

# **Enhancing Dafny Support in Visual Studio Code**

## **Bachelor Thesis**

Department of Computer Science  
University of Applied Science Rapperswil

Spring Term 2020

Authors: Marcel Hess, Thomas Kistler  
Advisors: Thomas Corbat, Fabian Hauser  
Expert: Guido Zraggen (Google Switzerland, Zürich)  
Third Reader: Prof. Dr. Olaf Zimmermann

---

# Assignment for Bachelor Thesis “Dafny VSCode Server Redesign”

## Marcel Hess / Thomas Kistler

---

### 1. Supervisor and Advisor

This bachelor thesis will be conducted with the Institute for Software at HSR. It will be supervised by Thomas Corbat and Fabian Hauser, HSR, IFS.

### 2. External Examiner

- Guido Zraggen - Google

### 3. Students

This project is conducted in the context of the module “Bachelorarbeit Informatik” in the department “Informatik” by

- Marcel Hess
- Thomas Kistler

### 4. Introduction

“Dafny is a programming language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs.

The Dafny programming language is designed to support the static verification of programs. It is imperative, sequential, supports generic classes, dynamic allocation, and inductive datatypes, and builds in specification constructs. The specifications include pre- and postconditions, frame specifications (read and write sets), and termination metrics.” - (Microsoft, 2019)

In a preceding bachelor thesis at HSR a Visual Studio Code plug-in to support Dafny development has been developed. It facilitates a language server for source code analysis and aids the programming with context sensitive completion suggestions, automated refactorings and performs formal verification on the fly (Dafny VSCode Server). This language server is accessed through the language server protocol (LSP). The VSCode Server relied on the DafnyServer for these analyses, which had been accessed through a proprietary API<sup>1</sup>. For a visual overview of the architecture see Figure 1. In the preceding term project by Marcel Hess and Thomas Kistler the architecture has been improved. They eliminated the unnecessary separation of the Dafny VSCode Server and the DafnyServer.

---

<sup>1</sup> <https://github.com/DafnyVSCode/Dafny-VSCode>

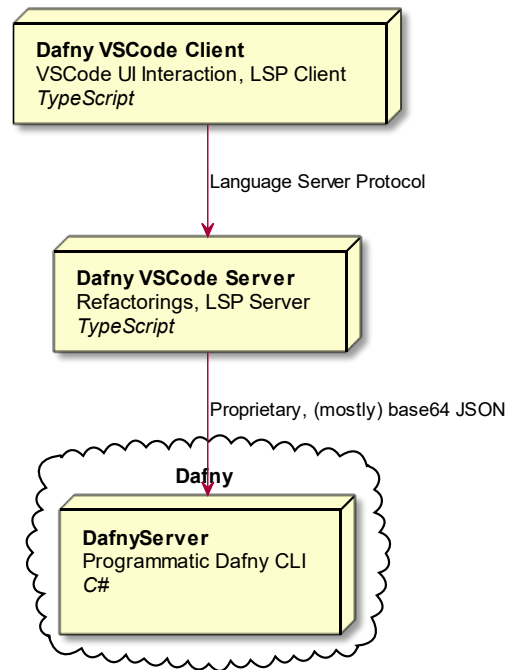


Figure 1 Initial Dafny VSCode Plug-in Architecture

## 5. Goals of the Project

The primary goal of this project is to continue the work on the Dafny VSCode Server and improve it to a releasable state. The following tasks are planned:

- Project clean up:
  - Splitting the repository of the client and the server into distinct repositories
  - Rebase the language server to the newest Dafny upstream and bring it into a state that can be merged back. This includes avoiding non-portable parts, like local paths and environment workarounds.
  - Clean up code of the server and the client (no unused/dead code). Eliminate the command line calls for the Dafny Library by accessing it directly
  - Clean up tests, but leave e2e testing be
  - Make SonarQube run
  - Clean logging
- Improve existing features according to the Conclusion of the preceding project
- Implement a proper symbol table to improve “go to definition” and support further features
- Optional: Implement new features:
  - Hover information
  - CodeLens
  - Refactorings like Extract Method
  - Automated Contract Generation

## 6. Documentation

This project must be documented according to the guide lines of the “Informatik” department [4]. This includes all analysis, design, implementation, project management, etc. sections. All documentation is expected to be written in English. The project plan also contains the documentation tasks. All results must be complete in the final upload to the archive server [5]. Two copies of the documentation must be handed-in:

- One in color, two-sided
- One in B/W, single-sided

## 7. Important Dates

<b>17.02.2020</b>	<b>Start of the semester and the bachelor thesis</b>
<b>Until 10.06.2020</b>	<b>Hand-in of the abstract to the supervisor for checking (on abstract.hsr.ch) Poster hand-in to SGI (until 10:00)</b>
<b>12.06.2020, 12.00</b>	<b>Final hand-in of the report through archiv-i.hsr.ch</b>

## 8. Evaluation

A successful bachelor thesis awards as 12 ECTS point. The estimated effort for 1 ECTS is 30 hours. (See also the module description <sup>2</sup>). The supervisor will be in charge for all the evaluation of the project.

<b>Criterion</b>	<b>Weight</b>
1. Organisation, Execution	1/6
2. Report (Abstract, Management Summary, technical and personal reports) as well as structure, visualization and language of the whole documentation	1/6
3. Content	3/6
4. Final presentation and oral exam	1/6

Furthermore, the general regulations for bachelor theses of the department “Informatik” apply.

Rapperswil, 17.02.2020

Thomas Corbat

Lecturer  
 Institut für Software (IFS)  
 Hochschule für Technik Rapperswil

---



# 1 Abstract

## Initial Situation

Dafny is a formal programming language to prove a program's correctness. In a preceding bachelor thesis, a plugin for Visual Studio Code was created to access Dafny-specific static analysis features. The plugin communicates with a language server using Microsoft's language server protocol, which standardizes communication between an integrated development environment (IDE) and a language server. The language server itself used to access the Dafny library, which features the backend of the Dafny language analysis, through a proprietary JSON-interface. In our preceding term project, the language server was integrated into the Dafny backend to render the JSON-interface obsolete.

## Objective

This bachelor thesis continues the preceding term project and contains two major goals. First, the previously implemented prototype has to be improved in usability, stability and reliability. Second, a symbol table has to be implemented to facilitate the development of symbol-oriented functionality like *Rename* or *AutoCompletion*. The symbol table is supposed to open a wide range for further development.

## Result

A symbol table was created with an adaptive data structure. Every symbol contains information about its parent, its children and its declaration. This keeps navigation within the symbol hierarchy very simple. The implemented features *GoToDefinition*, *Rename*, *CodeLens*, *HoverInformation* and *AutoCompletion* benefit from the symbol table and are no longer required to analyze the code themselves and provide better results. The symbol table can be used for future extensions, such as *AutoFormatting* or *CodeHighlighting*. Apart from features based on the symbol table, pre-existing functionality was revisited as well to improve the overall software quality. The final product has been deployed to the Visual Studio Code marketplace and is publicly available.

## Table of Contents

<b>1 Abstract</b>	<b>4</b>
<b>2 Management Summary</b>	<b>9</b>
2.1 Dafny	9
2.2 Language Server Protocol	9
2.3 Initial Solution	9
2.4 Goals	10
2.5 Results	10
2.6 Outlook	13
<b>3 Introduction</b>	<b>14</b>
3.1 Dafny	14
3.2 Language Server Protocol	15
3.3 Initial Solution and Motivation	16
3.4 Goals	17
3.5 Content References from the Pre-Existing Term Project	18
<b>4 Analysis</b>	<b>19</b>
4.1 Language Server Protocol	19
4.1.1 Message Types	20
4.1.2 Language Features	20
4.1.3 Communication Example	21
4.1.4 Message Example	22
4.2 OmniSharp	22
4.2.1 Basic OmniSharp Usage	22
4.2.2 Custom LSP Messages	23
4.3 Visual Studio Code Plugin	24
4.4 Dafny Language Features	25
4.4.1 Modules	25
4.4.2 Functions and Methods	26
4.4.3 Hiding	28
4.4.4 Overloading	28
4.4.5 Shadowing	28
4.5 Abstract Syntax Tree	30
4.6 Dafny's Abstract Syntax Tree	31
4.6.1 Requirements for the Symbol Table	31
4.6.2 Dafny Symbols	31
4.7 Dafny Expression and Statement Types	33
4.8 Refactoring of the Given Dafny AST Implementation	35
4.9 Prototype Functionality and Flaws to Improve	36
4.9.1 Syntax Highlighting	36
4.9.2 Verification	36
4.9.3 Compile	37
4.9.4 CounterExample	37
4.9.5 AutoCompletion for Identifiers	38
4.9.6 GoToDefinition	39
4.9.7 CodeLens	40
4.10 Continuous Integration (CI) of the Prototype	41
4.10.1 SonarQube	41
4.10.2 Tests	41



4.10.3 Docker	41
4.10.4 Security Aspects	42
<b>5 Design</b>	<b>43</b>
5.1 Technologies	43
5.2 Client	43
5.2.1 Initial Situation	43
5.2.2 New Architecture	44
5.2.3 Components	46
5.2.4 Logic	48
5.2.5 Types in TypeScript	49
5.3 Server	49
5.3.1 Main Component and Handlers	50
5.3.2 Core	51
5.3.3 CustomDTOs	52
5.3.4 Workspace Manager	52
5.3.5 Dafny Access	53
5.3.6 SymbolTableManager	54
5.3.7 Resources	55
5.3.8 Tools	55
5.3.9 Workflow Overview	56
5.4 Symbol Table	58
5.4.1 Symbol Table Design	58
5.4.2 Visitor Pattern	59
5.4.3 Global Symbol Table vs Symbol Table per File	60
<b>6 Implementation</b>	<b>61</b>
6.1 Client	61
6.1.1 Better Encapsulation Through Interfaces and Modules	61
6.1.2 Client Modules in Detail	63
6.1.3 Encapsulation of the VSCode Components	70
6.1.4 Download of the Dafny Language Server	71
6.1.5 Configurability	72
6.2 Server	73
6.2.1 Server Launch	74
6.2.2 Handler	75
6.2.3 Core	76
6.2.4 Workspace	76
6.2.5 DafnyAccess	77
6.2.6 Utilities	79
6.3 Symbol Table	80
6.3.1 Symbol Information	80
6.3.2 Symbol Table Creation	81
6.3.3 Symbol Table Navigator	84
6.3.4 Symbol Table Manager	87
6.3.5 Symbol Table Utilities	87
6.4 Features	87
6.4.1 AutoCompletion	87
6.4.2 CodeLens	91
6.4.3 Compilation	93
6.4.4 CounterExample	93
6.5 Mono Support for macOS and Linux	94



6.6	Testing . . . . .	95
6.6.1	Unit Tests . . . . .	95
6.6.2	Integration Tests . . . . .	97
6.7	Usability Test . . . . .	100
6.7.1	Code Documentation for Classes and Methods . . . . .	100
6.7.2	Cleaner HoverInformation . . . . .	101
6.7.3	Automatic Triggering of AutoCompletion . . . . .	101
6.7.4	Error Message Without Brackets . . . . .	101
6.7.5	Compile Improvements . . . . .	102
6.8	Continuous Integration (CI) . . . . .	103
6.8.1	SonarQube . . . . .	103
6.8.2	Client End-to-End Tests . . . . .	104
6.8.3	Static Program Analysis and Formatting for TypeScript . . . . .	104
6.8.4	Docker . . . . .	104
<b>7</b>	<b>Results</b>	<b>105</b>
7.1	Features for Dafny Developers . . . . .	105
7.1.1	Syntax Highlighting . . . . .	105
7.1.2	Verification . . . . .	106
7.1.3	Compile . . . . .	107
7.1.4	CounterExample . . . . .	107
7.1.5	HoverInformation . . . . .	108
7.1.6	GoToDefinition . . . . .	109
7.1.7	Rename . . . . .	110
7.1.8	CodeLens . . . . .	110
7.1.9	Automatic Code Completion . . . . .	112
7.2	Simplicity of Further Development . . . . .	113
7.2.1	Symbol Table . . . . .	113
7.2.2	Server Architectural Improvements . . . . .	114
7.2.3	Client Architectural Improvements . . . . .	115
7.3	Deployment . . . . .	116
7.3.1	Pull Request to Dafny . . . . .	116
7.3.2	Publication to the VSCode Marketplace . . . . .	116
7.3.3	Download of the Dafny Language Server . . . . .	117
7.3.4	Easy Installation . . . . .	117
7.4	Performance of the Language Server . . . . .	118
7.5	Metrics and Quality Measures . . . . .	119
7.5.1	Server . . . . .	119
7.5.2	Client . . . . .	121
<b>8</b>	<b>Conclusion</b>	<b>123</b>
8.1	Project Summary . . . . .	123
8.2	Deployment . . . . .	124
8.3	Outlook . . . . .	124
8.3.1	Completion of the Visitor . . . . .	124
8.3.2	LSP Extensions . . . . .	124
8.3.3	Refactorings . . . . .	125
8.3.4	Dafny Specific Functionality . . . . .	125
8.3.5	More IDEs . . . . .	125
8.4	Further Achievements . . . . .	125



<b>9 Project Management</b>	<b>127</b>
9.1 Time Management	127
9.1.1 Time Spent From Each Student	127
9.1.2 Schedule and Scope	130
9.2 Effects of the COVID-19 Pandemic	132
9.2.1 Meetings	132
9.2.2 Division of Labour	132
9.3 Quality Aspects	132
9.3.1 Code Reviews	132
9.3.2 Continuous Integration (CI)	133
9.3.3 Static Code Analysis	134
9.3.4 Test Coverage	135
<b>Glossary</b>	<b>137</b>
<b>References</b>	<b>138</b>
<b>Developer Documentation</b>	<b>140</b>
<b>Project Plan</b>	<b>164</b>
<b>Usability Test with Remo Herzog</b>	<b>178</b>
<b>Usability Test Code</b>	<b>183</b>

## 2 Management Summary

This chapter contains a brief overview of the bachelor thesis at hand. First, the technologies Dafny and language server protocol are explained to provide the necessary context. This is followed by a presentation of the objectives, results achieved and future prospects.

### 2.1 Dafny

Dafny is formal programming language. Within Dafny code, a developer has the option to state a so-called precondition. This is a fact that must be true at the beginning of the code. Once the code has been executed, the developer may state a postcondition. The postcondition is, contrary to the precondition, something that is true at the end of code. Dafny contains an internal engine to automatically prove postconditions. This means, given the precondition holds, the code will manipulate data only thus far, so that also the postcondition is valid. Dafny offers optimizations for such proofs as an advantage over other programming languages.

### 2.2 Language Server Protocol

The language server protocol (LSP) is a specification for communication. It regulates data exchange between an IDE and a so-called language server. This ensures a separation between the integrated development environment (IDE) and the actual programming language support of the language server. With this standardized protocol, a language server can be accessed by different IDEs.

A plugin for the IDE "Visual Studio Code" (VSCode) was implemented during this bachelor thesis. Once a user writes Dafny code using the plugin, the code is then transferred to the language server. The language server is now responsible for analyzing the code and provide proper replies to the editor whenever the editor requests something. A request could, for example, be to show information about a variable. The server has to provide all the necessary information to the IDE.

As a result of using the LSP, it is possible that Dafny can be supported by other IDE's with minimal effort.

### 2.3 Initial Solution

In a previous bachelor thesis by Markus Schaden and Rafael Krucker, a LSP client-server infrastructure for Visual Studio Code was created to support Dafny [1]. The plugin was particularly appreciated by the "HSR Correctness Lab" [2] to make coding in Dafny easier. The language server and the Dafny backend were separated into two different components. This had the disadvantage that they had to communicate over a proprietary interface. This solution was not optimal, since the Dafny backend was not accessed directly. Implementing new features was very time consuming.

In the preceding semester project [3], the language server was merged into the Dafny backend, so that the proprietary interface became obsolete as you can see in figure 1. Dafny could be accessed directly by the new software architecture. Within this bachelor thesis, the work of the existing solution will be continued.

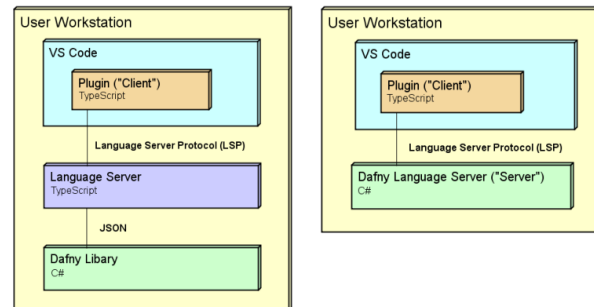


Figure 1: Architecture Before (Left) and After (Right) the Prototype was Created

## 2.4 Goals

At HSR, Dafny is taught within the course *Software Engineering*. Students have to make their first steps in Dafny, using a simple tutorial. To provide the students with the necessary support, they shall profit by common IDE features like:

- *SyntaxHighlighting* to make the readability more pleasant.
- *ErrorHighlighting* to correct faulty code efficiently.
- *CompileAndRun* to execute Dafny programs and check the output.
- *AutoCompletion* to code efficiently.
- *GoToDefinition* to be able to view definitions quickly.
- *CounterExample* to efficiently remedy faulty pre- and postconditions.

On the other hand, the plugin shall also be used in a professional environment in the future. Thus, more advanced features have to be supported. To facilitate the future development of such features, the codebase shall be refactored to achieve a clean and maintainable state. This especially involves a clear-cut architectural layout with well-organized dependencies.

Aside from a clean architecture, a major goal is the creation of a custom symbol table. A symbol can be a variable, a function or a class. The symbol table contains information about these items appearing in the Dafny code. It shall allow very simple navigation within the Dafny code, for example to locate the declaration of a variable. Once the symbol table has been generated, navigation within the code will be very easy for the language server. This has the advantage that many features can be realized with almost no logic. For example, the feature *GoToDefinition* just has to ask the symbol table where the definition of the symbol is. Consequently, a well implemented symbol table will also facilitate the development of further features.

## 2.5 Results

Within this bachelor thesis, the development of the pre-existing language server and its VSCode client was continued. Significant improvements could be achieved, which are described in this section.

The following features are supported as planned:

- Syntax highlighting.
- Verification: highlighting of errors and warnings.
- Compilation of Dafny code and execute it.

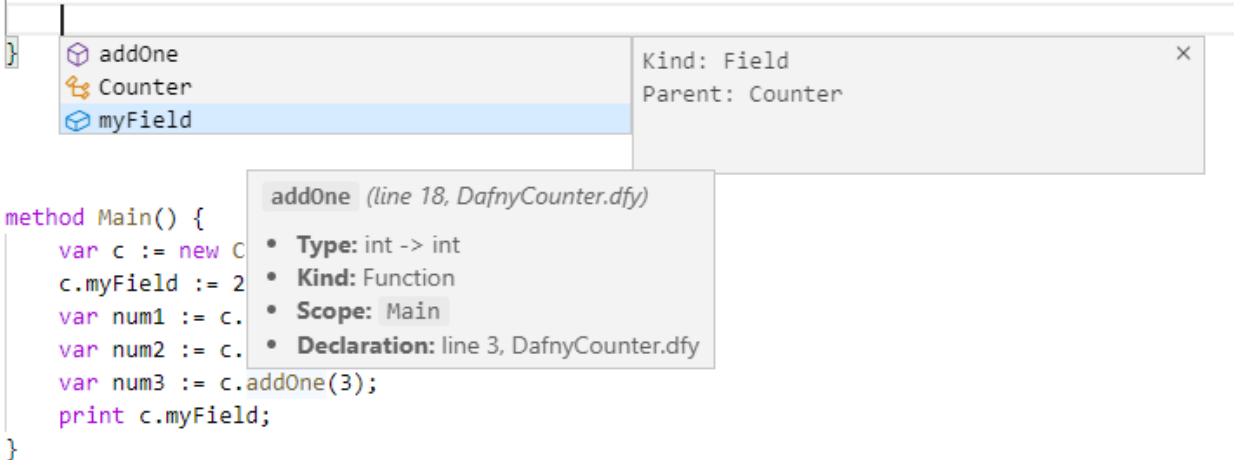
- Providing counter examples.

Additionally, the following functions could be implemented thanks to the new symbol table:

- *AutoCompletion* suggestion.
- *GoToDefinition*.
- *CodeLens* counts the usages of classes and functions and lets the developer show the usages with one click.
- *Rename* allows to rename symbols efficiently and reliably.
- *HoverInformation* provides the developer with useful information about symbols.

```
1 reference to Counter
class Counter {
  constructor(){}
  3 references to addOne
  function method addOne(x: int): int {
    x+1
  }
  var myField : int;
}

method Main() {
  var c := new C
  c.myField := 2
  var num1 := c.
  var num2 := c.
  var num3 := c.addOne(3);
  print c.myField;
}
```



addOne (line 18, DafnyCounter.dfy)

- **Type:** int -> int
- **Kind:** Function
- **Scope:** Main
- **Declaration:** line 3, DafnyCounter.dfy

Kind: Field  
Parent: Counter

Figure 2: The Features *HoverInformation* and *AutoCompletion* Profit From the Symbol Table

In figure 2, a short code example is shown. The code fragment would build a symbol table as it is abstractly represented in figure 3. The features shown in figure 2 such as *AutoCompletion* and *HoverInformation* directly use the offered symbol table.



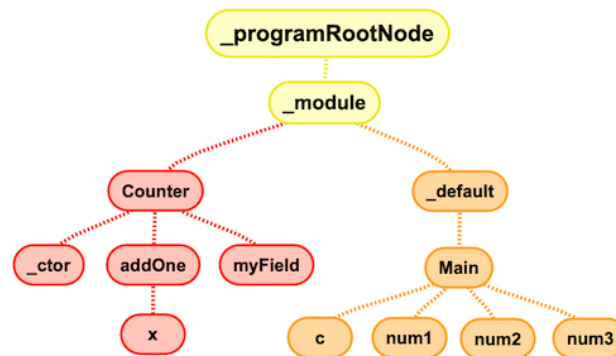


Figure 3: Symbol Table to Represent all Symbols Inside the User Code

Dafny developers can now benefit by a plugin, which provides a great user experience, but also gained much robustness compared to the prototype from our previous project.

Alongside the improvements in features, many internal aspects were also improved. This benefits developers who want to extend the Dafny language server in the future.

A component called "Dafny translation unit" was completely re-visited and simplified. The component accesses any Dafny functionality. For example, instead of passing Dafny options as an array of strings, they are now set by directly accessing Dafny's config class. Any results provided by the Dafny translation unit are buffered for later re-use at compilation or to create the symbol table. This makes the implementation significantly more performant.

The targeted symbol table could be implemented for the most important Dafny language features. Various challenges had to be accommodated, including the handling of default scopes, default classes, inheritance, external file import or variable hiding.

The symbol table opens the option to implement many more features than currently provided. For example, the LSP offers a highlight request, marking occurrences of a symbol. Any information required for this feature are already provided by the symbol table. Thus, adding this feature would be very simple.

Besides changes in the server, the VSCode client is now as lightweight as possible. This makes the adaption to other IDE's very simple. The server is now able to create a symbol table containing any information required for the language analysis features. Pre-existing features and algorithms were improved to gain more reliability and a better user experience.

## 2.6 Outlook

While the quality of the features as well as the general code quality could be massively improved, the project is not in a final state. Functionality of the project could be improved even further. Ideas include:

- Automatic generation of contracts.
- Debugging for Dafny.
- Create clients for other IDE's, which connect to our Dafny language server.

Alongside the widening of the feature range, it is necessary to complete the visitor. The visitor is the component generating an internal symbol table within the language server. Currently, only the most important Dafny language features are supported. For example, custom datatypes<sup>1</sup>, as used in formal programming languages, are not supported.

Nevertheless, the plugin is of a nice quality and is ready for deployment into the VSCode marketplace, once multi-platform support is implemented. Thus, students can work with it and make their first steps in the Dafny programming language using our plugin.

---

<sup>1</sup>for example `datatype Tree<T> = Empty | Node(left : Tree, root : T, right : Tree)`

## 3 Introduction

This chapter describes the initial solution of the project, as well as the motivation and the goals of the thesis in more detail. To provide the reader with the necessary context, the technologies touched by this bachelor thesis are explained at the beginning. This mainly concerns Dafny and the language server protocol (LSP).

### 3.1 Dafny

Dafny is a compiled language optimized to prove formal correctness [4]. It is based on *Boogie* [5], which uses the Z3 [6] automated theorem prover for discharging proof obligations [4]. That means that a programmer can define a precondition - a fact that is just given at the start of the code. The postcondition on the other hand is a statement that must be true after the code has been executed. Just as with the precondition, the postcondition is also defined by the programmer. In other words, it can be proven that under a given premise, the code will manipulate data only thus far, so that also the postcondition will be satisfied. Dafny will formally ensure this. If it is not guaranteed that the postcondition holds, an error is stated.

