MSO
Loopa

Jan Huls (4699610), Arwin Moormans (4965957)

November 2025

Design

Class Diagram

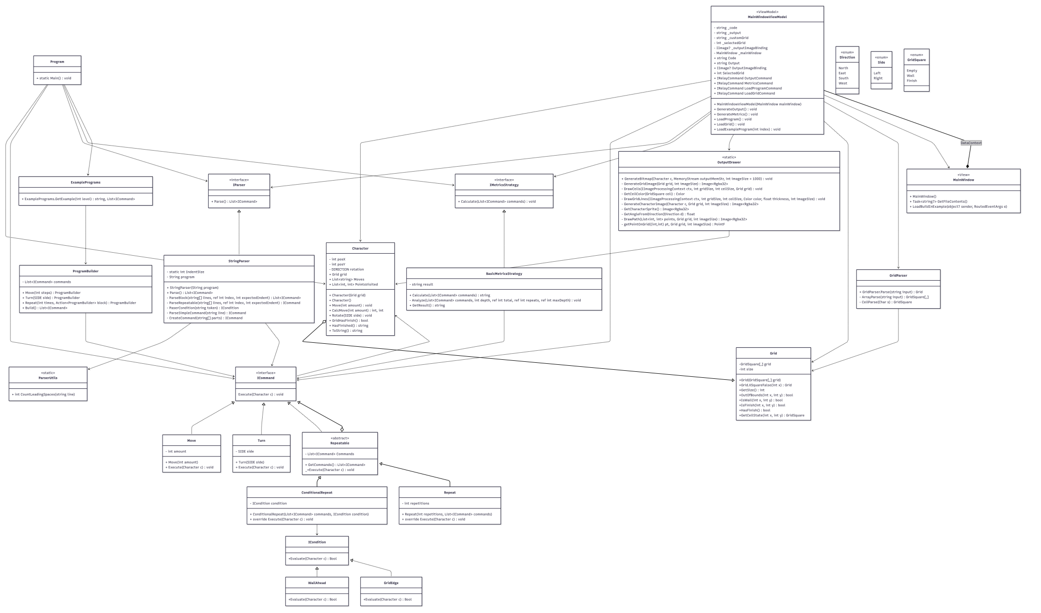


Figure 1: Class diagram of the Programming Learning App

Design Patterns

We mainly used the same design patterns as in the previous practical assignment for the core program. For the UI part, we split the model from the view, but we did not end up using the MVVM pattern as we initially planned, we just bound the model directly to the view. This was because we found it quite hard to implement the pattern with Avalonia, and since our UI is not that complex, we decided to just have the view interact directly with the model. This does however make it harder to switch to a different UI, since there is more coupling between the UI and the model, but with a few small changes to the model, it is still easily done.

Deviations From Previous Design

To implement the `ConditionalRepeat` class, we had to make a few changes, since we encountered problems with our program needing to paste all the moves it has completed. Previously we implemented a `ToString` method for all classes implementing `ICommand`, and while executing the commands, we added them to a list, and after executing we called `ToString` on them. This worked fine for the old commands, but the conditional repeat needs to check a character state to see how many times it should do its code block, which was not possible with our old implementation. To fix this we added a list of executed commands to the character, and each basic command (Turn and Move) adds itself to this list. This way we can easily print the list in the `Character` class. Another change was the addition of the abstract class `Repeatable`, which both `Repeat` and `ConditionalRepeat` inherit from. This was done to reduce code duplication, since both classes had a lot of similar code, such as holding a list of commands and executing them multiple times.

Assumptions

During the development of the project, we made a few assumptions to interpret the assignment.

- The grid is always square: We assumed that the grid is always square, meaning that the width and height are always the same. This assumption was made to simplify the implementation of the grid and character movement.
- The character always starts at position (0, 0) facing east: We assumed that the character always starts at the top-left corner of the grid, facing left. This assumption was made to have a consistent starting point for all scenarios.
- When using the 'sandbox' mode, the grid is finite and the character cannot move out of bounds: We assumed that in sandbox mode, the grid has a fixed size and the character cannot move outside the grid. If the character tries to move out of bounds, an error is thrown.

Code Quality

Resharper

We use the Rider IDE from Jetbrains as our main IDE for coding this project, luckily Rider has a great build-in code Quality tool called Resharper. Resharper provides good feedback on code quality which we have used it intensively throughout the whole project. Resharper gives errors (which are actual c# errors) and warnings which is feedback on our code quality. These warnings can be naming conventions, possible null values, redundant code and much more. This way we always wrote code that complies to the code quality standards

Metrics

Resharper automatically tests the cyclomatic complexity of our code, and if it is more than 10 in a method, it will give a warning. It also tests a number of other metrics, which when above a certain threshold, will give a warning or error.

