MSO Lab Assignment 2: Programming Learning App

Jan Huls (4699610), Arwin Moormans (4965957)

October 2025

Software Design & Patterns

Below you will find a Class diagram to show the structure of the application.

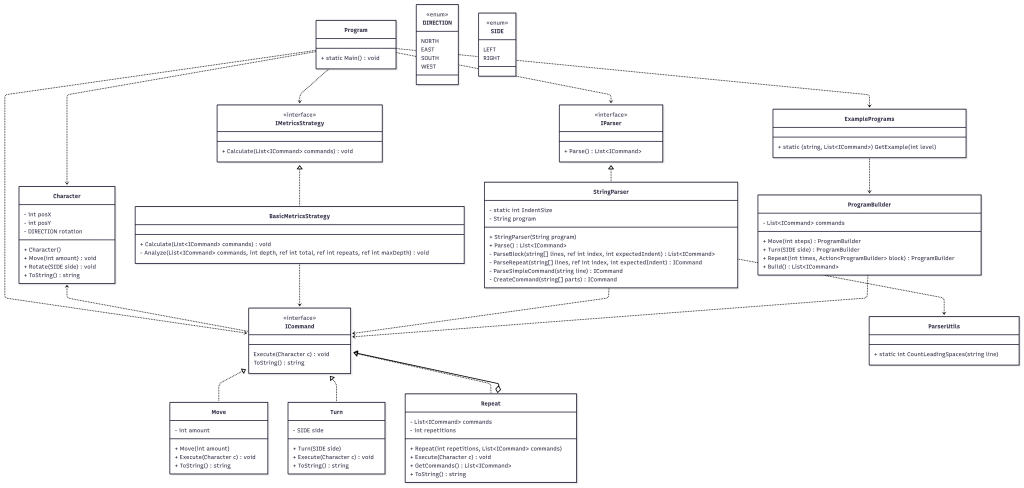

Figure 1: Class diagram of the Programming Learning App

We have used multiple patterns to structure our program.

The first pattern is the Composite pattern in the repeat command class. This class holds all the functionality of the repeat command, such as the commands it needs to reapeat and how many times. Because one of the repeated commands can be another reapeat command, we use the Composite pattern. This makes a tree like structure where non-repeat commands are leafs and other repeat commands are nodes, meaning `ICommand` is the component and `Repeat` is the composite. This way it is possible to call the Execute function of the root command and that will automaticly execute every command in the tree.

For calculating different metrics we use the Strategy pattern. This ensures that it is possible to add different kind of metrics later. By making this a Strategy pattern it is possible to easily change wich kind of metrics the user wants to see. The IMetricsStrategy is the Strategy interface and the BasicMatricsStrategy is the concrete Strategy.

At last we have used the Builder pattern to create the example programs. This way it is possible to create

Evaluation

Cohesion

Our design shows high cohesion in a couple of ways, which we will go through here.

The `ICommand` interface is implemented by all the command classes, which all have a single responsibility: executing a specific command. This means that each command class is focused on one task, leading to high cohesion within these classes. Furthermore, the `StringParser` class is only responsible for parsing strings from the given format into command objects. This single responsibility ensures that the class is cohesive and easy to maintain. The `MetricsCalculator` class is responsible for calculating metrics based on the commands executed. It uses the Strategy pattern to allow for different metric calculation strategies, which keeps the class focused on its core functionality while allowing for flexibility in how metrics are calculated. The character model in the `Character` class encapsulates all the properties and behaviors of the character, such as position and direction. This encapsulation ensures that all character-related functionality is contained within a single class, promoting high cohesion. The `ProgramBuilder` class is responsible for constructing example programs using the Builder pattern. This class is cohesive as it focuses solely on the creation of program instances, making it easy to understand and modify example programs.

Overall, our design emphasizes high cohesion by ensuring that each class has a clear and narrow responsibility, making the codebase easier to understand, maintain, and extend.

Coupling

We demonstrate low coupling in several ways throughout our design. Which we will discuss in this paragraph.

The design uses interfaces such as `ICommand` and `IMetricsStrategy` to decouple the implementation of commands and metric strategies from their usage. This allows for easy addition of new commands or metric strategies without affecting existing code, promoting low coupling. The `StringParser` class is responsible for parsing input strings into command objects, but it does not depend on the specific implementations of the commands. Instead, it relies on the `ICommand` interface, which reduces coupling between the parser and command implementations, it does however depend on the concrete command classes to instantiate them, which introduces some level of coupling, but this will always be necessary here, since the parser needs to know which commands to create.

Overall our program communicates through small interfaces, which reduces dependencies between different parts of the code. This ensures low coupling, making the system more modular and easier to maintain.

Changes

Here we will discuss 3 changes which are easy to implement in our program, and one change which is more complex.

Adding new commands is relatively easy in our design. To add a new command, one would simply need to create a new class that implements the `ICommand` interface and define the specific behavior for the command in the `Execute` method. This new command can then be integrated into the existing system without modifying other parts of the code, thanks

to the use of interfaces. If this command also needs to be parsed by the `StringParser`, the parser would need to be updated to recognize the new command string and instantiate the new command class accordingly.

Adding new metric strategies is also straightforward. To introduce a new metric calculation strategy, one would create a new class that implements the `IMetricsStrategy` interface and define the specific metric calculation logic in the `Calculate` method. The `MetricsCalculator` can then utilize this new strategy without any changes to its own implementation, allowing for easy extension of metric calculation capabilities.

Lastly, a new kind of parser could be added, for example to read from a different format. This would involve creating a new parser class that implements the `IParser` interface. The new parser would handle the specific format and convert it into command objects, similar to how the current parser works. This change would not affect other parts of the system, maintaining low coupling.

A more difficult change would be to implement an undo/redo feature. This would require a change in the `ICommand` interface to include methods for undoing and redoing commands. Each command class would need to implement these new methods, which could be a lot of work if more commands were added beforehand. Additionally, a new mechanism would need to be introduced to manage the history of executed commands, allowing users to navigate through their command history. This change would involve significant modifications to the existing command structure and could introduce additional complexity in managing command states.

Work Distribution & Retrospective

During the project, we divided the work pretty evenly. Both of us have experience with coding, so this made working together easier. We liked working together during werkcolleges, as we could discuss ideas and solve problems more efficiently, after which we would both take home tasks to do at home. There was not a clear division of tasks since we both worked together a lot, but we both worked on the code and the report. We really liked working together, as we could help each other and learn from each other. We did not encounter any major issues during the project, but we did have some minor disagreements on how to implement certain features, we were able to resolve these through discussion. We have no major changes for upcoming assignments, but we will try to communicate even more effectively to ensure that we are both on the same page.