Listing 1 shows an example. The value *a* is given, but it is required to be positive. This is the precondition. In the method body, the variable *b* is assigned the negative of *a*. We ensure that *b* must be negative, which is the postcondition.

---

```
1 method Negate(a: int) returns (b: int)
2 requires a > 0
3 ensures b < 0
4 {
5     b := -a;
6 }
```

---

Listing 1: Simple Dafny Example

This example is of course trivial. In a real project, correctness is not always that obvious. With Dafny, a programmer can be sure if the program is correct in a logical way. Since the proof is done with formal, mathematical methods, correctness is guaranteed.

If Dafny is unable to perform a proof, the user can assist by creating lemmas. Lemmas are mathematical statements. For example, a lemma could be that a factorial number is never zero. If we define a simple function `Factorial`, and later divide through the result of `Factorial`, Dafny will state that this might be a division by zero error. But if we assert that a factorial number can never be zero, verification can be completed successfully. This is illustrated in listing 2.

---

```
1
2 function Factorial(n: nat): nat
3 {
4     if n == 0 then 1 else n * Factorial(n-1)
5 }
6
7 lemma FactorialIsNotZero(n: nat)
8 ensures Factorial(n) != 0
9 {}
10
```

```
11 function Foo(n: nat): float
12 {
13     FactorialIsNotZero(n);
14     100 / Factorial(n)
15 }
```

---

Listing 2: Lemma Example for Factorial

## 3.2 Language Server Protocol

The language server protocol (LSP) was created to unify communication between an integrated development environment (IDE) and a language server. It specifies requests, such as *AutoCompletion*, *Rename* or *GoToDefinition*. If the user performs an action like renaming a symbol, the IDE will send the proper request to the language server. The message format is specified by the LSP and bases on JSON. JavaScript Object Notation (JSON) is a language independent data format [7].

The language server is responsible to calculate a proper result. For the example of a *Rename* request, the answer contains the information where to apply the renaming. It is the task of the server to analyze the source code and provide a *Rename* response with respect to language specific rules.

Since the language server is independent of the client, a language server can be used from within multiple IDE's. To provide support for another IDE, just the client has to be adjusted. Since all logic is contained within the server, this can be done with minimal effort. A developer only has to set up the connection to the language server and has to implement UI-tasks for the newly supported IDE.

The most important requests the LSP supports incorporate:

- Transfer code to server.
- Show errors and warnings.
- Rename a symbol.
- Go to the definition of a symbol.
- Show completion suggestions.
- Perform a refactoring.
- Show usages of a code item.

These are features that are commonly used by programmers, independent of the language.

Apart from these standard features, Dafny has the option to show a counter example, if a postcondition is violated. This request is not natively supported by the LSP. However, own custom requests and data responses can easily be added to the LSP. These have to be handled separately within the client though, since the LSP is not automatically displaying the result.

Because of these advantages, the project is based on this protocol. In chapter 4, which features will be implemented in the underlying project will be discussed.

### 3.3 Initial Solution and Motivation

In a previous bachelor thesis by Markus Schaden and Rafael Krucker, a LSP client-server infrastructure for Visual Studio Code was created to support Dafny [1]. The plugin was particularly appreciated by the "HSR Correctness Lab" [2] to make coding in Dafny easier. Its development was continued within the HSR under the lead of Fabian Hauser. In the following text, the existing project will be referred to as the *pre-existing project*. The pre-existing project offered a LSP-client for Visual Studio Code, which connected to a language server. Both, the language server and the plugin, were written in TypeScript. To communicate with the Dafny backend, the language server used a proprietary JSON-interface. Information provided by Dafny was parsed from the console. Functionality was therefore limited to the Dafny console output. Due to the preparation of this console output and subsequent preparation for LSP communication, it was very complex and time consuming to implement additional features.

Marcel Hess and Thomas Kistler continued the development of the project within a semester thesis [3]. The language server was migrated to C# and could be integrated into the Dafny backend as you can see in figure 4. Any Dafny functionality was made directly accessible and the proprietary JSON-interface, as well as console parsing, could be omitted. All features were re-implemented to satisfy the new architectural layout. The result of this semester thesis will be referred as the *prototype*. If the actual text of the preceding semester thesis is targeted, it will be called the *preceding semester thesis* throughout this text.

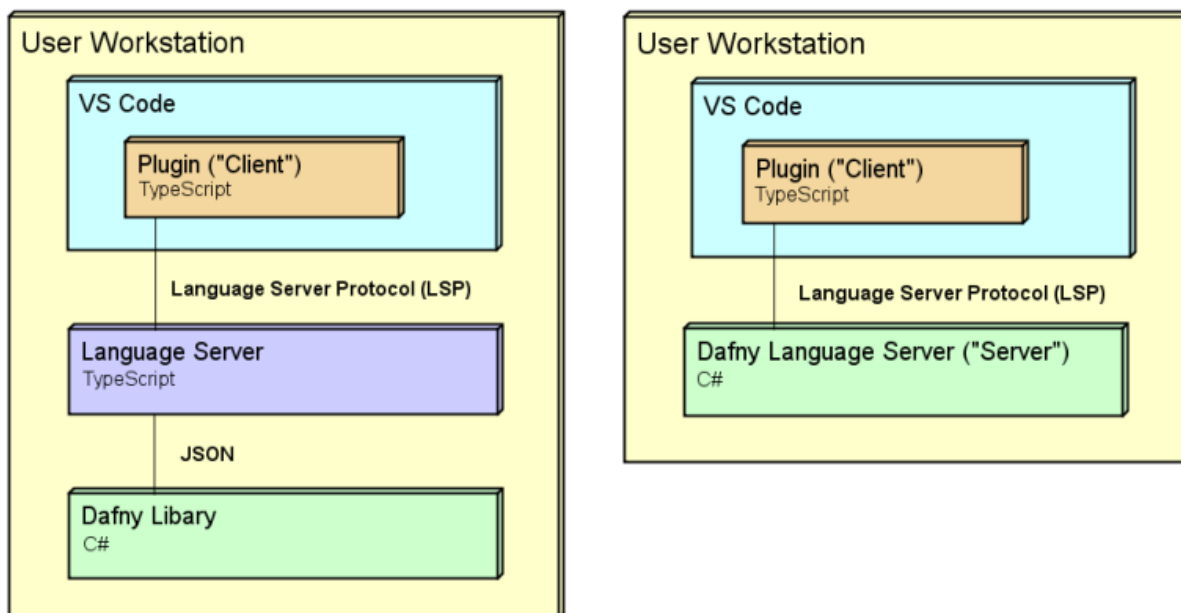


Figure 4: Architecture Before (Left) and After (Right) the Prototype was Created

### 3.4 Goals

At HSR, Dafny is taught within the course *Software Engineering*. Students have to make their first steps in Dafny, using a simple tutorial. To provide the students with the necessary support, they shall profit by common IDE features like

- *SyntaxHighlighting* to make readability more pleasant.
- *ErrorHighlighting* to correct faulty code efficiently.
- *CompileAndRun* to execute Dafny programs and check the output.
- *AutoCompletion* to code efficiently.
- *GoToDefinition* to be able to view definitions quickly.
- *CounterExample* to efficiently remedy faulty pre- and postconditions.
- *CodeLens* to count the usages of classes and functions.

Some of these features have already been implemented in a basic variant in the prototype. But almost all these features need to be improved. This will be realized by the implementation of a custom symbol table for features that base on code symbol navigation. The symbol table will contain a well organized tree structure, so that navigation can be done efficiently. Each symbol will have direct links to its parent, children and declaration for very fast access. Compared to the prototype, the pre-existing implementations can be extracted into the symbol table, reducing the lines of code (LOC) and increasing performance.

Additionally, new features based on the symbol table can be added. This targets especially *HoverInformation* and *Renaming*. Further optional features are automated *ContractGeneration* and refactorings like *ExtractMethod*.

On the other hand, the plugin will also be used in a professional environment. Thus, more advanced features have to be supported in the future. For example, if the new symbol table provides information about the scope-depth, the indentation width can be deduced for a feature like auto formatting.

To facilitate the development of further features, the code will be refactored to achieve a clean state. This especially involves a clear architectural layout with dependencies that only point downwards. This should make it as easy as possible for other developers to enrich the existing Dafny language server with additional features.

In addition, process optimizations should make the work for future developers easier. This includes the recording of SonarQube for the Dafny language server for static code analysis, as well as a clean logging, which should be helpful for troubleshooting.

Furthermore, our plugin should reach a stable state and be published into the VSCode marketplace. Dafny developers shall profit by an increased value compared to the pre-existing solution. The plugin has to be platform independent and easy to install.

The IDE-independent Dafny language server should be ready to be merged back into the original Dafny repository. This means that the server has to be extracted into a separate git repository, separated from the client. At the end of the project, a pull request into the Dafny repository should be possible.

### 3.5 Content References from the Pre-Existing Term Project

The following sections originated in the pre-existing semester thesis "Dafny Server Redesign" [3] and were re-included into this document for the sense of a comprehensive documentation. Minor changes, such as typos or the adjustment of the preceding project, are not mentioned separately.

Dafny and the language server protocol are fundamental technologies of this project. Our supervisors were following the development of the preceding semester thesis and are familiar with these technologies. However, the external expert, as well as the third reader were not involved. To provide them with a comprehensive understanding of all necessary context, the analysis of those technologies done in the preceding thesis was copied into this document.

- Chapter 1 Abstract
  - Paragraph 1 about Dafny
  - Paragraph 2 about the language server protocol communication
- Chapter 2 Management Summary
  - Section 2.1 about Dafny
  - Section 2.2 about the language server protocol communication
- Chapter 3 Introduction
  - Section 3.1 except for the paragraph about lemmas
  - Section 3.3, partially
- Chapter 4 Analysis
  - Section 4.1
  - Section 4.2, reworked, better example
  - Section 4.9, adoption of the starting positions from the results. How the improvements are to be implemented is new.
- Chapter 7 Results
  - Section 7.1.1
- Chapter 9 Project Management
  - The whole chapter is structurally based on the pre-existing term project. Partially, sentence fragments from general introductions were taken over. However, the data as well as the conclusions from it are new.

## 4 Analysis

Since this thesis is a sequel of the preceding semester thesis, work could be directly continued. Many topics were already analyzed in the prequel. To provide the reader with a comprehensive knowledge base about the language server protocol, sections 4.1 and 4.2 are repeated from the semester thesis.

First, we will start by explaining the language server protocol in detail. It is responsible for the communication between the IDE and the language server. Then, studies about OmniSharp follow, which offer an LSP implementation for Visual Studio. On the client side, plugins have to follow a specific data structure and provide certain entry points. This is discussed afterwards.

The basics about Dafny were already mentioned in section 3.1. In this chapter, we will go more into detail and study the Dafny-specific compilation process with its abstract syntax tree nodes.

After this technical analysis, the prototype is described. This way, the reader is informed what the state of the product was at the beginning of this thesis. Finally, research done in relation to CI is described.

### 4.1 Language Server Protocol

The language server protocol (LSP) is a JSON-RPC based protocol to communicate between an IDE and a language server [8]. In 2016, Microsoft started collaborating with Red Hat and Codenvy to standardize the protocol's specification [8]. The goal of the LSP is to untie the dependency of an IDE with its programming language. That means that once a language server is available, the user is free in the choice of the IDE, as long as it offers a client instance that is able to communicate with the server. The user can then use a variety of features, as long as the language server is capable to handle them. Those features can for example be

- *AutoCompletion*
- *HoverInformation*
- *Rename*
- *GoToDefinition*

and much more. A full list can be found within the LSP specification [9]. Custom requests, for example for features like *Compile* or *CounterExample*, can also be added to the LSP [8]. The major advantage of this concept is that the plugin can be kept very simple. The relevant information is delivered by the language server, which is IDE-independent. Figure 5 from the VSCode extension guide illustrates these benefits .



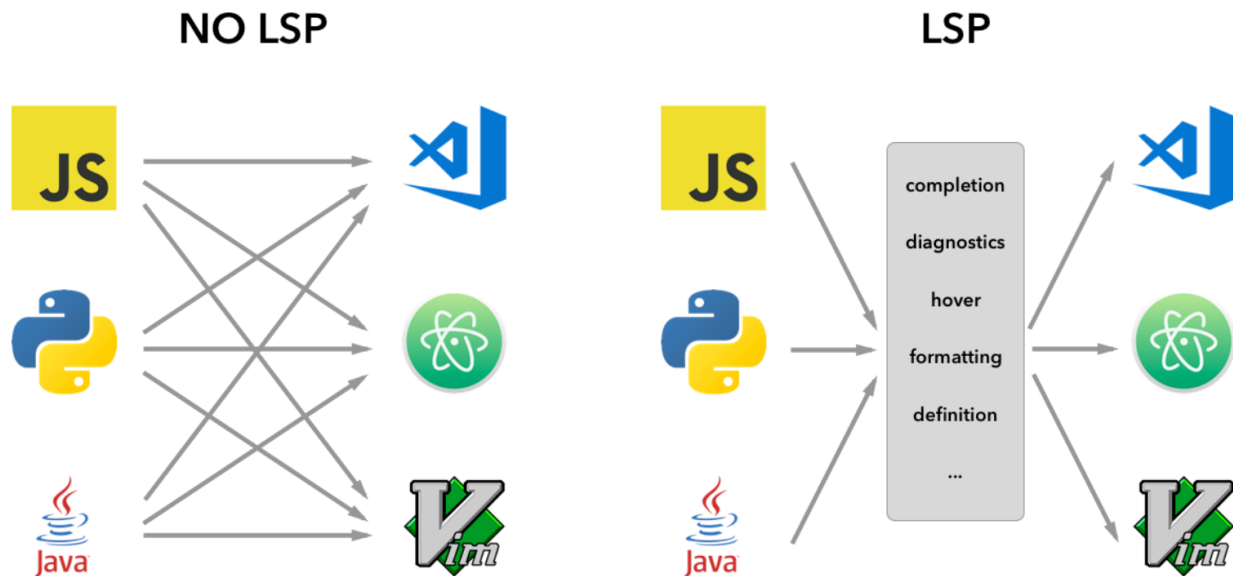


Figure 5: Communication Benefit of LSP, Image from *visualstudio.com* [10]

#### 4.1.1 Message Types

The LSP supports three types of messages.

- Notification: One-way message, for example for a window notification.
- Request: A message that expects a response.
- Response: The response to a request.

Each message type can be sent from both sides. From client to server and from server to client.

#### 4.1.2 Language Features

Aside regular communication, such as text document synchronization, the language server protocol specifies a large number of language features [9]. The requests are collected in the following itemization:

- |                      |                        |                    |
|----------------------|------------------------|--------------------|
| • completion         | • documentHighlight    | • formatting       |
| • completion/resolve | • documentSymbol       | • rangeFormatting  |
| • hover              | • codeAction           | • onTypeFormatting |
| • vsignatureHelp     | • codeLens             | • rename           |
| • declaration        | • codeLens/resolve     | • prepareRename    |
| • definition         | • documentLink         | • foldingRange     |
| • typeDefinition     | • documentLink/resolve | • selectionRange   |
| • implementation     | • documentColor        |                    |
| • references         | • colorPresentation    |                    |

Certain functions, which are needed for the Dafny plugin, are not offered by the standard. Fortunately the LSP can be extended with custom requests. This concerns the following features:

- *Compile*: Informs the server about the file to compile and compilation arguments. As a result, the server delivers a success or error message.
- *CounterExample*: The client sends a file identifier to the server and receives counter examples in response.

Custom LSP messages are further discussed in section 4.2.2.

### 4.1.3 Communication Example

The basic concept of the LSP is to inform the server about any client action. For example, when the user is opening a document, the notification `textDocument/didOpen` is sent. The message contains the URI of the opened document, so that the language server can load the source code. Whenever the user is typing, the notification `textDocument/didChange` is sent. It contains any changes that were made.

The language server on the other hand can verify the opened or changed document and test it for errors. If errors are found, the server is able to send a `textDocument/publishDiagnostics` notification back to the client. The client may now underline the erroneous code range [9]. Since the protocol is standardized, the client supporting the LSP is natively able to handle the diagnostics notification, without any actions necessary by a developer.

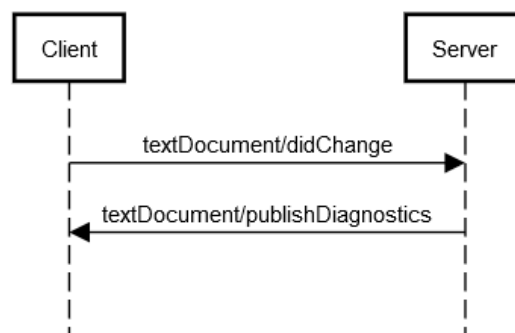


Figure 6: LSP Example Communication

#### 4.1.4 Message Example

The following message in listing 3 is a `textDocument/publishDiagnostics` notification as it appears in figure 6. It states that on line 4, from character 12 to 17, there is an assertion violation. Note the JSON-based message format.

---

```
1 [12:45:29 DBG] Read response body
2 {
3   "jsonrpc":"2.0",
4   "method":"textDocument/publishDiagnostics",
5   "params":{
6     "uri":"file:///D:/[...]/faill.dfy",
7     "diagnostics":[
8       {
9         "range":{
10          "start":{
11            "line":4,
12            "character":12
13          },
14          "end":{
15            "line":4,
16            "character":17
17          }
18        },
19        "severity":1,
20        "code":0,
21        "source":"file:///D:/[...]/faill.dfy",
22        "message":"assertion violation"
23      }
24    ]
25  }
26 }
```

---

Listing 3: LSP Message Example

## 4.2 OmniSharp

To create a language server, a proper LSP implementation is required. OmniSharp offers support for C# [11] in the form of a NuGet package. NuGet is the package manager in the Microsoft universe [12]. A language server client that can be used for testing is also offered.

### 4.2.1 Basic OmniSharp Usage

Mr. Martin Björkström published a comprehensible tutorial about OmniSharp's language server protocol implementation [13]. The tutorial provides the user with all the required knowledge to set up a language server in C#. Besides the setup of the server, it also illustrated how to create message handlers, for example for auto completions or document synchronization.

---

```
1 public class AutoCompletion : ICompletionHandler
2 {
3     public Task<CompletionList> Handle(CompletionParams request,
4         CancellationToken cancellationToken)
5     {
6         //...
7     }
8 }
```

---

Listing 4: LSP Handler Implementation

Listing 4 illustrates that the user simply has to implement an interface provided by OmniSharp. Within the `request` parameter, all required information is passed to the handler. For *AutoCompletion*, this is the file and the cursor position, as well as some context information, how the completion event was triggered. The task of the language server is to figure proper suggestions and return them in the form of a `CompletionList` back to the client.

Since OmniSharp is open source, we could find all available interfaces and thus all available handlers in their git repository [14]. Aside from the official LSP specification, this collection is very helpful to perceive LSP's possibilities.

#### 4.2.2 Custom LSP Messages

The current problem domain does not only require premade LSP messages like `completion` or `publishDiagnostics`, but also custom requests such as `CounterExample`, which is Dafny-specific. Such a message is not natively supported by the language server protocol. Since no example or documentation could be found on custom messages, Martin Björkström was contacted in the OmniSharp Slack channel [15]. Mr. Björkström and his team were able to quickly provide the solution for this issue.

The server can simply register custom handlers by implementing the correct interfaces. The following three items have to be specified:

- Name of the message, e.g. `CounterExample`
- Parameter type, e.g. `CounterExampleParams`
- Response type, e.g. `CounterExampleResults`

The parameter and response types can be custom classes and more flexibility. The following skeleton code demonstrates how a custom request handler can be implemented:

```
1 public class CounterExampleParams : IRequest<CounterExampleResults> { ... }
2 public class CounterExampleResults { ... }
3
4 [Serial, Method("CounterExample")]
5 public interface ICounterExampleHandler : IJsonRpcRequestHandler<
    CounterExampleParams, CounterExampleResults> { }
6
7 public class MyHandler : ICounterExampleHandler
8 {
9     public async Task<CounterExampleResults> Handle(CounterExampleParams
        request, CancellationToken c)
10    {
11        CounterExampleResults r = await DoSomething(request);
12        return r;
13    }
14 }
```

Listing 5: LSP Handler Implementation for *CounterExample*

### 4.3 Visual Studio Code Plugin

To provide a plugin for Visual Studio Code, a custom extension must be programmed. To get a rough understanding of how to build such an extension and how to run and debug an own plugin, a tutorial was already studied during the preceding semester thesis. In the tutorial "Your First Extension", basic concepts for own plugins were learned [16]. Examples for integrating a Language Server were found on the advanced examples page "Extension Guides" [17].

Creating a simple plugin extension generates the folder structure shown in figure 7.

```
.
├── .vscode
│   ├── launch.json // Config for launching and debugging the extension
│   └── tasks.json // Config for build task that compiles TypeScript
├── .gitignore // Ignore build output and node_modules
├── README.md // Readable description of your extension's functionality
├── src
│   └── extension.ts // Extension source code
├── package.json // Extension manifest
└── tsconfig.json // TypeScript configuration
```

Figure 7: VSCode Extension File Structure

The primary components are the two TypeScript files for the extension manifest and the extension source code.

Basic plug-in configurations are specified in the manifest. These include, for example:

- The name of the plugin.
- The version.
- Command registrations.

- The specification of which files the plugin should be active on.

The plugin launches by calling the method `activate` within `extension.ts`.

---

```
1 import * as vscode from 'vscode';
2
3 // This method is called when the extension is activated.
4 export function activate(context: vscode.ExtensionContext) {
5     // entry point
6 }
```

---

Listing 6: `extension.ts`

The logic contained in the client is supposed to be minimal. The primary logic is handled by the Dafny language server. To communicate with the language server, the language server must first be started locally as a process by the plugin.

In the prototype, it is assumed that the server is located in a specific directory relative to the client. Since the Dafny language server is not part of the plugin delivery to the VSCode marketplace, this is an issue for the plugin release. An automated download of the newest release of the Dafny language server was missing. Therefore, a goal of this bachelor thesis was to perform an automated download of the Dafny language server to the local machine of the user.

## 4.4 Dafny Language Features

With regard to the symbol table, Dafny had to be studied more in detail. For example, shadowing describes the existence of multiple variables with the same name, but different visibility scopes. Concepts like these are highly relevant for the construction of a symbol table.

To be aware of which such concepts are supported - or prohibited - by Dafny, we studied the *Dafny Reference Manual* [18]. This section discusses the most relevant Dafny language concepts with regard to the symbol table.

### 4.4.1 Modules

Dafny code can be organized by modules. A module is comparable to a namespace in C# or C++. Modules can also be nested. To use a class defined in another module, the user has three options. Imagine a method `addOne` defined in a module `Helpers` as shown in listing 7.

---

```
1 module Helpers {
2     function method addOne(n: nat): nat {
3         n + 1
4     }
5 }
```

---

Listing 7: Module Example

- The user states the module name explicitly in front of the method he wants to call, namely `Helpers.addOne(5)`.
- The user imports the module, for example with `import H = Helpers`. Afterwards, they may type `H.addOne(5)`.
- The user imports the module in opened state: `import opened Helpers`. Now the user is eligible to skip the namespace identifier and can just write `addOne(5)`.

Importing a module in opened state may cause naming clashes. This is allowed, but in this case, the locally defined item has always priority over the imported one. For example, in listing 8, the `assert` statement is violated, since the overwritten `addOne` has priority [19].

```
1 module Helpers {
2     function method addOne(n: nat): nat {
3         n + 1
4     }
5 }
6 function addOne(n: nat): nat {
7     n + 2
8 }
9
10 import opened Helpers
11 method m3() {
12     assert addOne(5) == 6; //violated
13 }
```

Listing 8: Naming Clash

To import a module defined in another file, the user has to import the file using the expression `include "myFile.dfy"`. This includes all content of the included file into the current file.

#### 4.4.2 Functions and Methods

Dafny has two types of methods, or functions respectively. For a programmer used to C# or C++, this concept may be confusing at first, but is very simple:

- A *method* is what a programmer from C# or C++ may be used to. A sequence of code, accepting some parameters at the beginning and returning some values at the end. It can be a class member or be in global space.
- A *function* is more like a mathematical function. It takes an input and returns a single value. The function must consist of only one expression. Furthermore, functions are not compiled into the final executable and may only be used in specification context. That is, in contracts or `assert` statements to prove logical correctness prior to compilation [19].
- A *function method* is just both at once. It contains a single expression with a single return, but is also compiled and thus also available in regular context [19].