Reflection & Code reviews

Using Resharper took a bit of time to get used to. Normally we didn't really care about naming conventions or possible null values and only cared about the code working. But now that we were forced to, it actually made the project a lot more well-organized. Throughout the project we frequently took a look at each others code and gave feedback on each other's work. Giving each other ideas of how the code could be improved or refactored, or making sure that we both understood each other's code. By using Resharper and checking on each other, we have written clean code that complies to the code quality standards.

Evaluation

Analysis

We strive, of course, for high cohesion and low coupling to make the design of our code as extensible and understandable as possible. To achieve low coupling we have seperated the project in 3 sub projects.

- `MSOProgramLearningApp`: This project holds the core logic of our application and holds all the data structures such as the character, grid and commands. It also is responsible for parsing strings into commands, moving the character and drawing an output image. This project on its own is just the commandline interface we made at the previous practical assigment.
- `MSOAvaloniaApp`: This projects uses the Avalonia library to create a window for the actual finished application. All the UI related code is in here, and it uses a reference to the `MSOProgramLearningApp` project for all the core logic.
- `MSOTestProject`: This project is used for testing the logic in the `MSOProgramLearningApp` project. It holds unit and system tests that we will discuss later.

This way we also make sure that each project has high cohesion because it only does a specific part of our application. Since the core of our program is the same as the previous practical assignment, we wil not go into detail about the analysis of the `MSOProgramLearningApp`. We do however note that the `OutputDrawer` class is lightly coupled to the rest of the program, since it is programmed to the interface of the rest of the classes, it also boasts high cohesion since it only draws the output image. For the `MSOAvaloniaApp` project we also made sure to have high cohesion by separating the UI code from the core logic. The UI code only handles displaying the window, getting user input and showing output to the user. All the core logic is handled by the `MSOProgramLearningApp` project. This way we can easily change the UI in the future without having to change the core logic of the program. The `MainWindowViewModel` also has high cohesion, since it only holds the data that is displayed in the UI and handles user input. It does not hold any core logic of the program, it just calls the appropriate methods from the `MSOProgramLearningApp` project.

Changing Requirements

Our program meets all the requirements set in the assignment description, but if we were to change or add requirements, we would have to make some changes to our design. For example, if we were to add a different UI framework, we would have to change the `MSOAvaloniaApp` project to use the new framework. This would not affect the `MSOProgramLearningApp` project, since it is decoupled from the UI. We could likely reuse most of the code from the `MainWindowViewModel`, but we would have to change the way the UI is displayed and how user input is handled. If we were to add new commands to the program, we would have to add new classes implementing the `ICommand` interface in the `MSOProgramLearningApp` project. This would not affect the UI project, since the UI only calls the methods from the core project. Adding a new parser for a different file format would also be quite easy, we would simply have to add a new parser class in the `MSOProgramLearningApp` project, and have it implement the `IParser` interface. We would have to update the command parser to recognize the new commands, but this would be a small change.
Overall, our design is quite flexible and can easily accommodate changes in requirements without having to make major changes to the codebase, a harder change would be to make the grid non-square, this would require changes to the `Grid` class and possibly the `Character` class, and since a few classes use the `GetSize` method from the `Grid` class, we would have to update those classes as well.

Testing

Unit tests

We wrote quite a few unit tests to test the core program functionality after making changes. The tests are in a separate project `MSOTestProject`, which we run after making changes to the `MSOProgramLearningApp`, ensuring functionality of our program's core. We have tests for 4 major aspects of our code:

