on the grid. When a user wants to move to a blocked cell, or a cell outside the grid, a runtime exception should occur. Create custom-made exceptions for these different types of runtime errors. After running the program, there should be a clear sign of either success, failure (e.g. the character is in the wrong position), or the occurrence of a runtime error.

Shape exercises

In this type of exercise, the input grid has some kind of shape that the user should 'draw'. You do not have to check for shapes that are impossible to draw with the current commands (e.g. with two separate components).



The files for these exercises look similar, but the 's' shows the starting position for the shape:

```
+soo+
oo+oo
o+++o
oo+oo
+ooo+
```

When loading a shape exercise, the grid (image to draw) should also be shown on the screen. The user should write a program that draws this exact shape. The character should not be visible.

When **running** a program for this exercise, the drawing will start from the indicated position. The resulting path should be shown on the grid. For these exercises, it does not matter if the path is drawn outside the shape or grid. After running the program, there should be a clear sign of either success, or failure.

For each of the exercises, make sure to create several program files for testing.

Exporting programs

Programs should be **exported** to several formats, which the user can choose after pressing an 'export' button or menu-item. Implement the text format, and chose one other format. However, take into account that it should be easy to add another export format.

- (required) Text-format. This is the format also used for reading programs from a file.
- (option) HTML. This is an export in HTML format, with a simple header, the commands in bold, and a list of statements as an item list.
- (option) Export to a simple JSON format that you may design yourself.

New commands

A new command, the conditional loop, is introduced.

`RepeatUntil [cond] [command list]`	The command list is executed as long as the condition is not true anymore. There are **two conditions** that can be checked: if there's a wall in the next cell, and if we're on the edge of the grid. In case no grid is loaded, this would always return false.

Below are some new example programs.

Example 1 "FindExit1"	Example 2 "FindExit2"

`Repeat 3`
`  RepeatUntil WallAhead`
`  Move 1`
`  Turn right`
`Repeat 2`
`  RepeatUntil GridEdge`
`  Move 1`
`  Turn right`
	`RepeatUntil WallAhead`
	`  Move 10`
	`  Turn right`

Expanding the language has an effect on other functions, make sure to update those:

- We should be able to **import/export** with the new commands.
- **Execution**. In the textual trace, each executed command is shown.
- We want to add some new commands to the **hard-coded examples**, for the advanced (using a repeat statement), and expert programs (with nested repeats).
- The number of repeat-commands **metric** should also count the new commands.

Tasks

Part 0 Personalization of your application

Invent or generate a catchy name, main character, and (optional) a logo/layout.

Part 1 Extending and adapting the software design

Expand your design with the above features. Adjust your design if necessary, so that the new features can be added easily, and incorporate any feedback from the TAs.

Describe the following parts in a separate PDF file:

- Updated design. Update your UML class diagram. If it gets very complex, split it into multiple diagrams (possibly accompanied by a package diagram). Make sure you show clearly how they are related. For classes that need an additional explanation, provide a description in a separate text section with the specifics.
- Indicate which design patterns you have used, and indicate for each pattern which problem it solves, and how it is implemented (by describing how the pattern elements map to your design).
- Deviations from the original design. Please explain where you deviated from the original design from the previous assignment, and how you processed the feedback.

If you have made any assumptions about how something should work, give these in a list. If you asked a TA to clarify something that is not in the description, add that to the list as well (mentioning that you got the answer from a TA).

Part 2 Implementation and code quality

Write C# code for the new features. Make sure your code is of good quality by, among other things:

- Adhere to a code standard.
- Use a tool that provides feedback on your code quality, such as SonarQube or Resharper.
- Calculate various metrics and inspect their results (see lecture 7b).
- Provide comments where necessary, and clear naming.
- Conduct code reviews of each other's code in which you check the above items.

Describe in a reflection (about 1 page) what measures you have taken regarding code quality. Provide several concrete examples of refactorings[1] that you have implemented. Also show how the results of applied metrics led to changes in the code.

Part 3 Evaluation

Evaluate your final design. Explain the different kinds of variation that you have encountered in this domain, and how your design handles this variation. Explain why you chose to use design patterns for certain problems, compare with alternatives and explain why you did not choose those alternatives.

Also explain how it is capable of handling changing requirements. Finally, give an analysis of your design regarding cohesion and coupling. How does your design exhibit high cohesion and low coupling?

Part 4 Testing

Test your application extensively with the following elements:

- Unit tests. Where necessary, create fake objects (mocks, stubs) to test code with dependencies in isolation.

[1] https://refactoring.com/catalog/

- System tests. Use user stories and scenarios to test various functions within the UI.

Perform these tests manually on a regular basis, and keep track of the results (pass/fail + reason) in a document (e.g. an Excel sheet).

To score 'exceptional' for this part, you have performed the above basic very well and supplemented it with something extra: property-based testing, a mocking framework, testing and evaluation with real users (think of family, friends), or another relevant test technique.

Ensure good quality of your automated tests[2], and attach a screenshot of the final test run. Also submit the results of the manual tests. Provide a brief reflection on the approach, test quality, and results (about half a page).

Part 5 Work distribution & retrospective

Add your grader (TA) from P2 to the repository, so they can inspect your contributions.

Give a detailed overview of the task distribution: who has worked on which part?

Conduct a *retrospective* in which you reflect on your collaboration. What went well, and what didn't? What have you learned looking back at all lab assignments?