Listing 9 shows an example containing all three cases.

---

```
1 function method minFunctionMethod(a:int, b:int) : int {
2     if a < b then a else b
3 }
4
5 function minFunction(a:int, b:int) : int {
6     if a < b then a else b
7 }
8
9 method minMethod(a:int, b:int) returns (r:int) {
10    return if a < b then a else b;
11 }
12
13 method Main() {
14    //Regular Context:
15    var a := minMethod(2,3);           //OK
16    //var b := minFunction(2,3);      //Error
17    var c := minFunctionMethod(2,3);  //OK
18
19    //Specification Context:
20    //assert minMethod(2,3) == 2;     //Error
21    assert minFunction(2,3) == 2;    //OK
22    assert minFunctionMethod(2,3) == 2; //OK
23 }
```

---

Listing 9: Function and Method Difference

Further concepts include:

- A *predicate* is a function returning a bool value.
- An *inductive* predicate is a predicate calling itself.
- A *lemma* is a mathematical fact. It can be called whenever Dafny cannot prove something on its own. By calling the lemma, the user tells Dafny a fact it can use for its proof [18]. An example can be found in listing 10.

---

```
1 lemma ProvingMultiplication(c: int, m: int)
2     ensures c*m == m + (c-1)*m
3 {}
```

---

Listing 10: Lemma



### 4.4.3 Hiding

Hiding is when a derived class redefines a member variable of the base class. Dafny supports inheritance with traits. A trait is basically an abstract class. While the trait can define a class variable, any class deriving from it is not allowed to redefine that class variable. Consider the following example. The commented code line would cause an error [18].

---

```
1 trait Base {
2     var a: int
3 }
4
5 class Sub extends Base {
6     constructor() {}
7     //var a: int          //Error
8 }
```

---

Listing 11: Hiding

This means that this issue does not have to be considered any further with regard to the symbol table.

### 4.4.4 Overloading

Overloading describes the definition of a method with an existing name, but a different signature. This is, with a different parameter list. Dafny prohibits this language concept to be able to uniquely identify each method by solely its name [18]. This means that within each module, each method name is unique.

### 4.4.5 Shadowing

Shadowing means that a class method redefines a variable that was already previously defined as a class member. This means that two variables with the same name exist. The local variable can be accessed via its name, but to access the class member, the keyword `this` needs to write in front of the variable name. One can even go further and redefine a local variable in a nested block scope. Consider the following code snippet in listing 12. It defines a class with a member variable `a`. It is initialized with value 2 in the class constructor. In method `m`, the variable `a` is printed. This will output 2, since the class variable is the only one we are aware of. Next, a variable with the same name is redefined. The class variable is now shadowed by the local variable. Printing `a` will now print the local variable. To access the class variable, the `this`-locator is required.

---

```
1 class A {
2     constructor () { a := 2; }
3     var a: int
4     method m() modifies this
5     {
6         print a;           // 2
7         var a: string := "hello";
8         print a;           // hello
9         print this.a;      // 2
10    {
11        print a;           // hello
```

```
12         var a: bool := true;
13         print a;      // true
14         print this.a; // 2
15     }
16 }
17 }
```

---

Listing 12: Complex Shadowing Example

In line 11, a nested scope is opened. Printing `a` at first will still yield the local variable. However, in the nested scope, we can redefine `a` again, shadowing the own local variable. Further calls of `a` will then return the boolean value. `this.a` will still yield 2, even inside the nested scope.

This behaviour can be summarized with the following three rules:

- If the variable was defined locally before its usage, the local definition is significant.
- If the variable was not defined locally before its usage, the parent scope is significant.
- If a class member is called via the `this` identifier, the class member is significant.

During analysis of this problem, the in listing 13 recursive algorithm was created to solve the problem of finding a symbols relevant declaration.

---

```
1 private Symbol FindDeclaration(ISymbol target, ISymbol scope)
2 {
3     foreach (ISymbol s in scope.AllSymbols)
4     {
5         if (s.Name == target.Name && s.IsDeclaration)
6         {
7             return s;
8         }
9     }
10    if (scope.Parent != null)
11    {
12        return FindDeclaration(target, scope.Parent);
13    }
14 }
```

---

Listing 13: Finding Symbol Definition

The code had to be adjusted later on, to also respect inherited symbols and to also check inside the default scope. If all declarations are known, the algorithm does already work for symbols defined after the first usage.

## 4.5 Abstract Syntax Tree

Whenever source code is compiled into an executable assembly, the first step is to *parse* the code. The parser of a compiler works with two major concepts. One of them is the abstract syntax tree (AST), the other is the symbol table. The AST is a tree that contains information about the scope of symbols. Consider the following code snippet.

```
1 while(i < 5) {  
2     i := i + 1;  
3     print i;  
4 }
```

Listing 14: AST Demo Snippet

The tree segment for this snippet would contain the while-statement as the root node. It then has two branches, one for the condition, and one for the body of the while-statement. The body itself consists of a list of statements. In the above example, there are two statements. The first one is an assignment. The assignment has again a left and a right side. The right hand side is a binary expression, with the  $+$ -operator in between two operands. The left operand is a name segment expression, and on the right hand side is a literal expression. The second statement inside the body is a simple print statement with only one item to print, the name segment expression  $i$ . While these are just three lines of code, the AST is already quite large compared to the code.

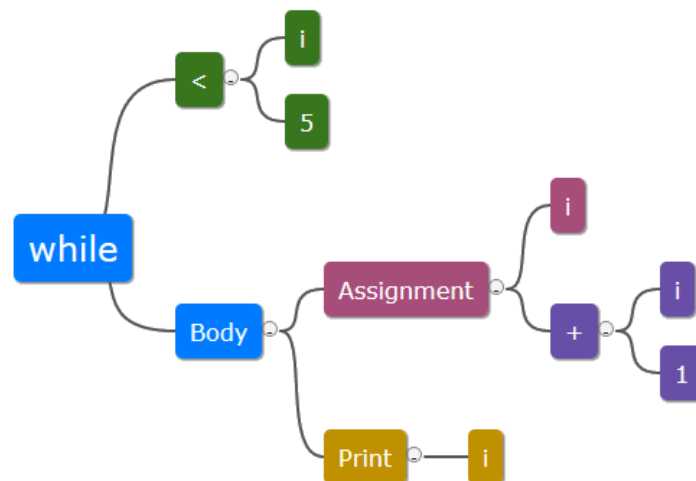


Figure 8: AST Example

The tree structure can be well seen in figure 8. The colors just represent different branches of the tree for clarity. Usually, the AST does not contain information about the type of symbols. This is where the symbol table comes into play. The symbol table contains that information and is connected to the AST, for example by the use of a dictionary, linking an AST-node to a symbol table entry. With the help of the symbol table, it can be figured that the name segment  $i$  is of type `int`. The two concepts are strongly coupled.

## 4.6 Dafny's Abstract Syntax Tree

In this chapter, we will analyze Dafny's implementation of AST-nodes. However, before we do that, we will derive what information we exactly need. Then we can compare this requirement against to what Dafny is providing.

### 4.6.1 Requirements for the Symbol Table

To be able to implement the required feature set for this project, the following questions must be answered by the symbol table or the AST, respectively.

- *Cursor position*
  - Which name segment (if any) is at the cursor's position? To have a symbol as an entry point for the *GoToDefinition* feature.
- *GoToDefinition*
  - Where is the symbol declared? To go from the received entry symbol to the declaration symbol.
- *CodeLens*
  - How often, and where, is a declaration used? To display the references to the plugin user.
- *Rename*
  - What are all occurrences of a symbol? To rename all symbols correctly.
- *AutoCompletion*
  - Within which scope was the event triggered? To suggest only symbols in the reachable scope.
  - Which declarations are available inside a certain scope? To suggest all symbols over several files, which are accessible.
  - What is the context; what was just typed before (e.g. a dot or the keyword `new`) To provide the user with more specific suggestions.

### 4.6.2 Dafny Symbols

In an optimal case, Dafny's own implementation of its symbol table and AST would already contain all of this information. Unfortunately, this was not the case. Dafny uses the Coco/R library for its compilation. This library can generate a source code parser, is thus just generating the AST [20]. As one can note regarding figure 8, the AST is not really providing the information required in the list above. For example, the question where `i` is declared is not answered. Also, some information is not relevant for the language server. Within a binary statement, the operand is not of interest for example.

Figure 9 shows all available properties and fields of a name segment. A name segment is just any occurrence of an identifier, for example of a variable or of a method.

OptTypeArguments	List<Type>
Name	string
Type	Type
IsImplicit	bool
Resolved	Expression
ResolvedExpression	Expression
SubExpressions	IEnumerable<Expression>
tok	IToken
type	Type
Accept	void
AsStringLiteral	string

Figure 9: Properties and Fields of a Name Segment

While `ResolvedExpression` looks like an interesting property, it just points to itself in a regular case, not to the declaration. Thus, if a name segment is encountered, for example as the right hand side of an assignment, the AST node provided by Dafny does not contain any information about its origin.

A better example may be on a higher level. Regarding the AST-element `method` in figure 10, it contains properties and methods for its body, but not exactly which name segments are declared inside that body. Also, there is no way to know where that method is used.

Body	BlockStmt
AllowsNontermination	bool
AssignedAssumptionVariables	ISet<IVariable>
BodyForRefinement	BlockStmt
CompileName	string
Decreases	Specification<Expression>
Ens	List<MaybeFreeExpression>
Ins	List<Formal>
IsRecursive	bool
IsTailRecursive	bool
Mod	Specification<FrameExpression>

Figure 10: Properties and Fields of a Method

To be in control of which information is stored, it was decided to implement an own symbol table / AST combination. This component will be called *symbol table* throughout this thesis and the project source code, although it will have a double linked tree structure, as we will see later. *Symbol tree* would consequently be a more accurate name.

## 4.7 Dafny Expression and Statement Types

Dafny is a very versatile language. While it offers common object oriented language features, it also contains formal language features, comparable to languages like Haskell [21]. This results in a huge variety of AST-nodes. The most important ones are discussed in this section.

Dafny works with three major base classes in its AST. These are:

- Expression
- Statement
- Declaration

Besides these, some AST-nodes are isolated, such as `AssignmentRHS`, which is the right hand side of an assignment. `LocalVariable` is another example for a separated class, that does not extend any base class one would expect. Why this isolation of certain AST-nodes was made by Dafny could not be evaluated. Both items are technically expressions and could thus be subclasses of `Expression`. Figures 11, 12, 13 and 14 follow the structure given by Dafny and present the AST-elements that were analyzed. These are also the nodes a dedicated `Accept` and `Visit`-method was implemented for. Since most of them are self-explanatory, only chosen nodes are explained.

### Expressions

Expression	Example
NameSegment	a
IdentifierExpression	a
ExprDotName	myClass.myMember;
BinaryExpression	a > 0
NegationExpression	-a
UnaryOpExpression	!a
ITEExpr	if a > 0 then a else 0;
ThisExpr	this.myMember;
ParensExpression	((a))
ChainingExpression	1 < 2 < 3 < 4 > 3 == 3 > 2;
AutoGhostIdentifierExpr	var a := 2;
LiteralExpression	2
ComprehensionExpression	forall k: int :: 0 <= k <  a  ==> 0 < a[k]
SeqSelectExpr	a[0.. a ]
SeqDisplayExpr	[1, 2, 3]
ApplySuffix	foo();

Figure 11: Dafny Expression Types

The reader notes in figure 11 that many expressions contain other subexpressions. For example, a `IdentifierExpr` is nothing more than containing a `NameSegment`.

An interesting node is `AutoGhostIdentifierExpr`. It occurs, whenever a variable declaration also contains its initialization, for example `var a := 2`. Internally, this declaration is represented as `var a : int; a := 2`. As we see now, the second `a` is a pseudo-variable, or how Dafny calls it, a `AutoGhostIdentifier`.

`ComprehensionExpression` contains a variety of subclasses, such as the `ForallExpr`, `SetComprehension` and `LambdaExpr`. All of these contain bound variables (the parameters of the lambda for example) and an expression afterwards.

`|a|` denotes the size of the set `a`.

## Statements

Statement	Example
BlockStmt	{ ... }
IfStmt	if (a) { ... }
WhileStmt	while (a) { ... }
AssertStmt	assert a == 2;
PrintStmt	print "a=", 2
ProduceStmt	yield a;
VarDeclStmt	var a : int;
UpdateStmt	a := 2;
AssignSuchThatStmt	a :  a%2 == 0;

Figure 12: Dafny Statement Types

An interesting statement in figure 12 is `AssignSuchThat`. Here, Dafny chooses what to assign, such that the condition holds. The programmer does not know exactly, which number will be assigned.

## Declarations

Declaration	Example
ModuleDecl	module M { ... }
AliasModuleDecl	import Mod = M;
ClassDecl	class C { ... }
TraitDecl	trait T { ... }
Field	var f : int;
Method	method foo() { ... }
Constructor	constructor() { ... }
Function	function even(a: int) : bool { a%2==0 }
Predicate	predicate even(a: int) { a%2==0 }
Lemma	lemma L(a:int) ensures Factorial(a) > 0 { }

Figure 13: Dafny Declaration Types

## Other

Variable	Example
NonGlobalVariable	method foo( <b>i:int</b> ) returns ( <b>o:int</b> ) { ... }
AssignmentRhs	a := 2;
ExprRhs	a := (2 > a + 1);
TypeRhs	a := new C();
LocalVariable	var a : int;

Figure 14: Dafny Various Types

These form the most important AST-nodes. Dafny provides many more of them. Due to the limited time frame, not all of them could be analyzed. A benefit of the visitor pattern is that it can just be implemented once for the base interface `IAstNode` with an empty action. This way, the visitor will just do nothing for nodes that are not supported.

## 4.8 Refactoring of the Given Dafny AST Implementation

During analysis of the Dafny AST, it was noticed that the file `DafnyAst.cs` is overwhelmingly large. It contains eleven thousand lines of code and a large number of classes. This is so extensive that even Visual Studio struggles with it and stopped working occasionally on performing refactorings.

Since this file and its contained classes will have to be extended by `Accept`-methods to implement the visitor pattern, it was considered to refactor the whole file.

Splitting the file into individual class files and dividing it into a separate package would provide much better maintainability. The following advantages are particularly evident:

- Clearer separation.
- Better overview.
- Better IDE performance.
- As a result, less error-prone coding.

However, there would also be individual disadvantages:

- Inconsistency: Other Dafny files would still be rather large. Refactoring should then be extended.
- It is not a planned enlargement of our thesis to refactor Dafny code.
- Time-consuming.
- Dafny may not want a refactoring at all, because they are used to the current situation.
- By swapping out all lines of code, the top level of the git history would be disturbed for git blame.

It was decided not to carry out a refactoring. It would be very time consuming and we would have to extend the refactoring to the whole Dafny project. Since the time frame of the bachelor thesis is limited, resources should rather be used at the own code segments and the core goals of the bachelor thesis, such as the implementation of the symbol table. However, refactoring the code of Dafny itself is one of the possible extension points for this project.



## 4.9 Prototype Functionality and Flaws to Improve

In the prototype, some features were already implemented and possible extension points were mentioned. This chapter describes the state of the prototype as it was at the beginning of this bachelor thesis, as well as the necessary improvements. It also describes how we intend to implement these improvements.

### 4.9.1 Syntax Highlighting

The syntax highlighting is realized through a given Dafny grammar file. Figure 15 shows how syntax highlighting looks inside Visual Studio Code.

As you can see, keywords like `method`, `returns`, `requires` and `ensures` are marked in purple. Types like `int` are printed in blue and comments become green. Symbols, such as classes and methods, are displayed in a brownish color. Just these simple rules increase the readability significantly.

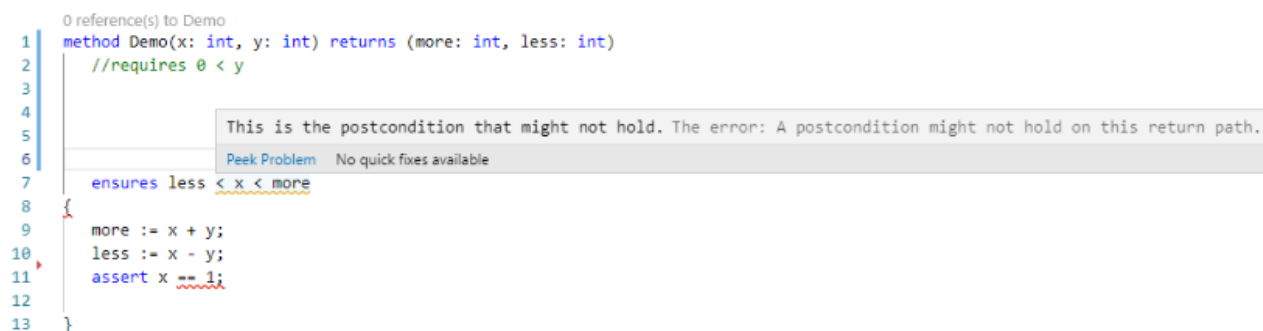
```
method Demo(x: int, y: int) returns (more: int, less: int)
{
  requires 0 < y
  ensures less < x < more
  {
    more := x + y;
    less := x - y;
    // assert x == 1;
  }
}
```

Figure 15: Syntax Highlighting

In the current public plugin an update was made to the grammar file [22]. Among other things, new keywords are now recognized and visually marked, such as `expect`. This update will be applied to the Dafny grammar file for our plugin version as well.

### 4.9.2 Verification

The verification feature underlines logical errors in the user's code. An example is shown in figure 16.



```
0 reference(s) to Demo
1 method Demo(x: int, y: int) returns (more: int, less: int)
2   //requires 0 < y
3
4
5
6
7   ensures less < x < more
8 {
9   more := x + y;
10  less := x - y;
11  assert x == 1;
12
13 }
```

Figure 16: Postcondition Violation

The logical verification is permanently active. The user just has to type Dafny code and it will be verified. The current version only supports logical errors, not syntax errors.

Whenever an assertion or postcondition violation appears, the code block is underlined in red. The actual postcondition is underlined in yellow. The user can hover over the error to get additional information about the problem.

In our thesis we want to extract the syntax errors from the Dafny diagnostic report and show them to the user as well. Dafny works with so called `ErrorReporters`, that should provide the necessary information.

In addition, it is noticeable that only the curly opening parenthesis is underlined in case of errors concerning methods. This can easily be overlooked by the user. For a more distinctive marking, we would now like to underline the complete method block from the opening to the closing parentheses. With the previous version, this implementation would have been possible only by extensive string parsing of the code - and that with every change to the Dafny code. With the planned renewal of the symbol table, we automatically have access to the information where the corresponding closing parenthesis of a code block is.

### 4.9.3 Compile

The user has two options to initiate compilation: He/She can just compile the software or he can additionally run it on a console inside VSCode. The commands are available as hotkeys, in the context menu or via the VSCode command bar. The client will save the document, send its URI, and the server will report back if compilation was successful. If any errors are present in the code, they are reported as a window notification as shown in figure 17. This includes syntax errors. If the compiled program can be run, a PowerShell instance is started directly inside Visual Studio Code.

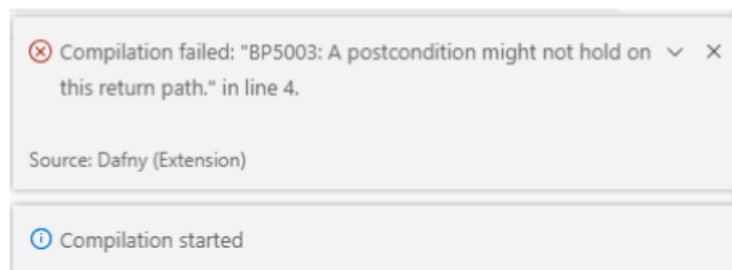


Figure 17: User Notification on Compile Errors

Dafny allows to specify own arguments for compilation, which our plugin should support, too. On the one hand, extended configuration settings should allow the user to set default values in the plugin, on the other hand the popup box integrated in VSCode can be used for individual argument input for each compile [23].

### 4.9.4 CounterExample

To show a counter example, the user has again the choice between a hotkey, the context menu or using the command bar. The counter example is then shown as a VSCode design element. If the user is no longer interested in the counter example, there is another command to hide it. The suppression of the counter model is completely handled inside the client.

The example in figure 18 demonstrates a counter example. The precondition is commented. Without this requirement, the postcondition can be violated with  $y = 0$ .

```

1 | method Demo(x: int, y: int) returns (more: int, less: int)
2 |     //requires y > 0
3 |     ensures less < x < more
4 | {
5 |     more := x + y; less = (**less#0); more = ((- 24))'1; x = ((- 24)); y = 0;
6 |     less := x - y; less = ((- 24))'2; more = ((- 24))'1; x = ((- 24)); y = 0;
7 | }
8 |
    
```

Figure 18: Counter Examples in the Prototype

In the current version the counter example is automatically hidden when the code is edited. If the user switches to another tab in VSCode, the counter example is also deactivated.

New client logic has to be used to store for which files the counter example is active. The counter example can be displayed to the user again, as soon as he switches back to the original Dafny file with the new logic. Furthermore, an automatic update of the counter example should take place whenever the user edits the code.

The representation of the counter example will also be improved. A clear and concise display and increased user friendliness is targeted.

#### 4.9.5 AutoCompletion for Identifiers

Whenever a valid interim result of a Dafny file is sent to the server, a naive file symbol table is created and buffered. As an example, if one would have a Dafny code snippet like below, the best and only match would be method `m`. This is exactly what is suggested for the user as you can see in figure 19.

```

class C {
    constructor() { }
    method m() {
        // do something
    }
}

method Main() {
    var acc := new C();
    acc.
    
```

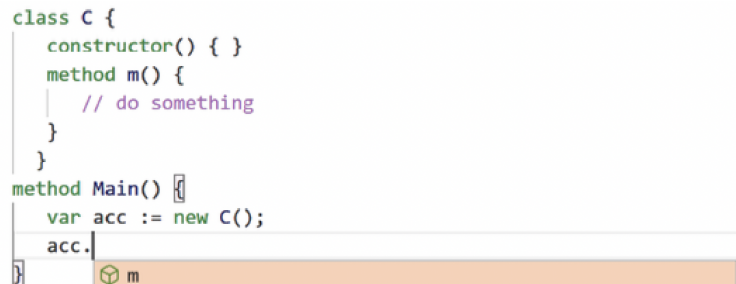


 Figure 19: *AutoCompletion* for Instance of Class `C`

Moreover, *AutoCompletion* is also able to provide general completion suggestions if one is not typing a word. In this case, just all available symbols are proposed. This is illustrated below in figure 19. Please also note that every symbol type (i.e. methods, classes and variables) have their own icons.

```
method Main() {
  var a := 1+2;
  var acc := new C();
}
```

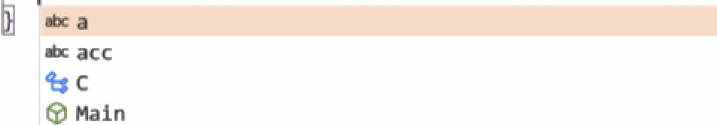


Figure 20: General Suggestions with Symbol Types

Unfortunately, both implemented versions are neither very reliable nor very performant. *AutoCompletion* was implemented via string matching. That means if *AutoCompletion* is called after a dot, the word before the dot is extracted from the source code. Afterwards, the complete symbol table is iterated through and as soon as a symbol with the matching name is found, it is assumed that this symbol is the originally declared symbol. If there are multiple symbols with the same name, or if the declaration takes place after the first usage, *AutoCompletion* would no longer work properly.

For general suggestions, simply all symbols are suggested. This includes symbols not visible in the current scope.

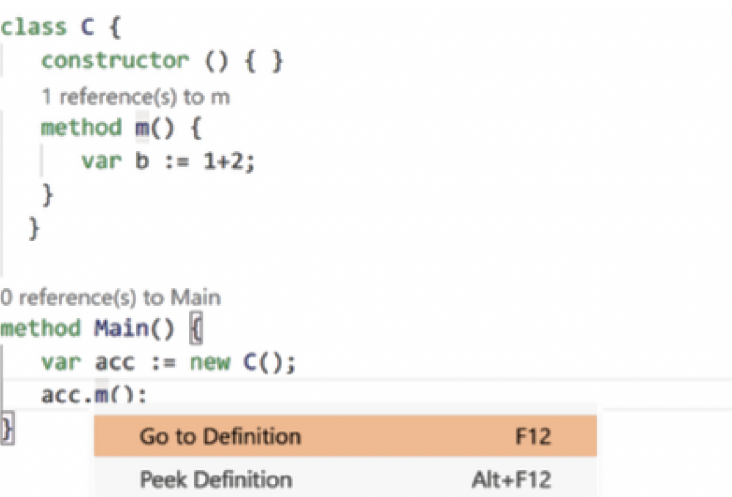
*AutoCompletion* has to be rebuilt so that it functions reliably and correct. This is only possible by the implementation of a proper symbol table.