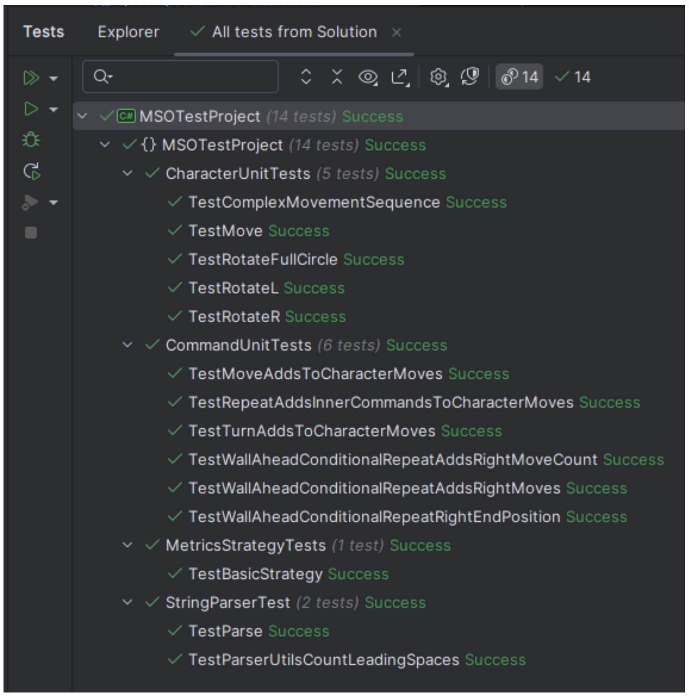
1. `CharacterUnitTests`: In this class, we test movement and rotation of the character, since this is the only logic this tests implements.
2. `CommandUnitTests`: Here we test the execution of the commands on a mock character.
3. `MeticsStrategyTests`: We use this class for testing whether our metrics calculator works for a specific list of commands.
4. `StringParserTests`: In the `StringParserTests` class we test whether or parser gives the expected output when parsing a string.

System tests

We manually tested the application by running through a few different scenarios, making sure that we got the expected output each time. These tests changed a bit over time, since we added new features to be tested. Here is a list of the different scenarios we tested:

1. Basic movement: Running the basic example from the assignment description, and checking if the character ends up in the right position and orientation, without errors.
2. Nested repeat command: We ran the advanced example from the assignment description, which includes a nested repeat command, and checked if the character ends up in the right position and orientation, without errors.
3. Metrics: We ran metrics on the examples to see if the metrics were calculated correctly.
4. Conditional repeat: We created a scenario that uses the conditional repeat command, and checked if the character ends up in the right position and orientation, without errors.
5. Catching file errors: We tried to load files with invalid formats, and checked if the program catches the errors and displays an appropriate message to the user.
6. Catching out-of-bounds errors: We created scenarios where the character would try to move out of bounds, and checked if the program catches the errors and displays an appropriate message to the user.

Final test run



```
Tests   Explorer   ✓ All tests from Solution ×
                                   14  ✓ 14
✓ MSOTestProject (14 tests) Success
  ✓ {} MSOTestProject (14 tests) Success
    ✓ CharacterUnitTests (5 tests) Success
      ✓ TestComplexMovementSequence Success
      ✓ TestMove Success
      ✓ TestRotateFullCircle Success
      ✓ TestRotateL Success
      ✓ TestRotateR Success
    ✓ CommandUnitTests (6 tests) Success
      ✓ TestMoveAddsToCharacterMoves Success
      ✓ TestRepeatAddsInnerCommandsToCharacterMoves Success
      ✓ TestTurnAddsToCharacterMoves Success
      ✓ TestWallAheadConditionalRepeatAddsRightMoveCount Success
      ✓ TestWallAheadConditionalRepeatAddsRightMoves Success
      ✓ TestWallAheadConditionalRepeatRightEndPosition Success
    ✓ MetricsStrategyTests (1 test) Success
      ✓ TestBasicStrategy Success
    ✓ StringParserTest (2 tests) Success
      ✓ TestParse Success
      ✓ TestParserUtilsCountLeadingSpaces Success
```

Figure 2: Output of all the Unit Tests

Work Distribution & Retrospective

Again the collaboration went smoothly, we basically kept working together in the same way as the previous 2 projects since that worked very well. Splitting the tasks equally and working on them together in the 'werkcolleges', and helping each other out when needed. Then continuing to work on our tasks at home and send each other messages when needed. We distributed the work as even as possible, here is a short (maybe incomplete) overview:
`Arwin`

- Layout of UI in Avalonia
- Made the `OutputDrawer` class using ImageSharp and added basic drawing functionality
- Added functionality for loading programs and grids through files
- Added buttons for loading different example programs
- Added a dropdown box for switching between different size grids
- Worked on this document

`Jan`

- Added functionality to the UI in Avalonia
- Finalised the `OutputDrawer` by drawing the character and grid walls
- Updated the Class diagram
- Added the new Commands

- Updated the Grid class to support walls
- Implemented all the Tests
- Worked on this document

Overall we really liked working together these 3 projects, and we learned a lot from it. For example, we have only made UI with Windows Forms before so we learned a lot by using Avalonia this time. It was also the first time we used Rider as our IDE. And of course we learned a whole lot of new design patterns and how to use them in practice. At last we did not encounter any major issues during this project, only a few discussions that we could resolve easily. We would like to work together again in the future.