Tip

If you have not done so yet, set up a Kanban-board for you and your partner with lanes such as 'back-log', 'implementing, 'testing', and 'done', and add cards for user stories and tasks, to organize and keep track of our progress. You can use software such as Trello or Microsoft Planner for this.

Make sure to prioritize; start with the essential elements first (knockouts), and only later with more advanced features and bonus points.

Bonus points

You can earn at most 1 bonus point, so you can choose a maximum of two of the options below. Make sure the features are included in the design, descriptions, and tests.

Logging. (0.5 bonus point) Implement using events[3] or the Observer pattern[4] functionality for a log file, so you can retrace the actions of a user trying to solve an exercise.

Add text to the logfile when:

- An exercise is loaded.
- A command is added
- The program is run + the output.

Code.org import (0.5 bonus point) While we can already import programs in textual format, we would also like to import programs that have been created in the popular code.org environment. In appendix A you will find some example programs in their JSON format. These programs are much more advanced, so you probably won't be able to import all of them, but simple programs can be converted to our format.

New command (0.5 bonus point) Add a new type of command to the language. It should not be a simple one such as 'Move2Steps', but something like a conditional, variables, etc.

New feature with pattern (0.5 bonus point). Implement an original new feature for which you use a design pattern that you haven't used anywhere else in the application. Be aware that this pattern should

[2] https://testsmells.org
[3] https://docs.microsoft.com/en-us/dotnet/standard/events/
[4] https://learn.microsoft.com/en-us/dotnet/api/system.iobservable-1

really make sense for the feature. Explain your choices in the design, and clearly indicate that this is your bonus feature.

General considerations

- *Language*. Both English and Dutch are allowed. The writing and layout of the document is a knockout criteria for a passing grade!
- *Task distribution*. The work should be distributed equally among the two team members, and *both team members contribute to all components*: design, programming, testing, and reflection. If this is clearly not the case, this will be reflected in the grading.
- *Grading*: check the attached Excel-sheet to see how you will be graded (note this sheet is subject to minor changes).
- *Authenticity:* The assignments should be you and your teammate's own work; you may not use work from other students/websites. The design of the classes, attributes, methods, tests, and patterns must be your own work. If in doubt, ask first if you can use something!
- *Using existing code:* If you use code that is not your own, you must cite your source in your report and in the code. This is only allowed for tools such as libraries to read files, UI elements, etc. You may use AI chat tools such as ChatGPT only for finding bugs in your code and generating code *within 1 method*. You should switch off Copilot and other AI-autocompletion for this project. If you generated some code, you must include the full chat transcript and indicate in the code which pieces were generated.

How to submit

Create a zip or rar file of the complete project, including files to build the project (i.e. the project and solution files if you use Visual Studio). Clean the project before you compress it, so that the zip file does not become unnecessarily large due to all kinds of intermediate files and executables. This means removing the .git folder and bin folders, as well as IDE specific files and folders, such as .vs, .vscode, etc.

If necessary, create a readme file with instructions on how to run the application. The TA's should be able to easily run your application!

Attach a PDF file with the textual parts.

Appendix A code.org programs

Program 1

```
{
    "id": "7",
    "children": [{
        "id": "6",
        "children": [
            {
                "id": "0",
                "type": "maze_moveForward"
            },
            {
                "id": "2",
                "children": [{
                    "id": "1",
                    "type": "turnRight"
                }],
                "type": "maze_turn"
            },
            {
                "id": "4",
                "children": [{
                    "id": "3",
                    "type": "turnRight"
                }],
                "type": "maze_turn"
            },
            {
                "id": "5",
                "type": "maze_moveForward"
            }
        ],
        "type": "statementList"
    }],
    "type": "program"
}
```

Program 2

```
{
    "id": "9",
    "children": [{
        "id": "8",
        "children": [
            {
                "id": "0",
                "type": "maze_moveForward"
            },
            {
                "id": "2",
                "children": [{
                    "id": "1",
                    "type": "turnLeft"
                }],
                "type": "maze_turn"
            },
            {
                "id": "4",
                "children": [{
                    "id": "3",
                    "type": "turnRight"
                }],
                "type": "maze_turn"
            },
            {
                "id": "5",
                "type": "maze_moveForward"
            },
            {
                "id": "7",
                "children": [{
```

```
                    "id": "6",
                    "type": "turnLeft"
                }],
                "type": "maze_turn"
            }
        ],
        "type": "statementList"
    }],
    "type": "program"
}
```


Program 3


```
{
    "id": "13",
    "children": [{
        "id": "12",
        "children": [{
            "id": "11",
            "children": [{
                "id": "10",
                "children": [
                    {
                        "id": "0",
                        "type": "maze_moveForward"
                    },
                    {
                        "id": "8",
                        "children": [
                            {
                                "id": "1",
                                "type": "isPathForward"
                            },
                            {
                                "id": "4",
                                "children": [{
                                    "id": "3",
                                    "children": [{
                                        "id": "2",
                                        "type": "turnLeft"
                                    }],
                                    "type": "maze_turn"
                                }],
                                "type": "DO"
                            },
                            {
                                "id": "7",
                                "children": [{
                                    "id": "6",
                                    "children": [{
                                        "id": "5",
                                        "type": "turnLeft"
                                    }],
                                    "type": "maze_turn"
                                }],
                                "type": "ELSE"
                            }
                        ],
                        "type": "maze_ifElse"
                    },
                    {
                        "id": "9",
                        "type": "maze_moveForward"
                    }
                ],
                "type": "statementList"
            }],
            "type": "DO"
        }],
        "type": "maze_forever"
    }],
    "type": "program"
}
```