In addition, it would be very useful for users if *AutoCompletion* is triggered after a `new` keyword and only available classes are automatically listed. This can be achieved by type filtering the symbols of the symbol table.

#### 4.9.6 GoToDefinition

With the *GoToDefinition* feature, a user can use the context menu or hit F12 to go to the definition of a symbol. In figure 21, an example is shown.

```
class C {
  constructor () { }
  1 reference(s) to m
  method m() {
    var b := 1+2;
  }
}
0 reference(s) to Main
method Main() {
  var acc := new C();
  acc.m():
}
```

Figure 21: *GoToDefinition* via Context Menu

Please note that in the current version, the cursor has to be on the left side of the symbol's keyword. Once pressed, the cursor jumps instantly to the definition of the desired symbol as it is shown in figure 22.

```
class C {  
  | constructor () { }  
  | 1 reference(s) to m  
  | method m() {  
  |   | var b := 1+2;  
  | }  
}
```

Figure 22: Cursor Position After *GoToDefinition*

*GoToDefinition* uses a similar logic to *AutoCompletion* and is therefore just as unreliable. The name of the currently selected word is read from the cursor position. Afterwards, the whole symbol table is iterated through - the first symbol with the same name is assumed to be the declaration. Then the position of this symbol is jumped to.

Similar to *AutoCompletion*, the cursor position can reliably identify the currently selected symbol and the new symbol table can determine the declaration symbol.

#### 4.9.7 CodeLens

Once a user opens a Dafny file that includes classes, methods or functions, a greyish line shows the reference count to that symbol. As an example, the class `C` has been instantiated twice and the method `m` was used once in the following code snippet in figure 23.

Unfortunately, clicking the grey references is not yet supported. This should open a popup box in which the corresponding uses are shown.

```
2 reference(s) to C  
class C {  
  | constructor () { }  
  | 1 reference(s) to m  
  | method m() {  
  |   | // do something  
  | }  
}  
  
0 reference(s) to Main  
method Main() {  
  var acc1 := new C();  
  var acc2 := new C();  
  acc2.m();  
}
```

Figure 23: *CodeLens* Example

The current implementation of *CodeLens* does not have an optimal runtime. For each symbol, the whole symbol table is iterated again to compare whether one symbol is a use of the other symbol. This results in a quadratic runtime.

With the new symbol table, each symbol knows how often and where it has been used. Therefore, we only have to iterate through the symbol table once. Furthermore, the iteration can be limited exclusively to declarations.

## 4.10 Continuous Integration (CI) of the Prototype

Continuous integration is a very important part for code quality and collaboration as you will read in chapter 9 – Project Management. Unfortunately, setting up the CI process in the prototype took almost until the end of the project and was not very successful.

This section describes the initial situation and the desired objectives regarding CI to optimize our workflow.

### 4.10.1 SonarQube

SonarQube makes static code analysis and reports bugs and vulnerabilities. On the client side, code was analyzed successfully with SonarQube by the CI process. If the code contained any TypeScript errors, an error was thrown and the build failed [3].

Integration testing and code analysis on the server side by SonarQube remained pending [3]. By splitting the client and server parts into separate git repositories, it should be easier to isolate, analyze and fix the remaining problem of a missing SonarQube report for the language server part.

### 4.10.2 Tests

The client's end-to-end tests, provided by the prototype could not be integrated, due to their heavy dependencies. This issue has to be re-visited at a given time.

On the server side, the project was automatically built by the CI server using *MSBuild*. Unit tests were executed using the *nUnit* console runner [24]. Dafny-internal tests are run using the runner *lit* [25].

### 4.10.3 Docker

The CI server bases on a Docker distribution. Docker's lightweight virtualization is ideally suited to run the CI environment.

Docker also realizes the principle "Cattle, not pets" [26]. Instead of having certain package dependencies that need to be updated continuously (pet), a "build, throw away, rebuild" procedure (cattle) is used. This way, the dependent packages will always be up to date and security patches and similar are automatically deployed.

Excluded from this principle are *Node*, *Z3*, *Go*, *Boogie* and *Sonar*. All of these have to be installed in specific release versions by the CI server. This is, since Dafny relies on specific deployments of these products. See the developer documentation for more details [27].

While working on the prototype, adjustments to the Docker image were very cumbersome. For each change, a separate commit was made to test if the pipeline would run to success. On the one hand, this was very time-consuming as the completion of each pipeline had to be awaited on the CI server, on the other hand, this resulted in an unpleasant commit history. To test changes more efficiently and allow local testing without commits, a developer-friendly way should be found and documented.

#### 4.10.4 Security Aspects

The Sonar report states four issues that the usage of launch arguments and loggers may be unsafe. However, since server and plugin run on the same machine, no active threat to a remote system exists. Thus, a special focus on application security is not necessary.

## 5 Design

This chapter contains discussions of fundamental design decisions that were made. The first section contains reasoning about the choice of technologies, such as programming languages. Afterwards, architectural discussions follow, separated by client and server.

### 5.1 Technologies

The choice of technologies is determined by external circumstances. The client is written in TypeScript, since this is required for VSCode plugins [23]. To work on the client, Visual Studio Code is a natural choice, since it contains an engine to directly launch and test the coded plugin. The server is written in C# using .NET Framework 4.6.1, since Dafny is already coded in this configuration<sup>2</sup> and direct integration is a key goal of the project. To work on the server, Visual Studio 2019 is used with the ReSharper extension. The CI will be realized with GitLab, since HSR offers it for free and the CI-pipeline of the prototype was pre-existing.

### 5.2 Client

The client consists of the VSCode plugin written in TypeScript. It establishes a connection to the server using the language server protocol.

The client is supposed to be very simple and only responsible for UI tasks, while as much logic as possible should be implemented at the server side. This allows to implement support for other IDE's with as little effort as possible.

In the following, it is discussed how this lightweight design of the client will be achieved. Furthermore, the separation into components is also explained.

#### 5.2.1 Initial Situation

The original client was already refactored within the prototype. A lot of dead code was removed and logic was moved to the server. Figure 24 gives an impression of the architecture at the beginning of this project.

---

<sup>2</sup>During the course of this project, Dafny was updated to .NET Framework 4.8



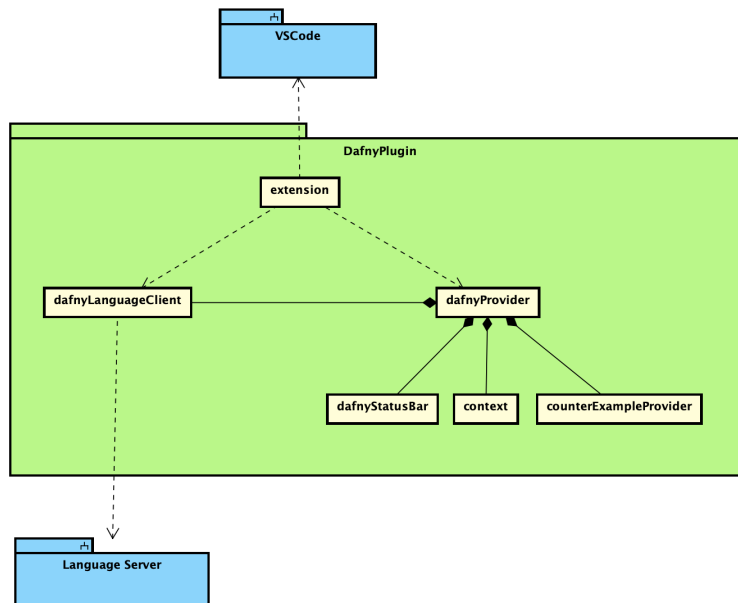


Figure 24: Client Architecture in the Prototype

In this simplified representation, the client architecture appears very tidy. However, the individual components were very large. Almost all members were public. This led to high coupling and low cohesion. Furthermore, there were various helper classes which were not grouped into sub-packages. This made it challenging to maintain the code. Moreover, it was difficult to identify all dead code passages due to the non-transparent dependencies.

As a result of those problems it was decided to redesign the client from scratch.

### 5.2.2 New Architecture

To achieve the goal of a more manageable architecture and to reduce coupling, the following measures were targeted:

- As a first step, all areas of responsibility were divided into separate components.
- The components were then grouped into packages as you can see in figure 25.

The new distribution into these packages is discussed in the following sections.

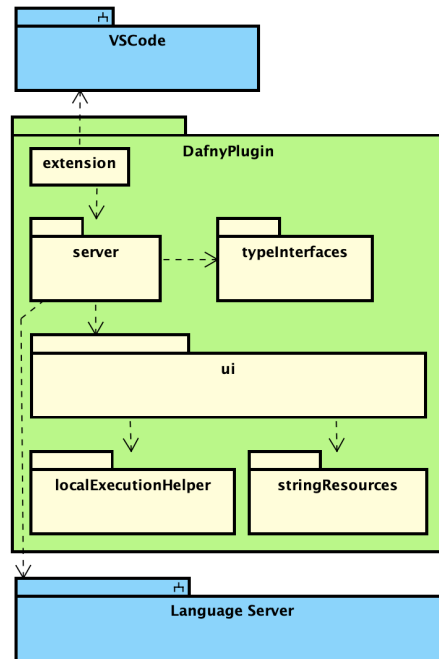


Figure 25: Client Architecture - Packages

In addition to the measures mentioned above, all logic was detached from the extension class, which forms the main component. This resulted in the root directory containing only a lightweight program entry point. The rest of the logic was split between the created packages.

As a little extra, each component contains code documentation to help other developers to get started quickly. This is also helpful because they are displayed as hover tooltips as shown in figure 26.

```

/**
 * This is the ui manager for basic instances like statusbar and a filewatcher.
 */
class DafnyUiManager
    This is the ui manager for basic instances like statusbar and a filewatcher.
export class DafnyUiManager implements IDafnyUiManager {
    
```

Figure 26: Code Documentation Example in TypeScript

**extension** – This component is the aforementioned `main`-component of the plugin and serves as an entry point. The incorporated code has been minimized. Only one server instance is started. The logic is located entirely in the server package.

**server** – The server package contains the initialization of the language server and the establishment of the connection between the client (plugin for VSCode) and the Dafny language server. In addition, all server requests, which extend the LSP by custom functionality, are sent to the server via this package.

**typeInterfaces** – In the new architecture, no `any` type should be used anymore. All types, in particular types created specifically for custom requests such as `CounterExampleResults`, were defined by interfaces.

**ui** – The UI is responsible for all visual tasks, especially VSCode commands and context menu additions. Core components like the status bar are also included in this package.

**localExecutionHelper** – This package contains small logic extensions like the execution of compiled Dafny files. The UI package accesses this package.

**stringResources** – All string resources and command identifiers are defined in this package. It is used by the `ui` package.

In the following sections, the individual components and their contents are described in more detail.

### 5.2.3 Components

Figure 27 shows a more detailed view of the client, including the components within the packages. The contents of `typeInterfaces` and `stringResources` have been omitted for clarity.

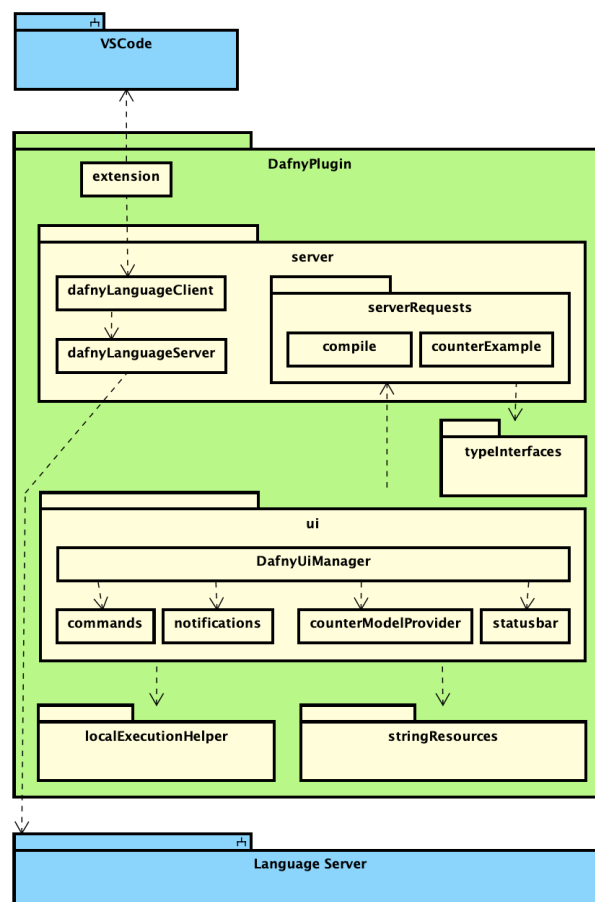


Figure 27: Client Architecture - Components

It can be seen that only `compile` and `counterExample` exist as dedicated server access classes. All other features, such as *CodeLens* or *AutoCompletion* are natively supported by the LSP. This means, that no additional client logic is necessary to support these features. Since *Compile* and *CounterExample* are custom requests, their handling has to be implemented manually within the client.

If VSCode receives a *AutoCompletion* response, the LSP standard defines how VSCode has to handle it - namely displaying the completion suggestions and insert the text which the user selects. This server-side-oriented implementation via the LSP is a great enrichment to keep the client lightweight. Adaption to other IDE's is very simple this way.

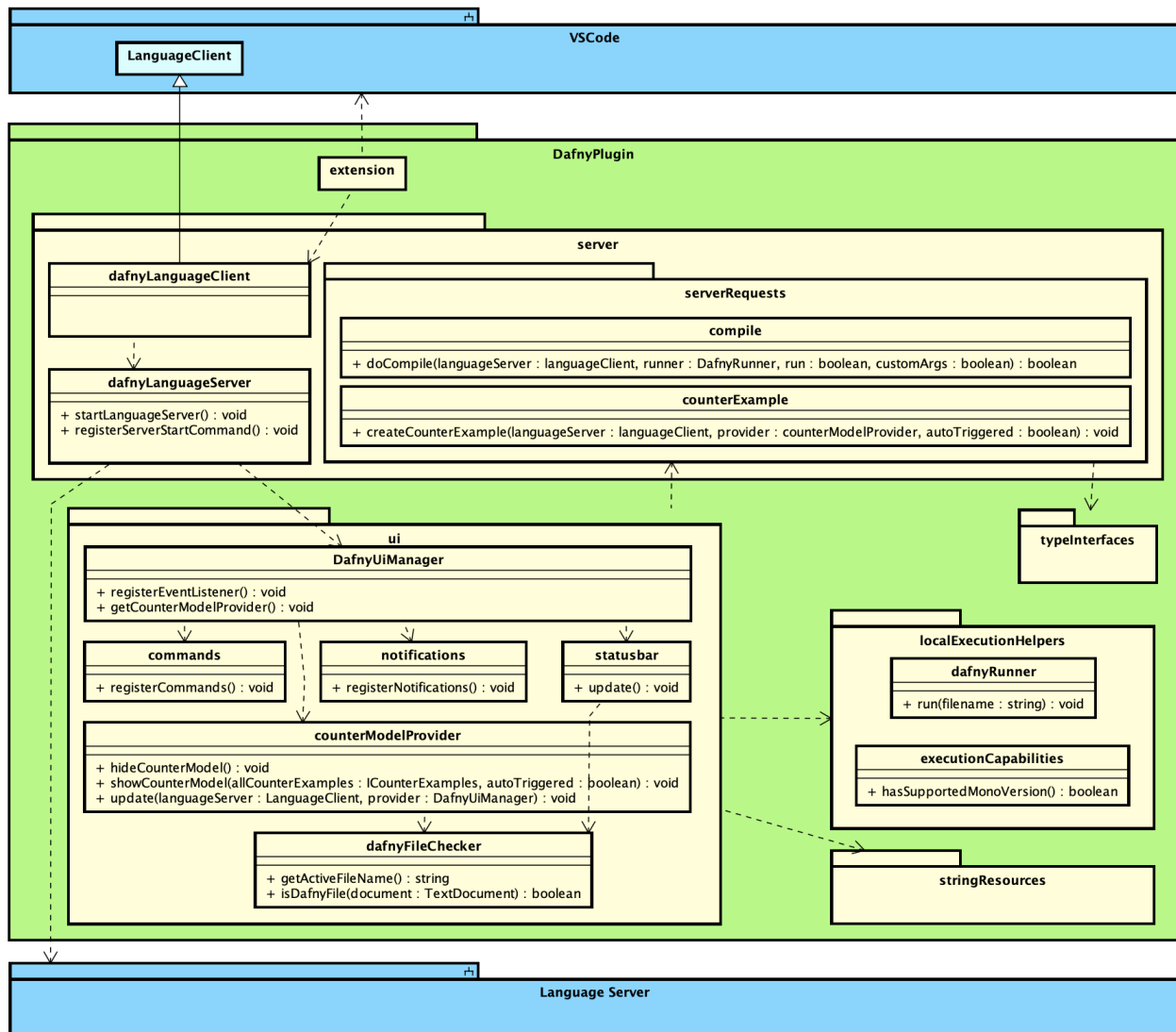


Figure 28: Client Architecture - Components and Public Methods

Figure 28 shows the public methods of each component. Only these are accessible. Instance variables were set to private visibility. Constructors were not included for simplicity. The contents of type interfaces and string resources were also omitted for clarity.

This distribution has certain upward dependencies, which is not optimal. The `ui` package accesses the server package and part of the server package accesses the `ui` package. Nevertheless, we have decided on this grouping, so that the server access functionality is encapsulated.

#### 5.2.4 Logic

The logic contained in the client has been reduced to a minimum. This has the advantage, that porting the client to other IDEs is as easy as possible. This section describes where and why the client still contains logic.

**Server Connection** – Handling of the connection to the language server and sending API requests. In addition, the client has a simple logic that certain server requests (such as updating the *CounterExample*) are sent at most twice per second.

**Execute Compiled Dafny Files** – The execution of compiled Dafny files is relatively simple. One distinguishes whether the execution of `.exe` files should be done with Mono (on macOS and Linux operating systems) or not.

**Notifications** – The client is able to receive notification messages from the server. These notifications are split into three severity levels:

- Information
- Warning
- Error

The corresponding logic in the client receives these messages and calls the VSCode API to display a popup message.

**Commands** – To enable the user to actively use features (such as *Compile*), the corresponding method calls must be linked to the VSCode UI. There are three primary links for this:

- Supplementing the context menu (right-click).
- Keyboard shortcuts.
- Entering commands via the VSCode command line.

**Status Bar** – The information content for the status bar is delivered entirely by the server. The client only takes care of the presentation. Therefore, certain event listeners must be registered, which react to the server requests. Furthermore, the received information is buffered for each Dafny file. This allows the user to switch seamlessly between Dafny files in VSCode without having the server to send the status bar information (like the number of errors in the Dafny file) each time.

**CounterExample** – Similar to the status bar, the `CounterExample` component will have a buffer. For each Dafny file in the workspace, a buffer stores whether the user wants to see the counter example or not. This way the counter example is hidden when the user switches to another file and automatically shown again when switching back to the original Dafny file.

### 5.2.5 Types in TypeScript

As already mentioned in the previous sections, `any`-types were largely supplemented by dedicated type interfaces. This prevents type changes of variables as known in pure JavaScript. Typed code is accordingly less error-prone - especially for unconscious type castings.

There are individual built-in datatypes like `number`, `boolean` and `string` [28]. For custom types, such as compilation results, we have defined separate interfaces. An example is shown in listing 15.

---

```
1 export interface ICompilerResult {
2     error: boolean;
3     message?: string;
4     executable?: boolean;
5 }
```

---

Listing 15: Type Interface Supplementing `any`-types

## 5.3 Server

This subsection documents the architectural layout of the server. The server consists of eight different packages aside the `Main` class:

- `Main`: Initializing the language server.
- `Handler`: Components handling LSP requests.
- `Core`: Contains provider-classes that supply the handlers with the necessary information.
- `CustomDTOs`: Contains all custom parameters and results like `CompilationResults`.
- `WorkspaceManager`: Represents the user workspace with its buffered files.
- `SymbolTable`: Handles anything related to the symbol table (creation, navigation, access).
- `Tools`: Individual, isolated services needed across the project.
- `Commons`: Classes accessed from multiple components, such as the language server configuration.
- `Resources`: Extracted strings.

Figure 29 shows a broad overview of the server layering. The packages are discussed in a detailed matter in the next sections from top to bottom.

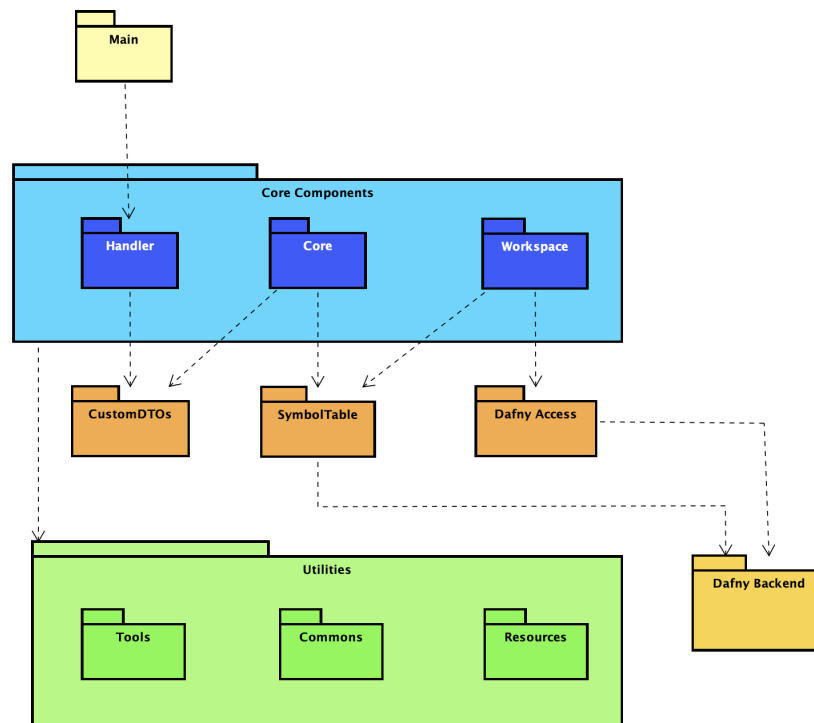


Figure 29: Server Overview

Blue UML components will refer to core components on the top layer. Orange components are related to the symbol table or Dafny access on the middle layer. Green components will refer to the utility layer.

### 5.3.1 Main Component and Handlers

The `Main` component is responsible to start the language server. The capabilities of the server can be registered by injecting a `Handler`. The handlers themselves will have to implement a specific interface by OmniSharp, for example `IRenameHandler`. These interfaces demand a `Handle` method with the proper arguments and return values to answer the request.

Figure 30 shows the basic, fundamental layout of the server architecture. Be aware that this illustration is highly simplified for a better understanding. Only one handler is shown.

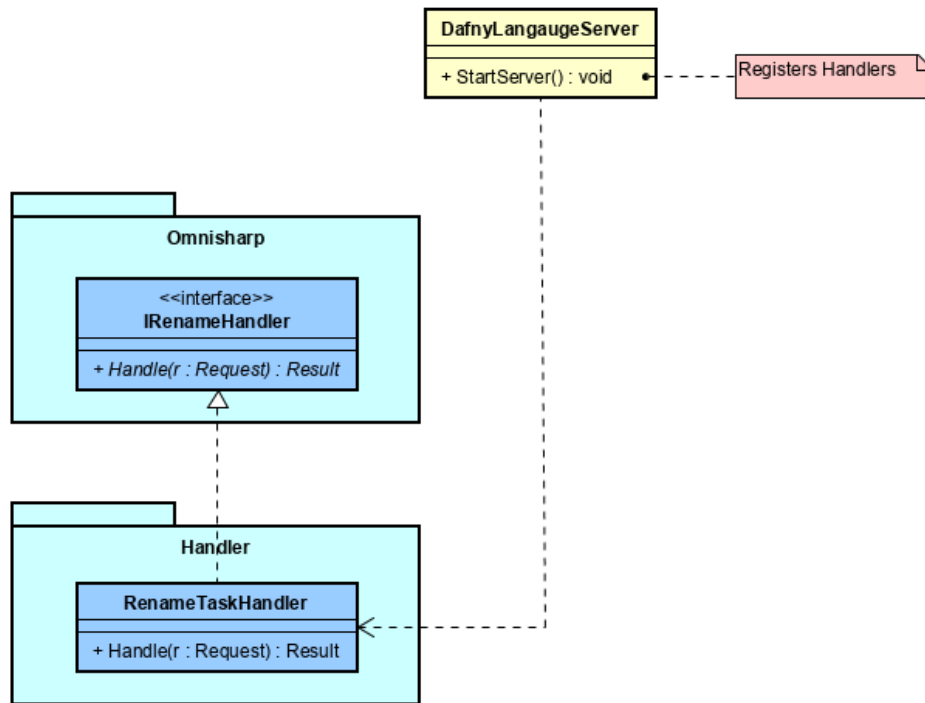


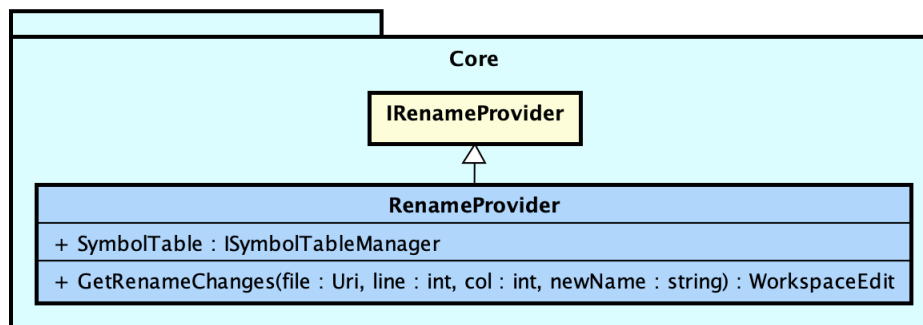
Figure 30: Basic Server Concept

The handlers contain some boilerplate code for registering the capability and keeping references to the language server. To keep classes concise, the `Handle` methods will not calculate the result themselves, but forward to call to a `Provider` from the `Core` package. This component is then responsible to provide the proper result. This way, the core logic can be easily tested since it is in a separate component.

### 5.3.2 Core

The core package contains the actual core logic for the features. There is one class for each feature, for example `RenameProvider` or `DiagnosticsProvider`. The provider is invoked by the handler and responsible to calculate the result, that has to be sent back to the client. For that, the provider will invoke the `WorkspaceManger` and the `SymbolTableManager` and operate on these. Figure 31 shows the core package with the example of the `RenameProvider`. Note that the symbol table is injected using the interface `ISymbolTableManager`. By just creating a fake instance, the providers can very easily be tested using this architecture.




 Figure 31: Core Package Excerpt with `RenameProvider`

### 5.3.3 CustomDTOs

To provide the results of custom LSP requests, result wrapper classes had to be created. Since both, `Handlers` and `Providers` require these, they were extracted to a separate package. It just contains the wrapper classes for:

- `CounterExampleResults`
- `CompilationResults`
- `CounterExampleParams`
- `CompilationParams`

### 5.3.4 Workspace Manager

A Dafny project consists out of multiple `.dfy` files. Thus, it is evident to create a class `PhysicalFile`. It just has two properties, a file path and the file content. Since the user is manipulating files by editing the code, it also provides an `Update` method.

To be able to provide all the necessary functionality, there is more information associated with a single file than just the content:

- Is the file valid?
- Does it contain errors?
- What does the compiled Dafny und Boogie program look like?
- What symbols occur in the file?

Thus, a class `FileRepository` will be created. It contains a `PhysicalFile`, but also all of the requested information in the list above. For that, a wrapper `TranslationResults` will be created, containing information about errors and compiled items. Finally, to have a link to the symbol table, a reference to a `SymbolTableManager` will be added.

Since `TranslationResults` and `PhysicalFile` are used in multiple parts of the code, these classes will be placed within a `Commons` package. Cyclic dependencies can be avoided this way.

To request information about a file, or to update a file, a component `WorkspaceManager` will be created. It contains a dictionary, linking a file identity (as URI, *Uniform Resource Identifier*) to a `FileRepository`. It offers a method to update a file, but also to get information about a file.

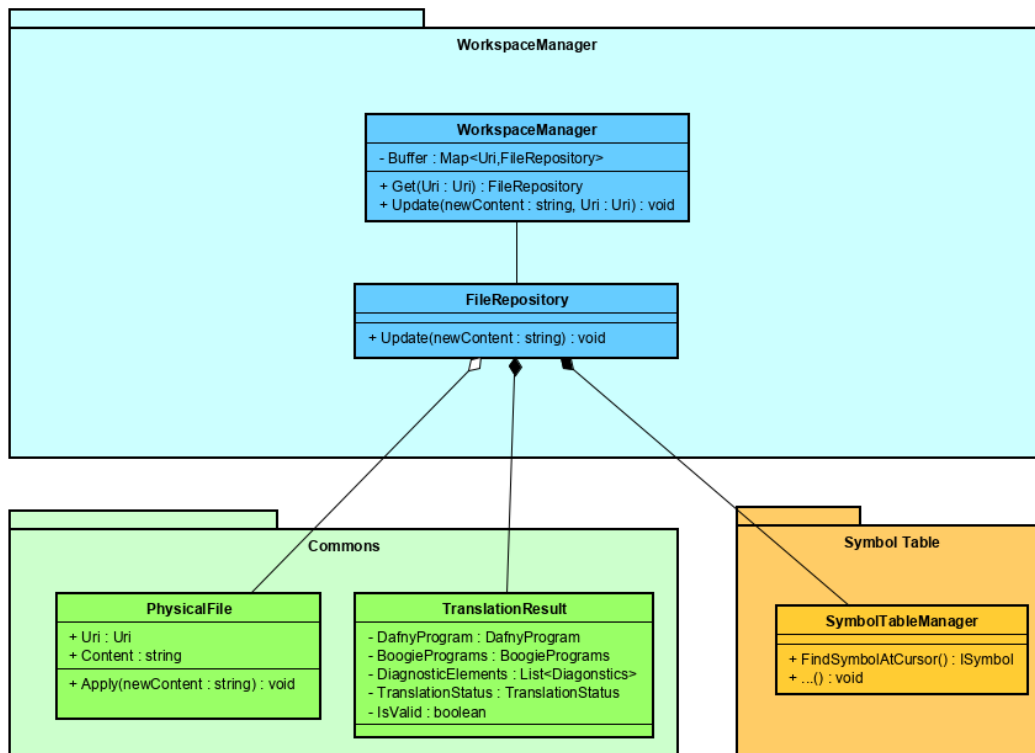


Figure 32: Workspace Component

Whenever a change is triggered, the `Update` method of the `WorkspaceManager` can be called to apply them. The manager will forward the call towards the `FileRepository`, which will adjust the `PhysicalFile`, but also generate new `TranslationResults` by invoking the `DafnyAccess` package. Last but not least, the symbol table is also recalculated invoking the proper component discussed later. Any changes are then stored in its `Buffer` property.

### 5.3.5 Dafny Access

This component is responsible to access Dafny's backend. The core class in this package is the `DafnyTranslationUnit`. It receives a `PhysicalFile` in the constructor. The only public method `Verify` will start the Dafny verification process for that file. As a result, the process will yield a list of errors, warnings and information objects, but also a precompiled `dafnyProgram` and a precompiled `boogieProgram` for later reuse. All of this information is stored in the `TranslationResults` wrapper class and returned as a result.

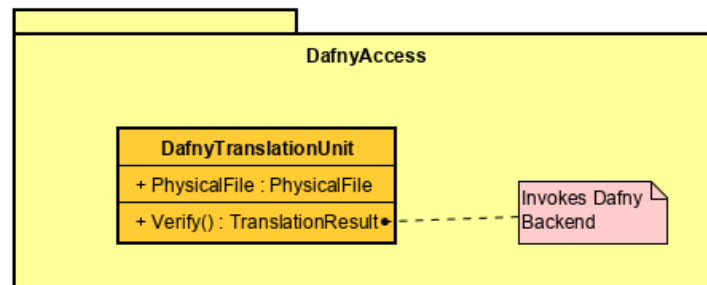


Figure 33: Dafny Translation Unit

### 5.3.6 SymbolTableManager

All tasks related to the symbol table are moved into a dedicated package. The symbol table is built based on a precompiled `dafnyProgram`. It is available from the `WorkspaceManager` for every file and can be reused by this package.

The symbol table manager consists of five classes:

- `SymbolInformation`, a class containing just raw data about a symbol.
- `SymbolUtilities`, a static class allowing operations on single symbols.
- `Navigator`, a class to navigate through the symbol table.
- `Generator`, the class that builds the symbol table.
- `Manager`, a class to provide easy access to the symbol table for outside users.

The class `SymbolInformation` is supposed to contain any necessary data about every symbol.

- In which file is it?
- On what line, on what column?
- What is its parent, what children are in the body of the symbol?

The class `Navigator` is responsible to navigate through the symbol table. Since every symbol is supposed to know about its parent and children, it is legit to speak about a symbol tree. The `Navigator` will move along the tree, for example to locate the symbol at a certain position.

The `Generator` is responsible for building the table. It will already make use of the navigator, namely to locate declarations of symbols. The result will be a root symbol, to which everything else is attached.

The `SymbolUtilities`-class allows logical operations on symbols. These were isolated for a better testability.

The `Manager` is constructed once a symbol table has been completely built. It stores the root symbol and invokes the navigator to provide easy access to the symbol table construct for the outside user.

All components are programmed against an interface, so that fakes can be created for testing.

### 5.3.7 Resources

String resources, such as error messages, are extracted to a specific resource package. This way, they are easy to maintain and adjustable at a central place. A similar system of centralization of string resources is followed as with the client.

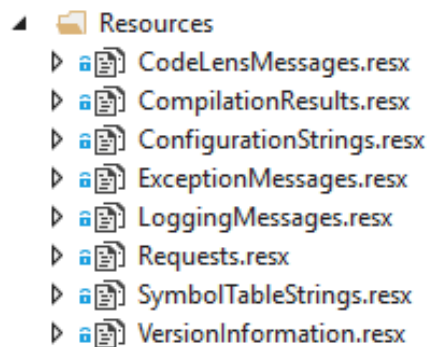


Figure 34: String Resources

As seen in figure 34, they are further split into sub-categories to keep them organized.

### 5.3.8 Tools

Classes that take care of general tasks are put into the `Tools` package. For example, one class in this package is responsible for delivering information about the folder structure, so that relative paths can be resolved. Another class is responsible for creating a proper logger, which can then be used throughout the language server. All these auxiliary classes were collected within the `Tools` package.

#### **Tools.ConfigInitializer**

The setup of the language server settings is using five different classes. To keep packages concise, a sub-package was created just for config initialization. The classes are:

- One class to parse launch arguments.
- One class to parse the config file.
- One class to report errors occurring during the process.
- One class to represent errors.
- One class coordinating the whole process and invoking all other classes.

Since the final config settings are accessed from multiple stages inside the code, the actual language server config is located inside the `Commons` package. It would be the sixth class related to the config initialization.

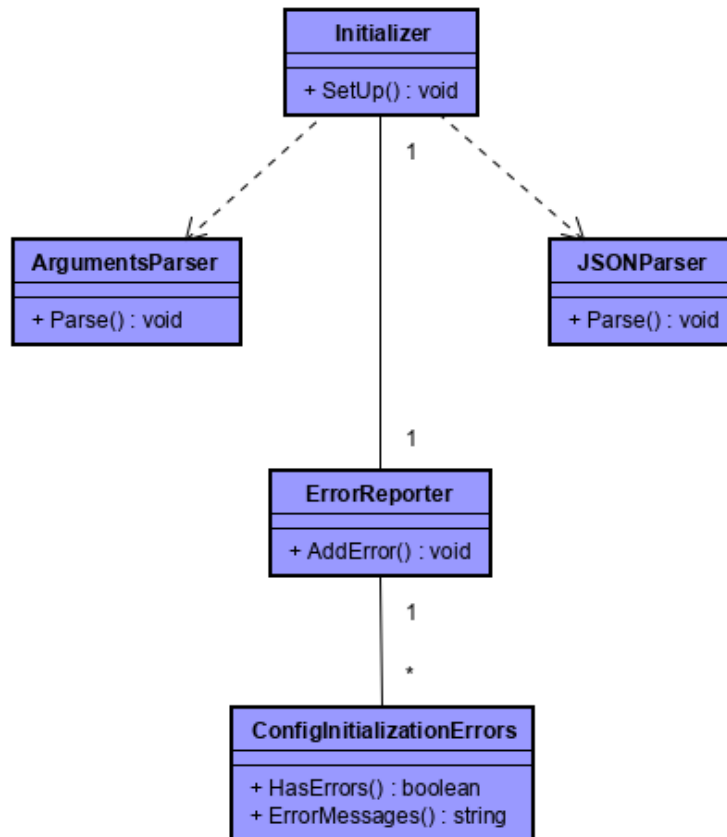


Figure 35: Configuration Initialization Layout.

The following parameters are configurable:

- Output stream file location
- Logfile location
- Minimum loglevel
- LSP transmission mode (full / incremental)

Unlike resources, config parameters affect the language server behaviour. Resources are just text representations, such as error messages.

### 5.3.9 Workflow Overview

For a regular request, the language server calls the proper handler to process the request. The handler will then retrieve the `FileRepository` from the `WorkspaceManager` and extract the necessary information. This information is forwarded to the provider, which calculates the result and returns it. This is illustrated in sequence diagram 36.

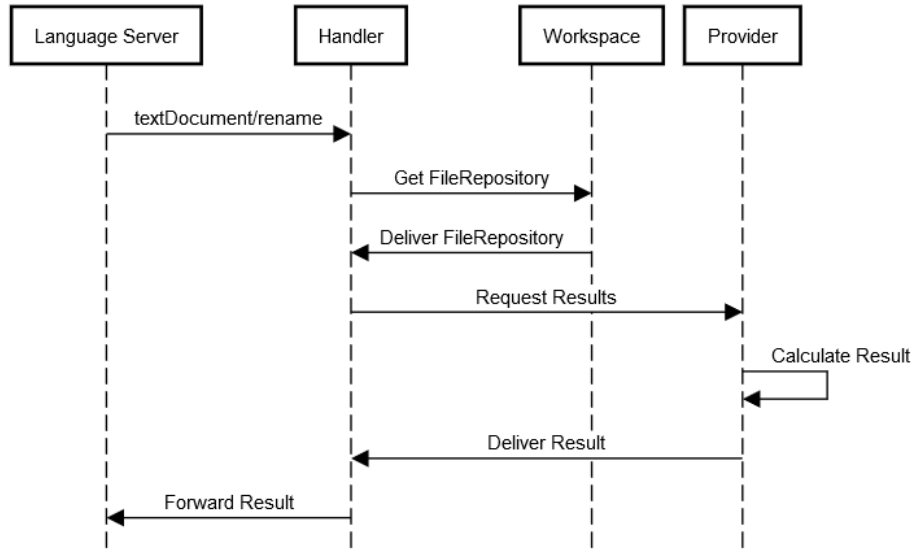


Figure 36: Sequence Diagram of a Rename Request

However, if an update is triggered, the workflow is slightly different. The handler will now actually request the `WorkspaceManager` to update a file, which will trigger the whole verification process and recalculate the symbol table, if possible. At the end, the handler retrieves the updated `FileRepository`. Now, it forwards the repository to the `VerificationProvider`, which extracts, assembles and sends diagnostics to the client. This is shown in figure 37.

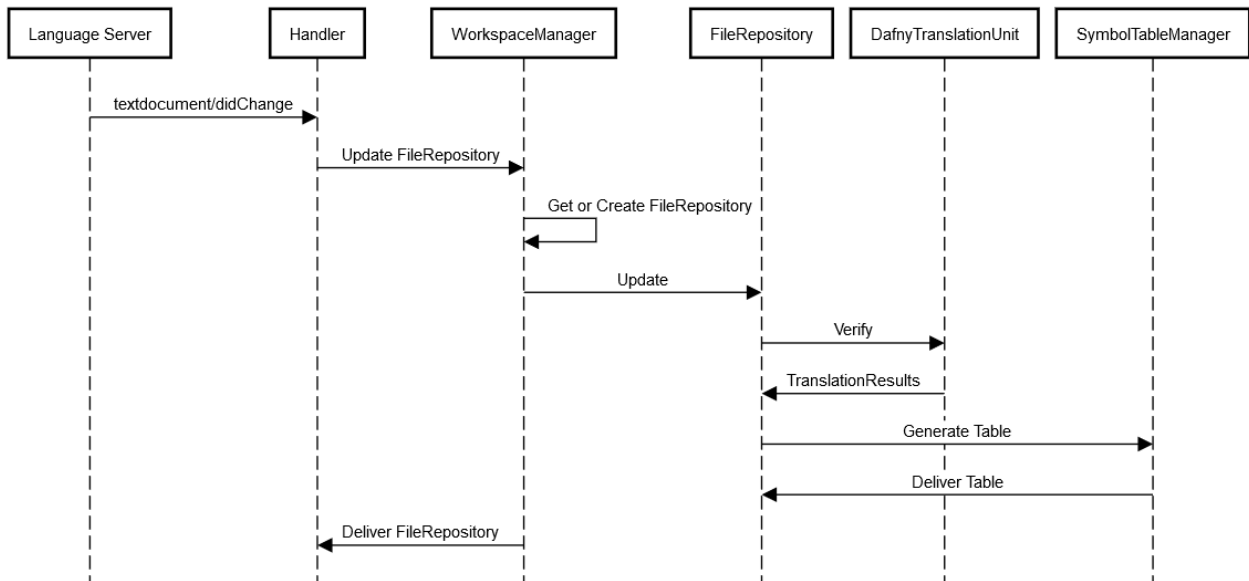


Figure 37: Updating a File After a File has Been Changed

## 5.4 Symbol Table

Since the symbol table is a core concept of this thesis, it will be explained further in this section. In the following, the design of the symbol table is described as well as it is built up with the help of the visitor pattern.

### 5.4.1 Symbol Table Design

To provide the necessary information for features like *Rename*, *GoToDefinition* or *AutoCompletion*, the symbol table needs to be very versatile and it must be able to handle the following challenges:

- Find the symbol at the cursor position.
- Go to the declaration of a symbol for the *GoToDefinition* feature.
- Find all occurrences of a symbol for the *Rename* feature.
- Get all available symbols within a scope for the *AutoCompletion* feature.
- Know about usages of a symbol for the *CodeLens* feature.

#### Find symbol at cursor

To overcome this task, each symbol knows about its position. The end of the symbol can be deduced by adding the identifier-length to the start position. A dedicated logic can then decide if the cursor is wrapped by the symbol or not.

#### Find available symbols in scope

For this task, information about scopes needs to be available. The mentioned positional data is thus extended by a `BodyStartToken` and a `BodyEndToken`. Of course, only symbols with an actual body will have these properties populated.

To find all available symbols, every scope needs to know what is declared inside it. Thus, a property `Children` is necessary. It is implemented as a hash map for instant access. One further has to divide between just declarations and all occurring symbols. Another property `Descendants` contains any occurring symbol in a scope, not only declarations.

#### Find declaration

For this task, each symbol has to hold a reference to the declaration. To be able to locate the declaration beforehand, it must be possible to move to the parent scope and scan it for declarations. Aside from the already mentioned `Children`-property, a `Parent` property is necessary.

#### Find usages of symbol

A list of usages needs to be provided, containing a reference to every usage of the symbol.

All of this leads to the following structure of `SymbolInformation`, shown in figure 38.

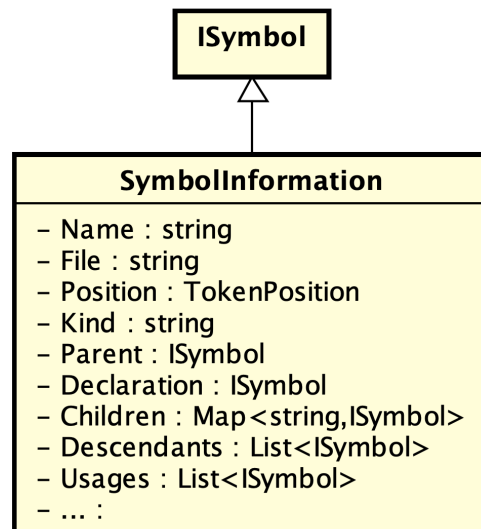


Figure 38: Symbol Table Design

### 5.4.2 Visitor Pattern

To generate a symbol table, it is common to use the visitor programming language pattern by Gamma et al [29]. The pattern is used to navigate through, mostly tree-based, data structures and execute operations while doing so. The goal of the pattern is to separate the navigation through the data structure, and the operations that take place when visiting.

Consider any tree based data structure. Every node in the tree is supposed to offer an `Accept(Visitor v)` method as shown in listing 16. This method will accept the visitor, this is, it will execute the visitor's operation on the node itself. Further, it will also call the `Accept`-methods of its child nodes. Thus, a typical implementation of an acceptor would look like this:

```
1 public void Accept(Visitor v) {
2     v.Visit(this);
3     foreach (Vertex child in this.Children) {
4         child.Accept(v);
5     }
6 }
```

Listing 16: Standard-Implementation of Accept

Note that the navigational aspect - the `foreach` loop - is inside the `accept` method, but nothing is told about the visit operation. The visitor can do whatever it wants with the node, for example print it to the console. To work with every node that may occur, the visitor must overload the `Visit(Node n)` method for each possible subclass of `Node`. Within a tree, this usually just concerns nodes and leaves. For a symbol table, possible node types are any kind of expressions and statements.

A visitor implementation could look like shown in listing 17.



---

```
1 public class Printer : Visitor {
2     public override void Visit(Node n) {
3         Console.WriteLine("Node: " + n.ToString());
4     }
5     public override void Visit(Leaf n) {
6         Console.WriteLine("Leaf: " + n.ToString());
7     }
8 }
```

---

Listing 17: Example for a Visitor

### 5.4.3 Global Symbol Table vs Symbol Table per File

The current design creates a symbol table per file, and attaches it to the file repository. This variant was chosen for simplicity. However, it is technically not correct. A better option would be to use a global symbol for the entire workspace. This has two reasons:

- If a file is included elsewhere - the file is not aware about this. *CodeLens* and *Rename* will not work properly.
- *AutoCompletion* may only suggest symbols in the current file. Symbols available in other files are not known.

A global symbol table is a possible extension point in the future. It would also increase performance and especially resource usage, since if a library is included by multiple files, its symbol table does not have to be built multiple times. If a file gets updated, only this specific part of the symbol table has to be recalculated. Symbols of other files just have to be re-attached properly. However, this task is not as easy, since one may run into problems when multiple default namespaces exist in files that do not include each other.

An alternative solution to make features like *Rename* and *GoToDefinition* know about where they are included is to keep a table with this information. If a file *A* is included in file *B*, an additional buffer within file *A* indicates that it was included by file *B*. Keeping all these references up to date is also connected with logical effort. Therefore, it is advisable to implement the global and updateable symbol table instead.

## 6 Implementation

This chapter describes the realization of the planned design. First, the characteristics of the client are shown. Then, basic components of the server are described. The symbol table and its properties are treated as a separate section. Then, interesting aspects about the implementation of the features themselves are presented. This is followed by a discussion about platform independence and a section about automated testing. The results of the usability test and the resulting conclusions and improvements are also contained in this chapter. It closes by the achievements in terms of continuous integration.

### 6.1 Client

The implementation of the client could basically be done as intended. The whole code could be broken down into the individual components defined in the design phase. Individual deviations and descriptions of the individual implementation concepts follow in this section.

These include a better separation of the components, a simplified migration concept for other, future IDEs, the download of the language server as well as basic configurations which are made possible for the end user.

#### 6.1.1 Better Encapsulation Through Interfaces and Modules

During the implementation process, many confusing module imports and component usages arose. Their purpose was generally unclear and thus difficult to understand. The imports caused a lot of dependencies, too. This problem had to be solved by further introductions of interfaces and encapsulation of modules.

Although interfaces were used for individual types, core components did not use their own interfaces. To reduce coupling, isolated modules were formed in a comprehensive refactoring process. The modules now no longer program on the class implementations, but against the interface.

For this purpose, one importable module with the name `_<Directory>Modules` was created for each directory. Figure 39 shows an overview of the interfaces. In addition, the dependencies among each other are shown. For simplicity, the contents of `stringResources` and `typeInterfaces` to each package have been omitted. The dependency of `extensions` on `VSCode` is based on dependency injection and represents the plugin entry point. Apart from this dependency, no dependency injection dependencies are shown in the diagram.

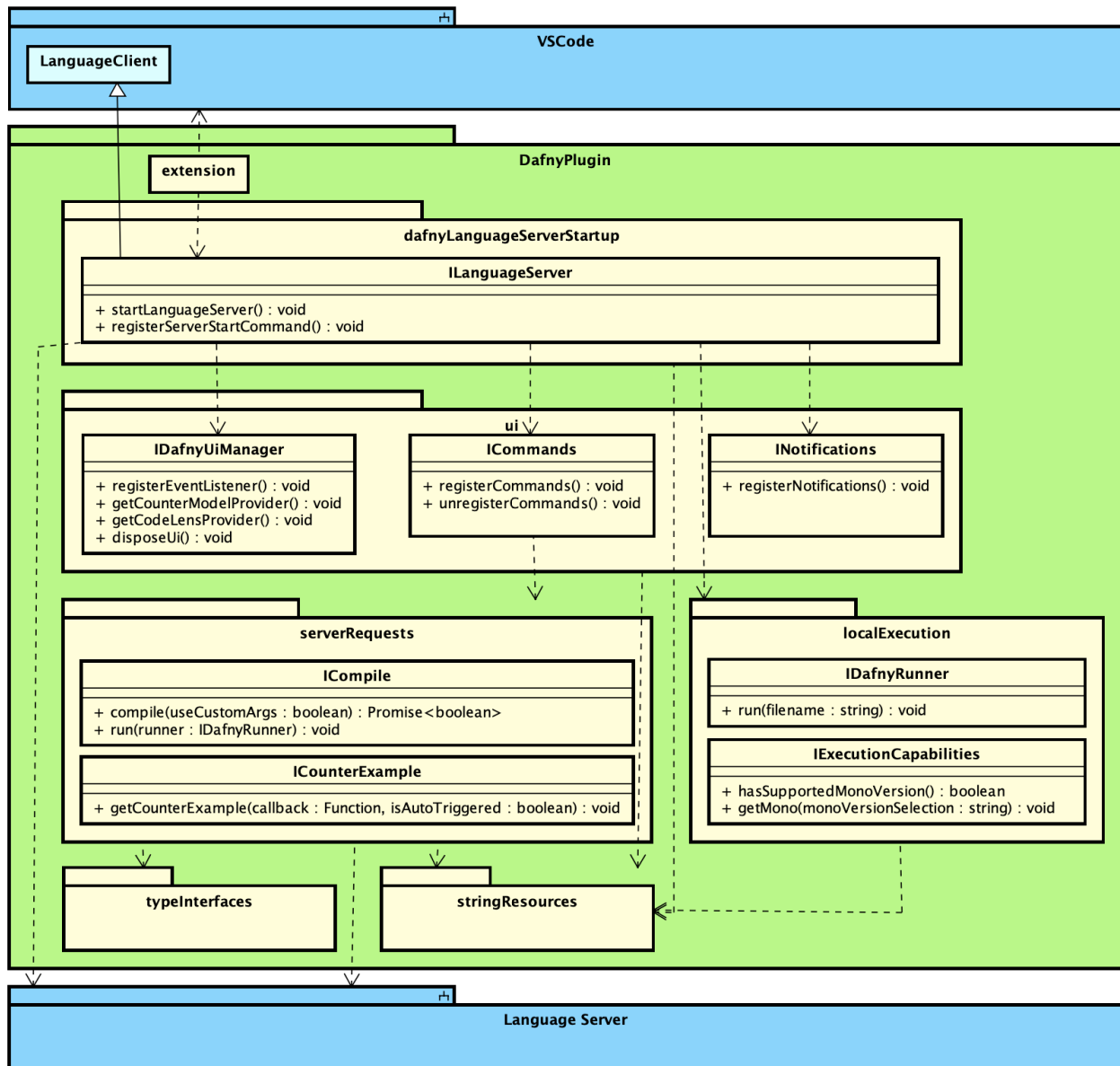


Figure 39: Second Major Client Refactoring

At first glance, the architecture appears much tidier. The dependencies are now pointing from top to bottom. Methods have been simplified and the number of parameters could be reduced significantly. Component identifiers have been renamed to be more understandable.

However, it is now also noticeable that there are considerably more dependencies on `stringResources`. While in the previous version only the module `ui` used `stringResources`, it is now used by almost all other modules. This has the following reason: Up until this refactoring, the task of `stringResources` was to be a central collection of all user interface (UI) strings. In the code review, it was decided that default values should no longer be set within the independent modules, but rather at a central location. This would make it easier to maintain these values.

### 6.1.2 Client Modules in Detail

The individual modules from the design concept are shown below. The components shown in green are exported to the outside of the module and can be used by other modules. The other, orange components are only used within the module.

For simplicity, constructors, private methods and private variables are omitted in the diagrams. This is, since implementations of interfaces only implement the methods specified as public.

In addition, there are two core modules, which were not considered in the design concept. The need for these modules only became apparent during the implementation phase. This concerns the isolation of VS-Code components and the download of the Dafny language server. These two modules will be discussed separately in the following sections.

#### **dafnyLanguageServerStartup**

The main module `dafnyLanguageServerStartup` is used by the `extension.ts` and contains basically the plugin's "main logic". It is to start the Dafny language server and establish a connection, as well as basically initialize the plugin. This includes creating the UI management component and command registrations. If no local language server is available, the language server is downloaded from a server using the `LanguageServerInstaller` component. This component is only used inside the package, and is not exported to the outside, as it can be seen in the figure 40. This component can also be used to update the language server locally. Updating means to delete the local existing language server and download the latest version. For more details see section 6.1.4.

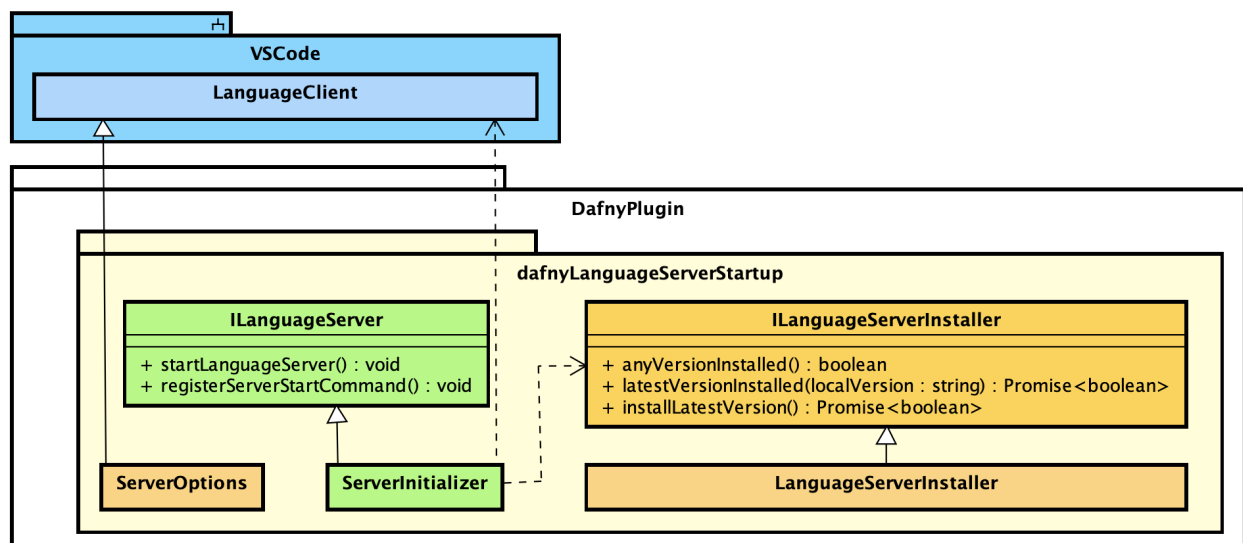


Figure 40: Module `dafnyLanguageServerStartup`

The logic of the main component `serverInitializer` itself was kept relatively small. To start the language server as a local process, two steps are required:

First, basic options are set for the language server. These options are set in the `serverOptions` component. This component implements the `LanguageClient` of VSCode. "Client" is somewhat confusing in that sense, since this module is used to connect the language server; from client side. Among other things, the options are set with which command the server is started and which file types, which are opened in VSCode by the user of the plugin, should be transferred to the server.

Second, `serverInitializer` registers a callback as the program entry point to the created `serverOptions` instance and then the language server is started with `this.languageServer.start()`. To run the Dafny language server, an instance of `DafnyRunner` is created. This component is part of the `localExecution` package.

## ui

This package is responsible for all visual representations. This includes the status bar information and the display of the `CounterModel`. It also registers VSCode commands and context menu additions.

`DafnyUiManager` is the basic UI entry point and manages the UI components. It creates the individual components and updates them specific events if necessary. This is the case, if the opened document is changed by the plugin user.

A `disposeUi` function is also provided, which destroys the UI elements. This is the case on a Dafny language server restart. On restart, all UI components are rebuilt, so that a clean plugin restart is performed if one of the features should be frozen and does not response anymore in an extreme case.

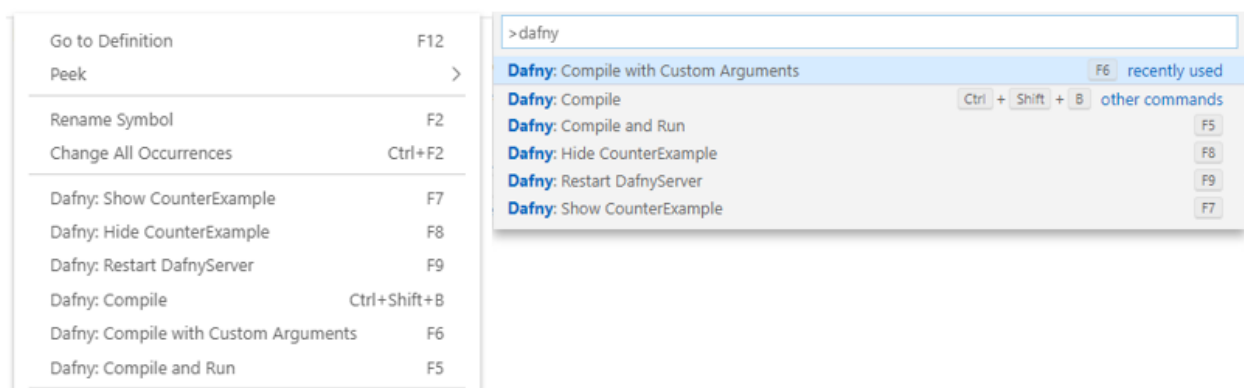


Figure 41: Commands in the Context Menu and the VSCode Command Line

The `Commands` package registers all commands that are offered to the user and links them to corresponding callback functions. These commands are available within the context menu and the available commands in the VSCode Command line, as shown in figure 41. Individual dependencies are injected. These includes: `IDafnyRunner` to call compile and run as a callback as you can see in listing 18. Furthermore, `IDafnyUiManager` is injected in order to access the `CounterModelProvider` with the appropriate commands and to show or hide it.

```
1 [..]
2 name: CommandStrings.CompileAndRun,
3 callback: () => {
4   const compile: ICompile = new Compile(this.languageServer);
5   compile.compile(false).then(() => compile.run(this.runner));
6 }
7 [..]
```

Listing 18: Excerpt from `commands.ts`

The following commands are registered:

- **Compile** – to compile the Dafny program.
- **Compile with Custom Arguments** – to pass a compilation request with user-specific compile parameters to the server.
- **Compile and Run** – after compiling the Dafny program is executed and the output is displayed.
- **Show CounterExample** – displays the counter example for the currently opened Dafny file, if available.
- **Hide CounterExample** – hides counter example for the current Dafny file.
- **Show References** – with a click on a *CodeLens* reference, the reference is shown to the user via a popup window.

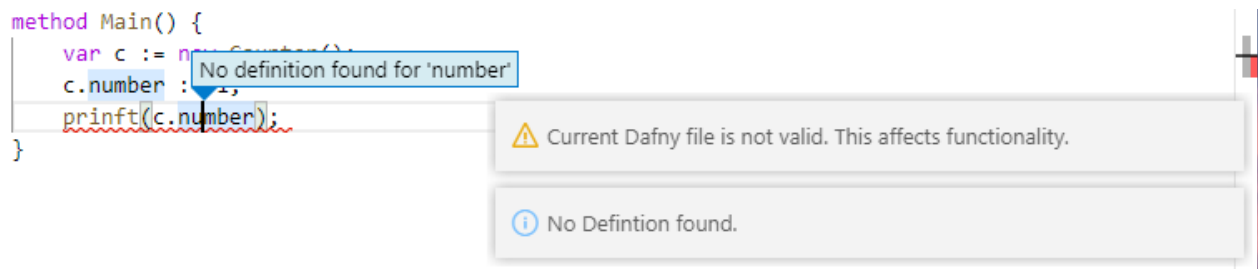
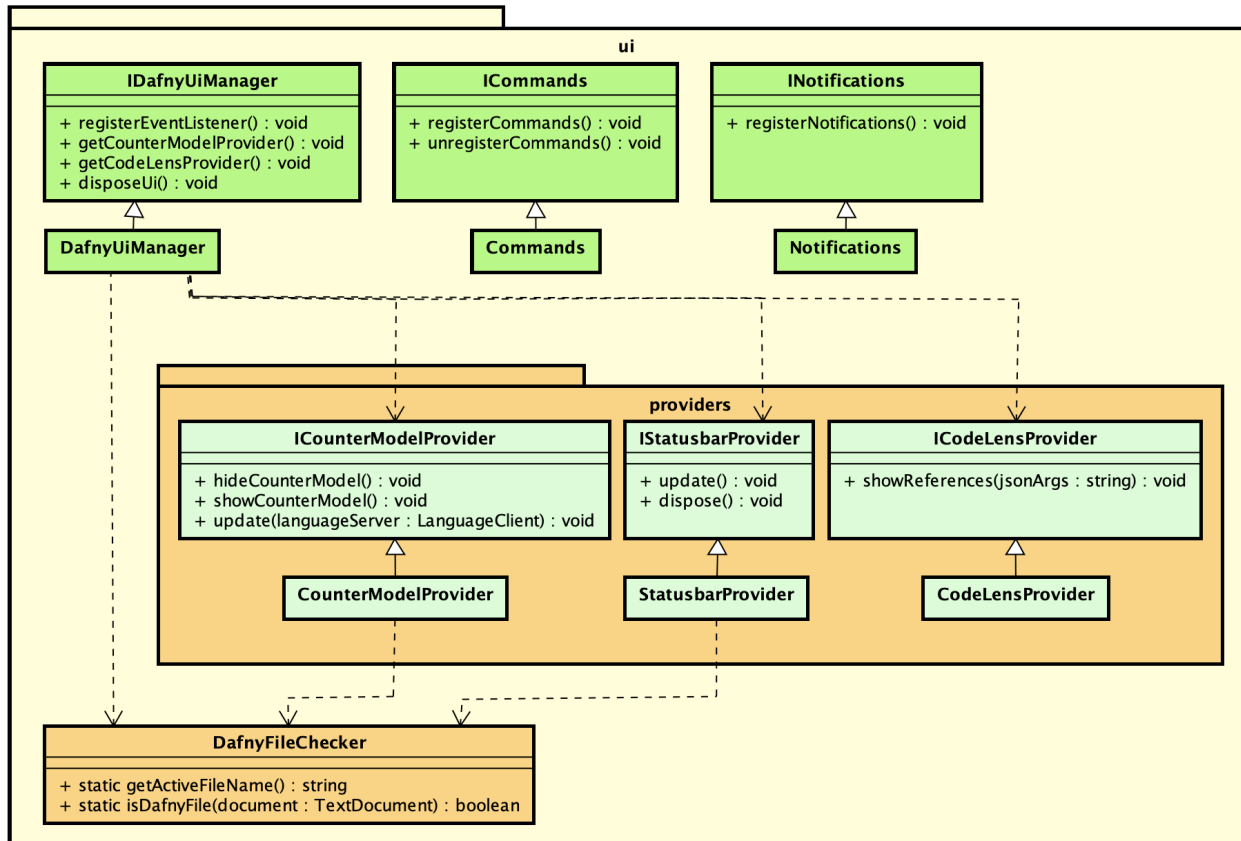


Figure 42: Warning Message Received From Server

The `Notifications` component is responsible for displaying messages received from the server. An example is shown in figure 41. There are three message types, each of which is displayed in a different color:

- **Error** – is shown in red.
- **Warning** – is displayed in yellow.
- **Info** – is shown in blue.

The logic to display the messages is directly provided by the VSCode API.


 Figure 43: Module `ui`

The `providers` module provides the individual UI components. The packages are exported to be used by the `DafnyUiManager` component. However, the `providers` module is not exported to the outside by the `ui` module, so it is shown in orange in figure 43. The module was created purely for grouping and a better file structure.

The `CodeLensProvider` component is very simple. *CodeLens* data transferred from the server is transferred to the VSCode API. The logic is limited to passing the data through. The actual display of the *CodeLens* popup window, as shown in figure 44, is done via the VSCode API. More about the logic of *CodeLens* can be found in section 6.4.2.



Figure 44: CodeLens Opened After Clicking on References

The `CounterModelProvider` component is used to display and hide counter examples. This component also stores in which Dafny files a counter example should be displayed and in which not. Therefore, if the user changes the opened file in the workspace, the counter example is automatically hidden or shown. The module supports a dark and light mode as shown in figure 45. This mode is automatically set accordingly, depending on whether VSCode is set to light or dark by the user.

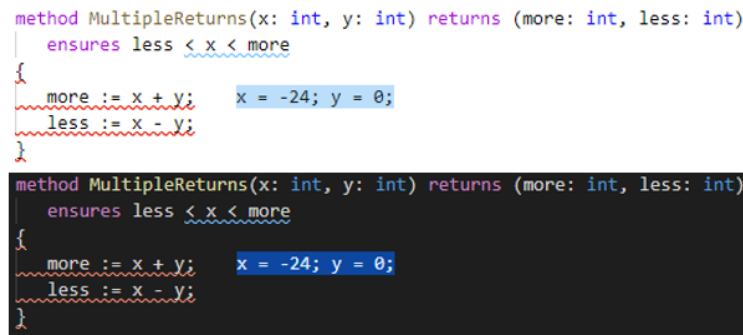


Figure 45: CounterModel in Light (Top) and in Dark Mode (Bottom)

Unlike the `Compile` feature, the `CounterModelProvider` creates a callback and passes it to the `serverRequest` module. Therefore an update-call of the `CounterModelProvider` requires a transfer of the language server instance to instantiate a `RequestCounterExample`. This is for the following reason: Unlike the `StatusbarProvider` or the `CodeLensProvider`, `CounterModelProvider` has an active influence on when exactly a request is sent. On the other hand, the components for the status bar and `CodeLens` hand react passively to an event.

This way the `CounterModel` component can regulate the server requests. For reasons of efficiency, the implemented client-side display of counter examples limits its server requests to a certain number per second. This means that even if a plugin user types very fast on the keyboard, a update requested by `CounterModelProvider` to the server is limited by default to two per second.

The `StatusbarProvider` reacts to messages from the Dafny language server. It primarily displays the following information to the user as shown in figure 46: Whether the Dafny file is valid, how many errors are present and the Dafny language server version.



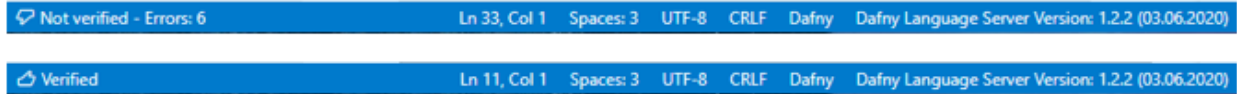
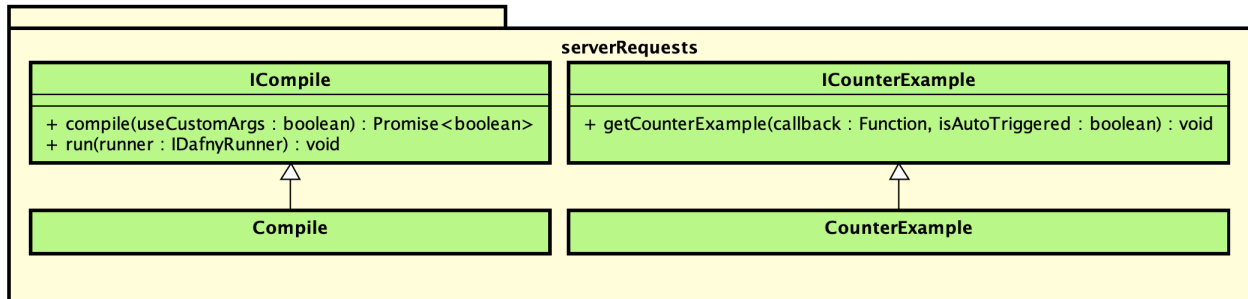


Figure 46: Dafny Status Bar for an Invalid (Top) and a Valid File (Bottom)

### serverRequests

This package is responsible for server requests which are not automatically handled by VSCode. Specifically, this concerns the LSP extensions for the compilation of Dafny programs and showing counter examples as seen in figure 47.

Both components receive an instance of `LanguageClient` injected to gain access to the server. As explained for the component `RequestCounterExample`, the `CounterExample` feature limits the server requests. Therefore only one instance of this component is generated and injected callbacks are used. `Compilation` is executed each time the user explicitly requests a compile. A new instance of `Compile` is created each time. Therefore, an asynchronous method is used for this component.

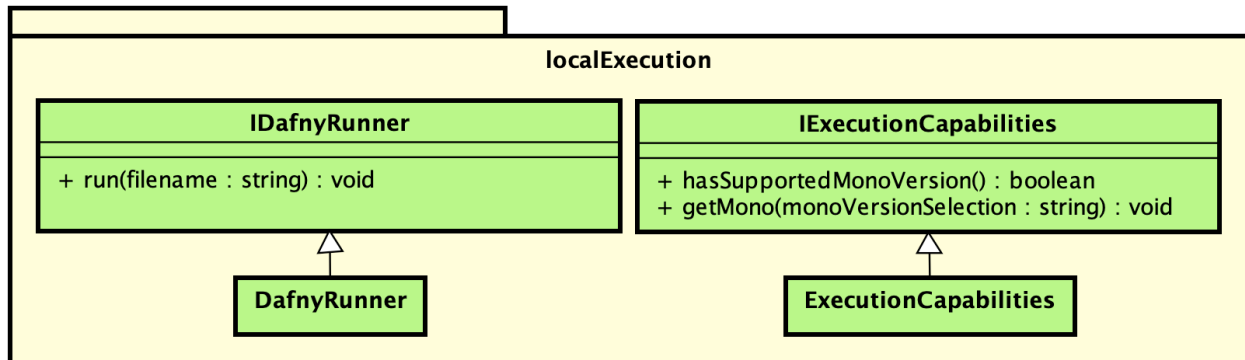

 Figure 47: Module `serverRequests`

### localExecution

This package is used for the local execution of programs. For example to execute `.exe` files. For macOS and Linux these are executed using Mono. This module takes care of the correct execution of compiled Dafny programs and of the Dafny language server, which is also an `.exe` file.

`ExecutionCapabilities` check for supported capabilities. Under Windows, `.exe` files can be executed natively. Under macOS and Linux, the Mono version is checked and if necessary the user is asked to install Mono.

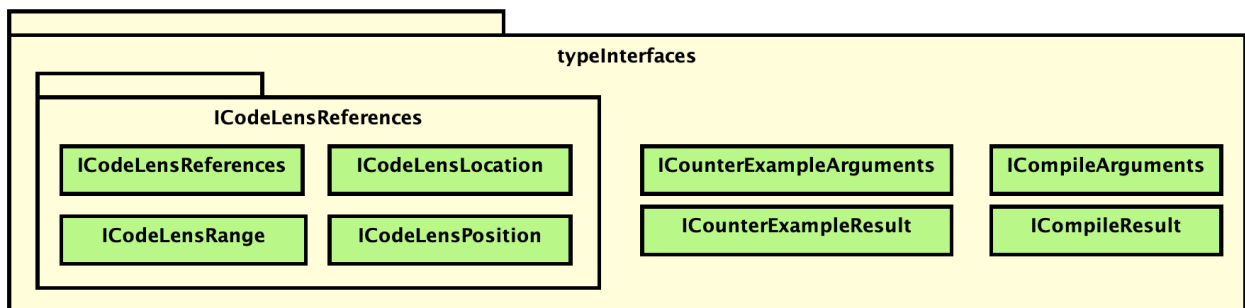
The component `DafnyRunner` executes compiled Dafny programs. It creates a corresponding terminal process and launches the executable. This component is only used by `Compile`. Of course, programming is done against the respective interface, as shown in figure 48.


 Figure 48: Module `localExecution`

### **typeInterfaces**

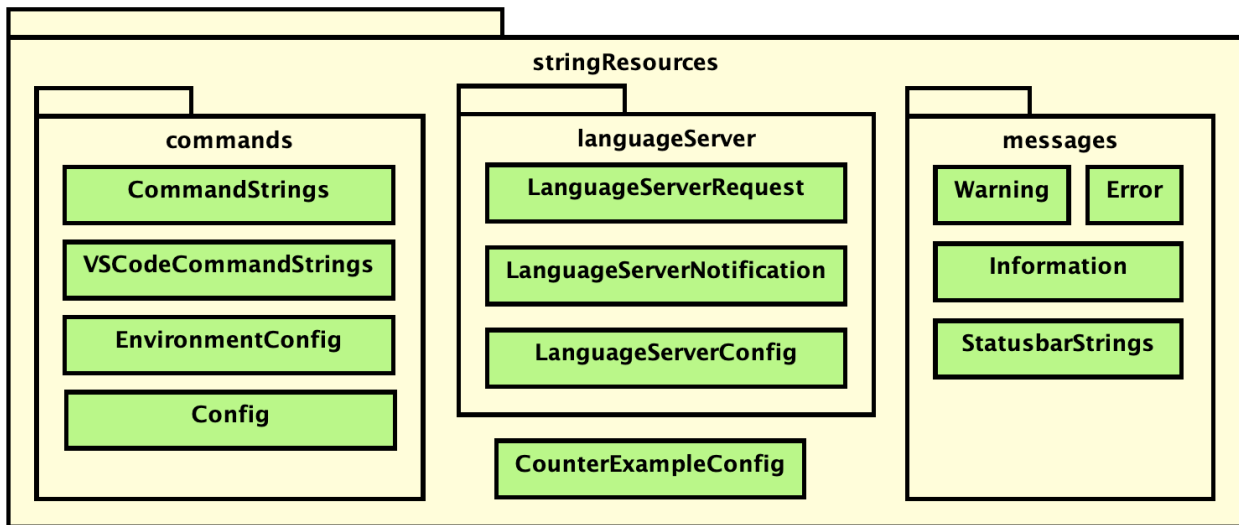
Special TypeScript type extensions, as they are used for the return values of server requests that are not contained in the LSP, are defined in interfaces. This concerns the return types of *Compile* and *CounterExample* as shown in figure 49.

In principle, there is a type interface for the arguments and for the results for each own implemented LSP feature. Only for *CodeLens*, the types are somewhat more complex, which is why a different grouping was created for this interface. `ICodeLensReferences` is a file, which contains several interfaces and no own module.


 Figure 49: Module `typeInterfaces`

### **stringResources**

All string resources are stored in this module. They are logically grouped by usage area, as shown in figure 50. Contrary to the design, this module does not only contain UI messages displayed to the user as mentioned in section 6.1.1. Instead, it is responsible for the central outsourcing of all strings that also have to do with configurations. This is, for example, which file extension a Dafny file has.


 Figure 50: Module `stringResources`

The structure consists of exported classes with static members. For simplicity, these members are not listed in the figure. In listing 19, an example is shown.

---

```

1 export class LanguageServerNotification {
2   public static Error: string = "ERROR";
3   public static Warning: string = "WARNING";
4   public static Info: string = "INFO";
5   [...]

```

---

 Listing 19: Excerpt from `LanguageServerNotification`

### 6.1.3 Encapsulation of the VSCode Components

During our implementation of the client we noticed that almost all our modules require different modules of VSCode. For example, the UI components needed different graphical elements while console processes are used to execute Dafny files. So the whole plugin had dependencies on the VSCode specific module kit, which could not be offered by other IDE's in the same way.

Because we wanted to design the client in a way that the migration to other IDE's is as easy as possible, we wrote an additional module `ideApi` as shown in figure 51. From this module the modules used from VSCode are exported with neutral names. All modules except `typeInterfaces` and `stringResources` use this module directly or indirectly. The idea here is that when migrating to another IDE, adapter components can be written in `ideApi` for the VSCode modules used in a centralized way.

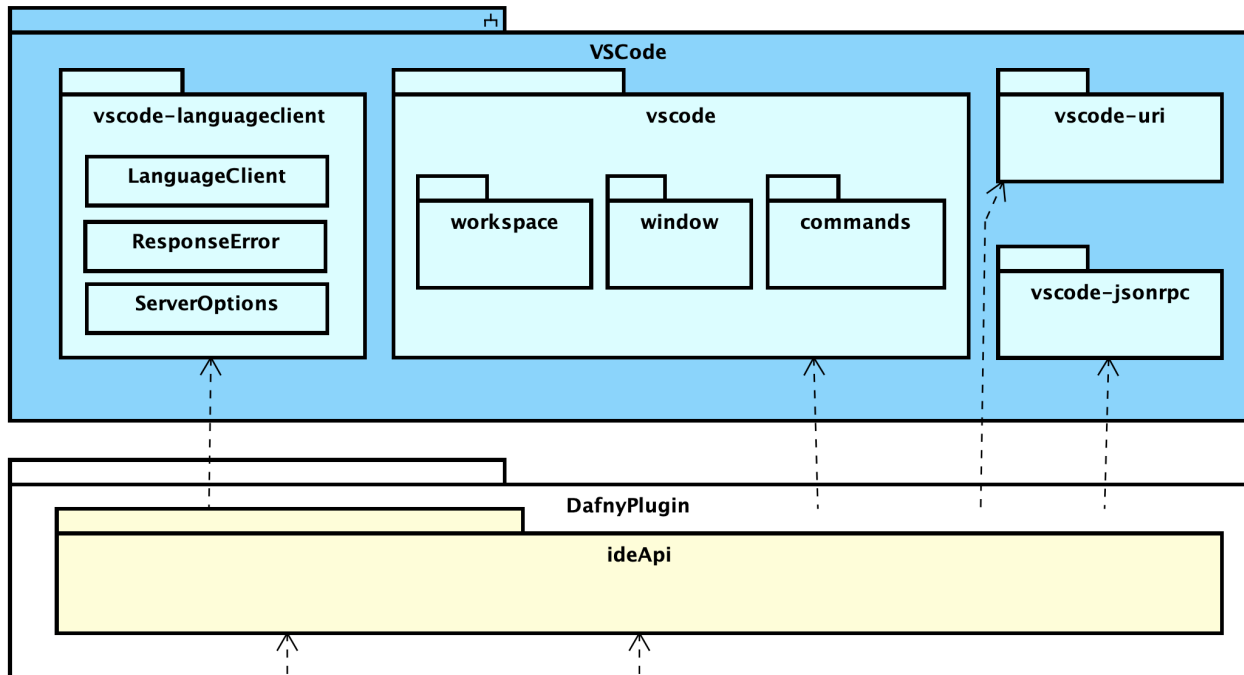


Figure 51: Extract of the Modules and Interfaces Implemented by the Module `ideApi`

However, we have not yet implemented our own adapter classes and interfaces in our refactoring. This has the following reason: Through this encapsulation, developers ideally only need to make changes in a central location to migrate the plugin to another IDE. In the worst case, however, he/she will have to reimplement the entire client code. It depends on the IDE for which the plugin should be adapted, and whether TypeScript is supported by the IDE's plugin development or not.

We did not analyze in detail in what languages plugins have to be developed for other editors. For example, for the plugin development of Atom, JavaScript [30] can be used, while for Eclipse, mostly Java [31] is used.

#### 6.1.4 Download of the Dafny Language Server

Since the original Dafny plugin included the language server (which was written in TypeScript) in the plugin itself, we assumed in the preceding semester thesis that we could integrate our new Dafny language server directly in the plugin, too and deploy it in to the VSCode marketplace.

During the design phase we noticed that this would be an unpleasant solution. In the Dafny project - where our language server will be integrated - builds are created per operating system during releases [32]. So we would have to install the builds for all platforms in our plugin, which would make little sense. Windows users would download macOS builds, too and vice versa as shown in figure 52.








 <a href="#">dafny-2.3.0.10506-x64-debian-8.11.zip</a>	43.7 MB
 <a href="#">dafny-2.3.0.10506-x64-osx-10.14.1.zip</a>	38.5 MB
 <a href="#">dafny-2.3.0.10506-x64-ubuntu-14.04.zip</a>	42.3 MB
 <a href="#">dafny-2.3.0.10506-x64-ubuntu-16.04.zip</a>	44.2 MB
 <a href="#">dafny-2.3.0.10506-x64-win.zip</a>	22.3 MB
 <a href="#">Source code (zip)</a>	
 <a href="#">Source code (tar.gz)</a>	

Figure 52: Dafny Release 2.3.0 Builds

Furthermore, it would have been impossible for us to update client and server independently of each other. Therefore, the language server is now downloaded by the client. Each time the language server is started or restarted within the client, it checks whether the latest version is installed. If not, the local Dafny language version is removed and the latest version is downloaded.

Therefore, the original TypeScript code of the old language server, which downloaded the Dafny client, was analyzed [33]. From this, we have implemented the core functionality in a clean interface: `ILanguageServerInstaller`. Since our language server is not yet integrated into the official build, we have implemented our own, temporary implementation of the downloader as `LanguageServerInstaller`. As soon as our Dafny language server is integrated in the Dafny project, the component can easily be replaced, because the plugin components program against the interface. Minimal adjustments are required to replace the component. In some cases, more suitable names were chosen for methods in the interface.

### 6.1.5 Configurability

There are basically two gradations of configurations. First, there are the configurations for developers in the `stringResources` component. This includes for example Dafny file extensions and other settings which are not to be changed directly by the user.

Second, there are the user configurations, which can be changed by the plugin user via the settings of VSCode. In figure 53, the configurations offered to the user are listed. There is an extract of the plugin description, which can be seen by users in the plugin marketplace.

Setting	Description	Default
<code>dafny.languageServerExePath</code>	Relative path to the <i>Dafny</i> language server executable ( <code>DafnyServer.exe</code> ).	<code>../../dafny-language-server/Binaries/DafnyLanguageServer.exe</code>
<code>dafny.compilationArgs</code>	Optional array of strings as <i>Dafny</i> compilation arguments.	<code>[ "/compile:1", "/nologo" ]</code>
<code>dafny.monoExecutablePath</code>	Monos absolute path. Only necessary if mono is not in system PATH (you'll get an error if that's the case). Ignored on Windows when <code>useMono</code> is <code>false</code> .	
<code>dafny.useMono</code>	Only applicable to <i>Windows!</i> Requires <i>.NET</i> 4.6 or higher when set to <code>false</code> .	<code>false</code>
<code>dafny.colorCounterExamples</code>	Customize the color (HEX) of Counter Examples. There are two default colors: for dark theme ( <code>#0d47a1</code> , <code>#e3f2fd</code> ) and light theme ( <code>#bbdefb</code> , <code>#102027</code> ). This color setting will override both defaults.	<code>{ "backgroundColor": null, "fontColor": null }</code>

Figure 53: Part of the Plugin README .md

For example the colors of the displayed *CounterExample* can be changed or other compile arguments can be set for Dafny as default. To do this, the plugin user can simply open his settings in VSCode and either accept adjustments directly there, or he will be forwarded to `settings.json` as shown in figure 54.

## Dafny extension configuration

### Dafny: Color Counter Examples

Customize the color (HEX) of Counter Examples. There are two default colors: for dark theme (`#0d47a1`, `#e3f2fd`) and light theme (`#bbdefb`, `#102027`). This color setting will override both defaults.

[Edit in settings.json](#)

### Dafny: Compilation Args

Optional array of strings as Dafny compilation arguments

[Edit in settings.json](#)

### Dafny: Language Server Exe Path

Relative path to the `DafnyLanguageServer.exe`

```
../../dafnyLanguageServer/Binaries/DafnyLanguageServer.exe
```

### Dafny: Use Mono

Figure 54: Part of the Dafny Extension Configuration in VSCode

## 6.2 Server

In this section, the server implementation is discussed. Just as in the chapter Design, the server structure will be analyzed from top to bottom, starting with the `Main`-method and ending at the utility layer. The implementation of the symbol table is split into a separate section.

### 6.2.1 Server Launch

The server starts by executing the `Main` method. As done in common practice, it is kept very short. All it does is launching the language server, which is already handled by another class. The full `Main`-method is shown in listing 20.

---

```
1 public static async Task Main(string[] args)
2 {
3     DafnyLanguageServer languageServer = new DafnyLanguageServer(args);
4     await languageServer.StartServer();
5 }
```

---

Listing 20: Main Function

The launch of the server itself is divided into four stages. First, preparational work is done. This already happens in the constructor of the language server. Preparation includes

- Reading and processing config variables.
- Setting up the logging framework.

Second, the actual server is launched. The logger will directly be injected and all handlers are registered. In the third stage, once the server is running, a message sending service is instantiated to notify the client about the successful server start and, if any, errors occurred during start up. Lastly, the console output stream is redirected to keep the language server stream isolated. The constructor and the `StartServer`-method are partially shown in listing 21.

---

```
1 public class DafnyLanguageServer{
2     public DafnyLanguageServer(string[] args)
3     {
4         var configInitializer = new ConfigInitializer(args);
5         configInitializer.Setup();
6         configInitErrors = configInitializer.Errors;
7         log = LoggerCreator.GetLogger();
8     }
9
10    public async Task StartServer()
11    {
12        log.Debug(Resources.LoggingMessages.server_starting);
13        server = await LanguageServer.From(options => options
14            .WithHandler<TextDocumentSyncTaskHandler>()
15            .WithHandler<RenameTaskHandler>()
16            ...
17        );
18        ExecutePostLaunchTasks();
19        await RedirectStreamUntilServerExits();
20        log.Debug(Resources.LoggingMessages.server_closed);
21    }
22 }
```

---

Listing 21: Starting the Language Server

## 6.2.2 Handler

Handlers are passed to the language server and are called whenever the language server receives a corresponding request. Services, such as logging or workspace management, can be injected and are thus available to each handler. As discussed in chapter 5, handlers are based around a `Handle` method. For example, every time the server receives a `textDocument/Definitions` request, that `Handle` method will be called. The parameter and return types are specific per request. *GoToDefinition* would pass a text document location as input, namely the cursor position, and it expects a `LocationLink` as response, namely where the cursor should jump to. Own requests can be realized according to section 4.2.2 by defining an own interface.

All handlers require two additional methods, apart from the actual `Handle`:

- `GetRegistrationOptions`: This method is called when the handler is registered. It allows to set options at the time the server is started. Such an option is, for example, after which characters *AutoCompletion* should be automatically triggered.
- `SetCapability`: It allows to set handler-specific capabilities, such as if the *Rename*-handler will also support a 'prepare rename' feature.

A lot of code for these classes is always identical and was thus extracted to a generic base class. This concerns the creation of a logger out of the logger factory and the handling of errors.

To keep all of this separated from the actual tasks, it was targeted to just forward the request to a core-provider. This could be well achieved. Many handlers look just as shown in listing 22.

---

```
1 public async Task<CompilerResults> Handle(CompilerParams request,
2     CancellationToken cancellationToken)
3 {
4     _log.LogInformation(string.Format(Resources.LoggingMessages.request_handle,
5         _method));
6     try
7     {
8         FileRepository f = _workspaceManager.GetFileRepository(request.
9             FileToCompile);
10        return await Task.Run(() => f.Compile(request.CompilationArguments),
11            cancellationToken);
12    }
13    catch (Exception e)
14    {
15        HandleError(string.Format(Resources.LoggingMessages.request_error,
16            _method), e);
17        return null;
18    }
19 }
```

---

Listing 22: Handling Compilation

The file to compile is given as an argument. The corresponding repository can be requested from the `WorkspaceManager`, which is injected to all handlers. The request is then forwarded to the according provider, which will calculate the results. All information required, such as the precompiled program, are



available from within the `FileRepository`. In case of an error, a message is sent to the user and the error is logged within the `HandleError` method.

Some handlers take additional actions, such as awaiting the result of the provider, and then sending user feedback according to the outcome. This is done for *GoToDefinition* for example. If the request was triggered at a declaration, an additional user message is sent. By handling all communication inside the handler, the message sending service does not have to be passed downwards to the core logic component.

### 6.2.3 Core

Within this package, the result of a request is assembled. While it may sound like complex logic is happening here, it turned out that for most cases, all information is just available by the symbol table. Consider the example of *GoToDefinition*. A code excerpt is shown in listing 23. As you can see, the actual task of finding the definition is resolved by the symbol table engine and just available as a property. The provider only has to assemble the result within the proper wrapper class.

---

```
1 ISymbol symbolAtCursor = _manager.GetSymbolAtPosition(uri, line, col);
2 ISymbol originSymbol = symbolAtCursor.DeclarationOrigin;
3
4 var position = new Position((long)originSymbol.Line - 1, (long)originSymbol.
    Column - 1);
5 var range = new Range { Start = position, End = position };
6 var location = new Location { Uri = originSymbol.FileUri, Range = range };
7
8 List<LocationOrLocationLink> result = new List<LocationOrLocationLink>();
9 result.Add(new LocationOrLocationLink(location));
10 return new LocationOrLocationLinks(result);
```

---

Listing 23: *GoToDefinition*, Core Provider

There are a few features that had to be extended with additional logic. These are described in section 6.4

### 6.2.4 Workspace

The workspace is a component representing any opened files by the client. Thus, it naturally consists only of a single property. This is a dictionary, mapping a file-location to an internal file-representation:

---

```
1 private readonly ConcurrentDictionary<Uri, FileRepository> _files;
```

---

Listing 24: Workspace Property

It offers methods to retrieve files and to update them. Since updates can be done in two different kinds, *incremental* or *full*, the update method is overloaded for both cases. The update requests are forwarded to the class `FileRepository`, which is used as the internal representation of a file. It of course contains the source code. However, this is not done directly as a string property, but wrapped in a class `PhysicalFile`. Thus, the actual representation on the hard disk is separated even further. The `PhysicalFile` class can

then also take responsibility for applying file updates.

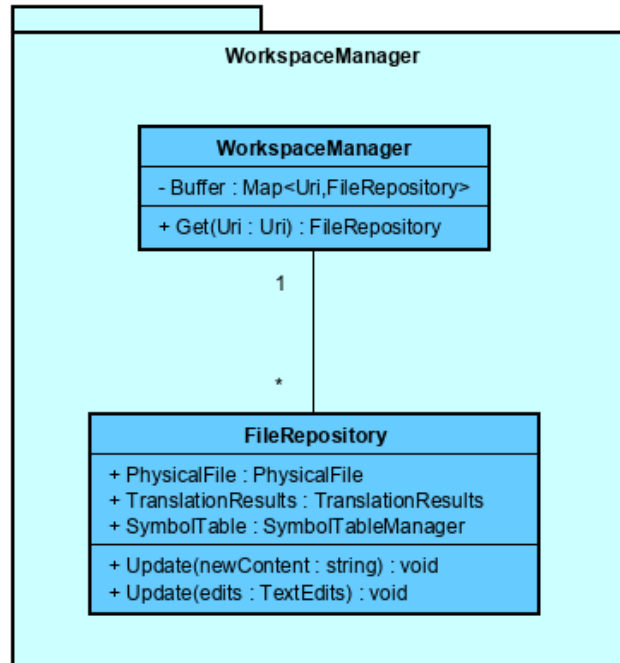


Figure 55: WorkspaceManager and FileRepository

Aside from the file content, each `FileRepository` will also contain `TranslationResults`. `TranslationResults` is a wrapper class for anything provided by the Dafny backend:

- Could the file be parsed?
- Could it be verified?
- Is it logically correct?
- What errors and warnings occurred?
- How far could it be compiled?
- What internal compilation results could be produced for later reuse?

Last but not least, the newly implemented symbol table is also attached to the file repository. To obtain all of these results, the class simply invokes the `SymbolTableGenerator` and the `DafnyTranslationUnit`. Thus, all information about a file is accessible from within the file repository.

### 6.2.5 DafnyAccess

Dafny Access is the package invoking the Dafny backend to obtain verification results. The core class in this package is the `DafnyTranslationUnit`. It was partially taken over from the pre-existing project, but as part of this bachelor thesis it was refactored and simplified.

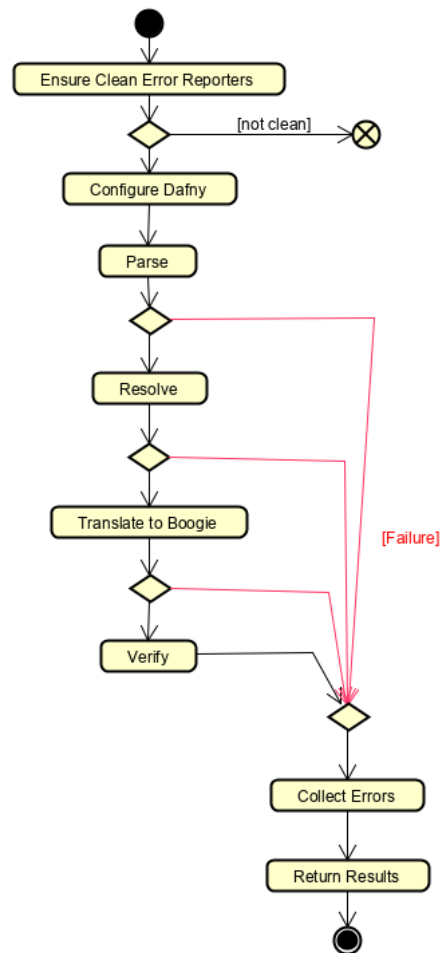


Figure 56: `DafnyTranslationUnit` Sequence Diagram

In the constructor, the translation unit accepts a `PhysicalFile`. The class then offers a single method `public TranslationResults Verify()`. Within the method, the following sequence of events as illustrated in figure 56 occurs:

1. It is checked that the instance has never been used before. This is to ensure that all error reporters are empty. Otherwise, errors would be reported multiple times.
2. Next, Dafny is configured. This includes the registration of the Dafny error reporter and setting any options to default. The only non-default option is that the engine is supposed to generate a model file, which can later be used for counter example extraction. The configuration is shown in listing 25.
3. The Dafny parser is called. This step will report any syntax errors.
4. The Dafny resolver is called, if parsing was successful. This step will do semantic checks, such as type checks. A `DafnyProgram` results upon success.
5. If successful, the precompiled `DafnyProgram` will be split into `BoogiePrograms`.
6. The Boogie execution engine is invoked to perform logical correctness checks on the `BoogiePrograms`.

7. Any errors that were reported are collected, converted and assembled in the field `_diagnosticElements`.
8. All results are wrapped by the `TranslationResult` class, providing the diagnostics, the Dafny program and the boogie programs. Also, within the property `TranslationStatus`, it is remarked how far the verification and translation process succeeded.

Note that in the following snippet, Dafny options are directly calling Dafny's backend.

```
1 private void SetUpDafnyOptions()
2 {
3     DafnyOptions.Install(new DafnyOptions(_reporter));
4     DafnyOptions.Clo.ApplyDefaultOptions();
5     DafnyOptions.O.ModelViewFile = FileAndFolderLocations.modelBVD;
6 }
```

Listing 25: Setting up Dafny Options

### 6.2.6 Utilities

The bottom layer of the project is formed by the utility layer. It contains three packages:

- Tools
- Commons
- Resources

`Commons` contains classes that are used within multiple parts of the code, such as `TranslationResults` and were already discussed. `Resources` contains string resources and were realized with `.resx` files. Within the `Tools` package, a variety of services can be found that do not necessarily directly correspond to Dafny, but are useful within the language server environment.

#### Config Initializer

This class is used prior to the server launch and initializes a few config settings. The settings are stored within the static class `Commons/LanguageServerconfig.cs`. The config initializer will first of all set hard coded default values. This is to avoid any kind of null pointer exceptions. Afterwards, the file `Config/LanguageServerConfig.json` is parsed with Newtonsoft's `Json.NET` library [34]. Any available values will be written to the static configuration class. Unknown or illegal values will not be set and errors are added to an error reporter. Finally, the launch arguments are parsed, again overwriting the config settings if applicable or reporting errors otherwise. A simple argument parser was implemented manually. Alternatively, a library could have been used for this task such as *Command Line Parser* [35]. The config initializer is implemented exception-safe. It will run to completion and at worst just provide default values. Errors can later be extracted from the `ErrorReporter`. Each task is carried out by a dedicated component.

#### LoggerCreator

This class simply sets up a *Serilog* [36] logger. For this purpose, the following information is extracted from the `LanguageServerConfig`:

- Minimum loglevel
- Path of the logfile

### MessageSenderService

This is a simple class accepting a `ILanguageServer` in the constructor. Afterwards, it provides methods to send notifications to the client. Similar to logging, methods for each severity level are available, such as `public void SendError(string msg)`.

### ReservedWordsProvider

This is a class providing a set of words, that are not suited for identifiers. This is, for example, `method`, `class`, or `return`. The class tries to read and parse `Config/ReservedDafnyWords.json`, which can be user adjusted in case the Dafny specification changes. If the file cannot be read or has a wrong format, a hard coded default list is used which was taken out from the Dafny Reference Manual [18]. A `HashSet` was used to provide fast access to the set.

While this component is specifically used solely for the feature *Rename*, it was extracted to be also available at other spots if required for future features.

## 6.3 Symbol Table

This chapter describes the implementation of the symbol table. The symbol table was designed to handle four different tasks:

- Providing Symbol Information
- Symbol Table Navigation
- Symbol Table Generation
- Symbol Table Management

### 6.3.1 Symbol Information

Obtaining information about a symbol was realized by a wrapper class. Logic was extracted from this class as far as possible to keep it short and concise. As we have seen in chapter 5, this class needs to contain a lot of properties. The most important properties are:

- Name
- File
- Position in file
- Body location, if any
- Kind and type
- Link to parent symbol
- Link to declaration symbol
- Hash with all child symbols (only declarations)
- List with all descendants (any symbol occurring in the body)
- List with all usages of the symbol
- Base classes
- Parameters
- The associated module
- Link to the associated default class for quick access

The provided properties are supposed to facilitate working with the symbol table. For example, if a symbol's definition cannot be found with regular methods, the property with the associated default class can be accessed to search the symbol there. No separate logic to find the default class is necessary. This is advantageous, since the default class is in the global namespace and not a direct ancestor of a symbol.

To enable efficient access to the children of a symbol, we have opted for a key-value data structure. The key is the child symbol's name, the value the actual `SymbolInformation` object. This hash structure enables access to a child symbol with a runtime of  $O(1)$ . For example, if a module `M` contains a class `C`, and within the class there is a method `f00`, one can simply start from the root symbol and navigate through the hash

maps in constant time. The `[]` operator was overloaded to make this as convenient as possible:

```
rootSymbol["M"]["C"]["foo"].
```

Technically, the symbol table is more of a double linked tree, than a table. To be consistent, the structure will be called symbol table.

While this is very fancy, the convenience comes at a price. Many properties do not apply for all kinds of symbols. Consider the following code segment in listing 26.

---

```
1 method foo() {
2     var x := 5;
3     print x;
4 }
```

---

Listing 26: Example Code Regarding Symbol Information

The symbol `foo` profits by almost all properties. It can have children (the variable `x`), it has a parent, and it can be used elsewhere in the code. Since `foo` is a method declaration, the declaration property of the symbol does not make sense. In this case, it just points to itself. This way, features like *GoToDefinition* will just jump to the same symbol, which is the expected behavior.

The declaration of `x` in the second line will have many null values within the `SymbolInformation`. While the parent is `foo`, it can not have any children. Since it is a variable definition, it still can have usages though.

The final usage of `x` in the last line (which we also consider a symbol), does not even have usages, since it is a usage itself. Thus, many properties are null for that last symbol.

Unused properties are set to `null`. This causes the risk of `NullPointerExceptions`, but should save a lot of memory compared to empty lists or dictionaries.

### 6.3.2 Symbol Table Creation

The symbol table generator accepts a precompiled Dafny program in the constructor. The generator offers a public method `GenerateSymbolTable`. It will first of all create a virtual root symbol. Any other symbols will be attached to the root node as descendants. The root node is also the final return value.

All modules (similar to `C#` or `C++` namespaces) will be extracted out of the Dafny program using functionality provided by Dafny. The modules will be sorted by depth, so that top level modules will be treated first and nested modules can be attached properly later on.

Once the modules are sorted by depth, the algorithm iterates over each of them. The proper parent symbol will be deduced. For a top level module, this is just the root symbol. Otherwise, for nested modules, the parent module will be located. Once the module is attached to the proper parent, the module will accept the declaration-visitor, which is described below. Once it has completed, all symbol declarations are registered in the symbol table. A second iteration is then started, using another visitor, which will ignore declarations but run through all method bodies and take care of symbol usages.

This way, symbols that are declared after the first usage can be found as well.

## Visitors

As already mentioned, the whole symbol table generation is realized using the visitor pattern. For that, Dafny code had to be adjusted to offer a `Accept(Visitor v)` method. This method will basically just navigate through the internal Dafny symbol representation. For example, when visiting a method, one would like to register the method itself. This is done by the expression `v.Visit(this)`. However, a Dafny-method also contains a Dafny-ensures statement, which may contain further symbols. Thus, all statements within the ensures clause have to be visited. The `Accept`-method will now just forward the call, using `foreach (var e in this.EnsureStatements) e.Accept(v)`. The same applies for method parameters and other items like the `requires` clause. Finally, the body of the method is to visit using `foreach (var stmt in this.Body) stmt.Accept(v)`. Once everything is done, the scope of the method is left by calling `v.Leave(this)`.

If you recall the paragraph before the last one, it was said that two runs are performed. One to capture all declarations, and one to visit all method bodies. Thus, the visitor has a boolean property `GoesDeep`, which decides whether method bodies are visited or not. The final `Accept` method for a method looks as shown in listing 27. The method is shortened, there are more clauses such as the `requires` clause.

---

```
1 public override void Accept(Visitor v)
2 {
3     v.Visit(this);
4     if (v.GoesDeep)
5     {
6         foreach (var ens in this.Ens)
7         {
8             ens.Accept(v);
9         }
10        foreach (var stmt in this.Body)
11        {
12            stmt.Accept(v);
13        }
14    }
15    v.Leave(this);
16 }
```

---

Listing 27: Accepting a Visitor

Note that the method is marked with the `override` keyword. This is the case since every `AST-Element` is either a statement, a definition or a declaration, among others. A virtual `Accept` method was implemented for these top level classes, which uses a default implementation. In case a specific `AST` element got forgotten by us, it will just use the default implementation, which is shown in listing 28.

---

```
1     public virtual void Accept(Visitor v)
2     {
3         v.Visit(this);
4         v.Leave(this);
5     }
```

---

Listing 28: Default Accept

On the other hand of the acceptor, there is the actual visitor. The visitor has to implement `Visit` methods for each of the AST elements that it is supposed to visit, for example the class `Method` from the preceding example.

The visitor itself will build up the symbol table. For that, it stores the current scope in a property. At the beginning this is the module the iteration was started with. When visiting a method, the current scope is always some kind of class that was visited before.

The visitor will create a symbol for every element it is visiting. To do so, all the properties we have seen earlier need to be populated. The property `Parent` can just be set with the scope the visitor has stored. Since the method itself will have its own body, the new scope will then be set to the method-symbol, which is just being created. All symbols that will be visited afterwards - which is anything inside the method - are then attached to the method. Once the method is done, `Leave()` is called, which will reset the scope to the parent scope for the next item to be visited.

Let us continue the example of visiting a `method`. This is a declaration, and thus will be treated by the first visitor, which will register all declarations outside method bodies. The `Visit`-method is shown in listing 29. Take note how the symbol is created: All required information is taken from the AST element `Method o` that is visited. This includes the name, the position, and so on.

---

```
1 public override void Visit(Method o)
2 {
3     var symbol = CreateSymbol(
4         name: o.Name,
5         kind: Kind.Method,
6
7         positionAsToken: o.tok,
8         bodyStartPosAsToken: o.BodyStartTok,
9         bodyEndPosAsToken: o.BodyEndTok,
10
11         isDeclaration: true,
12         declarationSymbol: null,
13         addUsageAtDeclaration: false,
14
15         canHaveChildren: true,
16         canBeUsed: true
17     );
18     SetScope(symbol);
19 }
```

---

Listing 29: Visiting a Method, First Visitor

The `CreateSymbol` method will set all properties accordingly. That means a symbol that can have children will be initialized with a list for children, while a symbol that cannot have children will just have a null entry there. Note that the end of the method, the scope is set to the just created symbol for future visitations.

The second visitor, which is responsible for method bodies and symbol usages, will also visit declarations. The visitor has actually no choice to skip them, since the `Accept`-method decide what is visited in which order. However, the second visitor no longer creates a symbol for them. Instead, the already created symbol is located and set as the environmental scope. The `Visit`-method thus gets reduced to listing 30.



---

```
1 public override void Visit(Method o)
2 {
3     var preDeclaredSymbol =
4         FindDeclaration(o.Name, SurroundingScope, Kind.Method);
5     SetScope(preDeclaredSymbol);
6 }
```

---

Listing 30: Visiting a Method, Second Visitor

To find that pre-declared symbol, the symbol table navigator is invoked. It will just iterate from parent to parent and returns the first symbol that is a declaration and matches the name. Challenges occurred when a symbol is defined in global scope or in an inherited base class. Both difficulties were resolved by adding separate checks for them. This is completely done by the `Navigator` component and not part of the visitor.

The second visitor will now also visit method bodies, since the `GoesDeep` property is set to true. The `Accept` method as seen in listing 27 is no longer stopping, once the body is treated. Within the body, local variables exist. Despite local variables being declarations, they were not handled by the first visitor, which is actually responsible for declarations. However, this is fine since local variables are not accessible before they were not declared. Furthermore, symbol *usages* such as method calls or variable usages are now encountered. The visitor will create proper symbols for these. Since these are symbol usages, it is not sufficient to just create a symbol and attach it to the parent scope. The following additional tasks have to be accomplished:

- Where is the symbol declared?
- Add a usage to the symbol's declaration.

To find the declaration, the symbol table navigator is called again just as before. Once the declaration is known, it is trivial to add a usage.

### 6.3.3 Symbol Table Navigator

To operate on the (partially) constructed symbol table, a separate component to navigate was created. It basically has two procedures. Remember that the data structure of the symbol table is basically a double linked tree.

- `TopDown`: Starting from a node, the navigator dives downwards and searches a specific symbol.
- `BottomUp`: Starting from a node, the navigator climbs upwards the tree and searches a specific symbol.

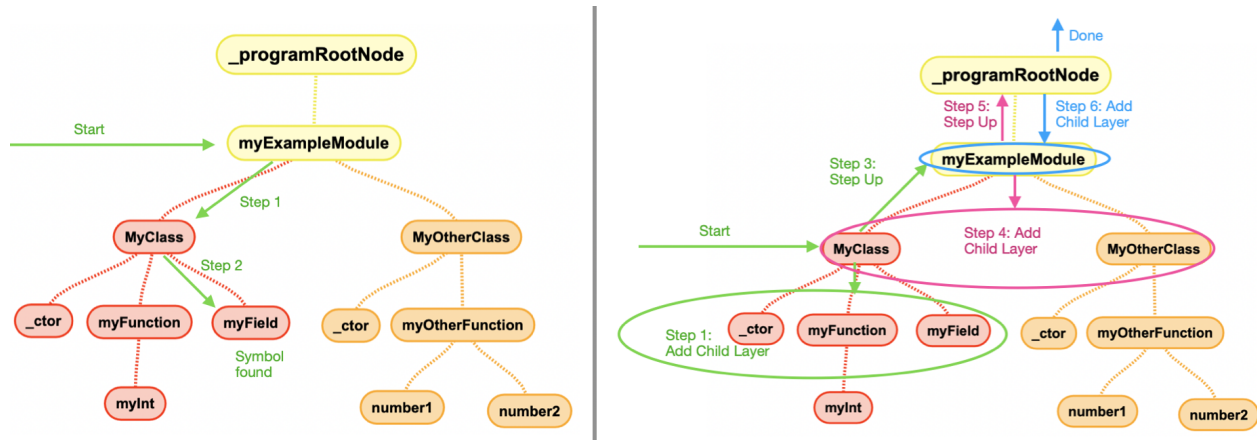


Figure 57: TopDown (Left) and BottomUp (Right) Visualized

Both options are implemented so that they can return a single, first match, or all symbols that match a criterion. To illustrate this, two examples are shown in figure 57. On the left side, the system searches specifically from top to bottom for the symbol that best matches a position specification. On the right side, starting from one symbol, all symbols in the available scope are searched.

With this type of tree inspection, the runtime is reduced, as it is no longer necessary to visit every symbol. The navigator overcomes the following tree challenges:

- Where is the definition of a symbol?
- What symbol is at the current cursor position?
- What *AutoCompletion* suggestions are indicated?

How these challenges were overcome, is discussed below.

### Where is the definition of a symbol?

This challenge uses the bottom-up procedure. It is started from whatever scope the visitor is working on. In every scope, the navigator will iterate over all symbols and search one, that is a declaration and matches in name and kind. For example, if we start inside a method, every symbol already visited in that method will be checked. If no matching declaration was found, the navigator will move to the parent of the method, which must be a class as shown in figure 58. Then, all symbols within that class are searched. Now, the navigator may find a class method that actually matches the search criteria. Since all declarations were registered beforehand, the process works even if the symbol is declared after the usage, unless it is a local variable, whereby it must be declared anyway before usage.

If the symbol cannot be found, the algorithm will search the default scope at the end as a last option to locate the symbol. Since the symbol tree is moved upwards, the process takes  $O(\log(n))$  time. It is executed for every occurring symbol, thus the building of the symbol table takes  $O(n * \log(n))$  time, while  $n$  denotes the amount of occurring symbols.

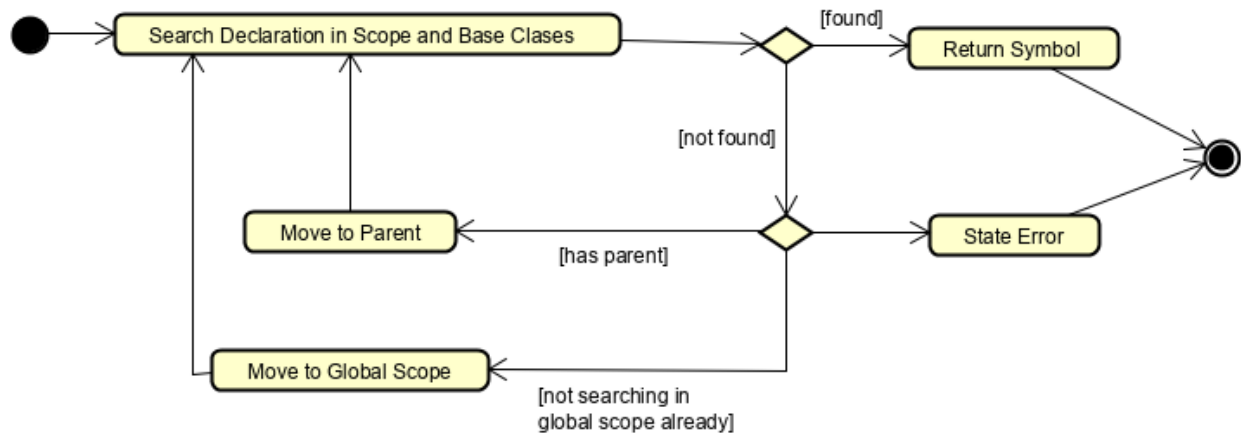


Figure 58: Finding a Declaration Using the Navigator

### What symbol is at the current cursor position?

For this task, top-down navigation is used. Starting from the root symbol, for each child it is checked if it wraps the position. If not, the symbol is ignored. But if so, iteration is recursively called on that child. Regarding figure 59, imagine we hand the blue line as the cursor position. The algorithm will then decide, that the cursor position is within module  $M$ . Afterwards, it will check all children of  $M$ . It will find out, that class  $Z$  is not surfacing the cursor position, and move to the next one. When checking class  $C$ , wrapping is successful, thus the next iteration will be done within  $C$ . Finally, variable  $c$  will be returned as the best match. Default namespaces and default classes had to be treated separately for this case. Since the tree is moved along a single branch, the runtime for this process is  $O(\log(n))$ .

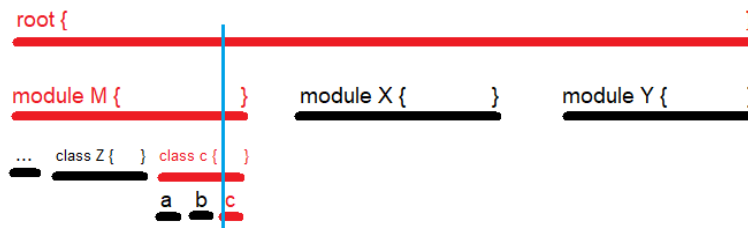


Figure 59: Top Down Navigation Path in Blue

### What *AutoCompletion* suggestions are indicated?

To build up autocompletion suggestions, all declarations available at a certain locations must be figured. First of all, the previous `TopDown` algorithm is used to find the current location. Then, all declarations in the current scope are requested, by using `BottomUp` again and requesting everything that is a declaration. This algorithm is diving once into the tree, and going once backup, thus requiring  $O(\log(n))$  time.

The navigator offers further methods to visit the whole tree if that is desired by the caller. The method `TopDownAll` will visit all branches, not just the one wrapping a certain location. Consequently, `TopDownAll` requires  $O(n)$  runtime and should not be used if possible. We took advantage of that method to print a complete symbol tree during debugging.

### 6.3.4 Symbol Table Manager

The manager is a rather simple component and can be used as an access point to perform operations on the symbol table. It is constructed with a root symbol of a fully generated symbol table and contains a navigator. It then offers methods such as `ISymbol GetSymbolAtPosition(Uri file, int line, int character)`. That method will use the navigator to provide the user with the desired result. Access to the rather complex navigator is encapsulated this way.

### 6.3.5 Symbol Table Utilities

If a symbol wraps a certain position was first of all a member method of `ISymbolInformation`. However, the class `SymbolInformation` got very long this way and testing against the interface was hard. Thus we decided to extract this very important logic to a dedicated component, which we could test without any effort. The utilities class offers three distinct types of `DoesWrap`-methods.

- Is a position wrapped by the entire range of a symbol?
- Is a position wrapped just by the body of a symbol?
- Is a position wrapped just by the identifier of a symbol?

This differentiation was necessary, since for *GoToDefinition*, only the identifier range matters. But for *AutoCompletion*, the body in which the cursor is matters. Lastly, clauses like `ensures` are in between those two. Figure 60 shows the distinction between the three methods.

```
class MyClass { method foo() {} var x : int; }
```




Figure 60: Top Down Navigation

- Green: Wrapped by the identifier range.
- Yellow: Wrapped by the body range.
- Red: Wrapped by the entire symbol range.

## 6.4 Features

As it was stated in the previous sections, many features just query the symbol table and assemble the information to a proper result format. This is especially the case for *Rename*, *GoToDefinition* and *HoverInformation*. An example was given while explaining the `Core`-package of the server implementation. These features are thus not described in detail any further, since they are nearly trivial. However, other features, especially *AutoCompletion*, *CodeLens* and the ones not depending on the symbol table involve a bit more complexity. In this section, these special cases are described.

### 6.4.1 AutoCompletion

The automatic code completion is an essential feature of every IDE and is a great help for the developer to write code efficiently. Making good suggestions to the user is a complex matter.

Through the new symbol table, it is easy for us to find symbols in the current scope. With the help of the `SymbolTableNavigator`, it is also possible to efficiently filter the proposals by conditions, which can be transferred as predicates.

The main difficulty and thus the complexity of *AutoCompletion* is therefore no longer the actual collection of symbol suggestions, but the evaluation of what kind of suggestions the user wants in the current context and extracting a symbol as entry point for the symbol table navigator component.

We support three fundamentally different types of user desires. To find out the user's desire, the following approach is taken. First, the cursor position is used to determine the current code position in the Dafny source file. Then, it is checked if there is a dot or "new " in front of the cursor position. Depending on whether one of the two triggers was found, a decision is made between the three user desires, as shown in figure 61.

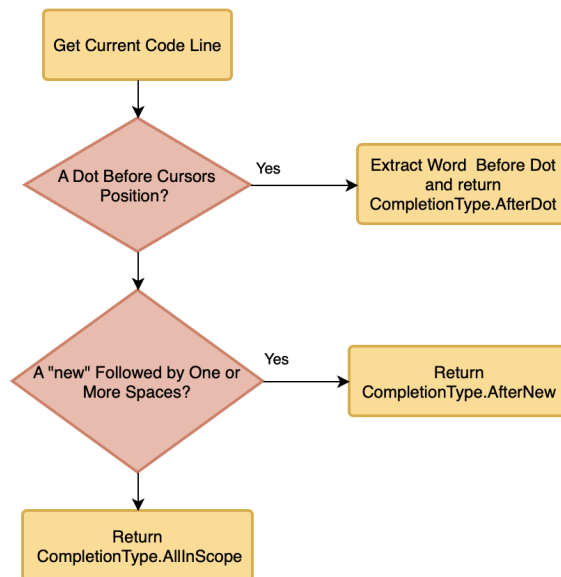


Figure 61: Evaluating Users Desire

### Proposals for the Object Instance

If a dot was found at the position of the cursor, the word before the dot is also extracted from the code line. First of all, the navigator has to find the enclosing scope at the cursor. For that, it is important to use the wrapping method, that checks if the cursor is within the body of a symbol, not just within the symbol's identifier. A proper method was implemented which does that. Afterwards, the related class belonging to the symbol before the dot is searched. During the search, the navigator starts in the scope at the cursor, and then moves up one level until a suitable symbol is found. For example, if the user is typing in the method `myMethod`, the symbol for `myMethod` is returned.

If a matching symbol is found, for example a class named `myClass`, all methods and fields defined inside `myClass` can be extracted using the symbols properties. These are now returned as the final suggestions.

For clarity, the described process is visualized in the following two figures. Listing 31 shows a code example, which leads into the resolving sequence in figure 62.

```
1 class myClass {
2     var ABC: int;
3     constructor () { }
4     method myMethod() { /* do something */ }
```

```

5 }
6 method Main() {
7     var abc := new ClassC();
8     abc.
9 }
    
```

Listing 31: Example Dafny Code - Cursor Position at Line 8

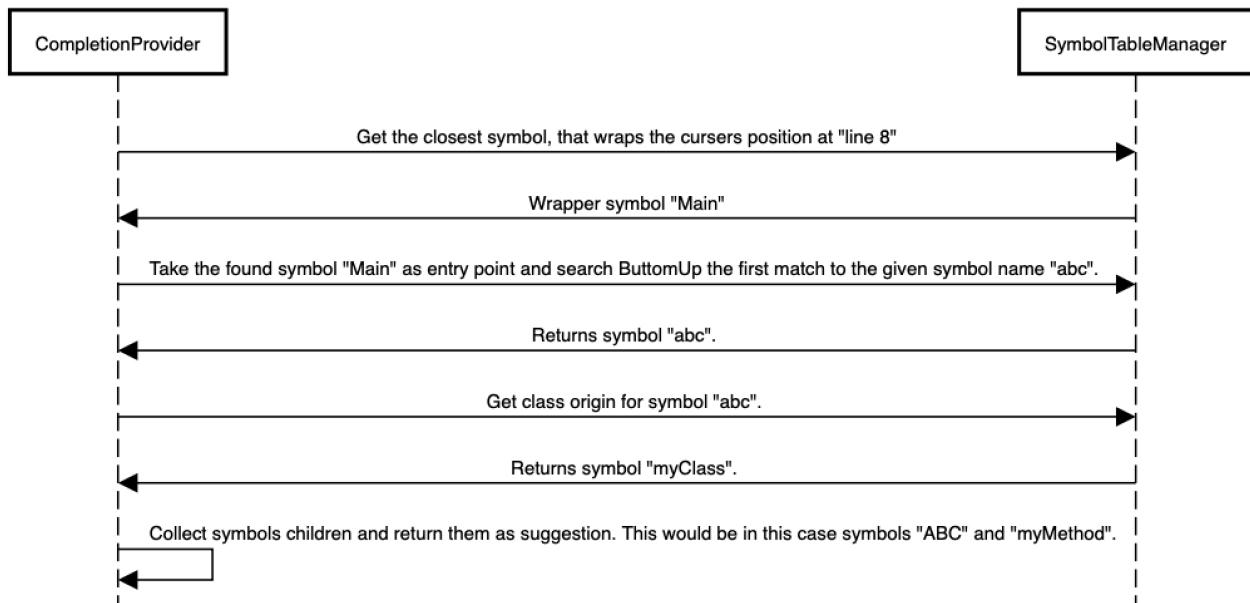


Figure 62: Sequence Diagram for Object Instances

### Class Suggestions

If it is assumed that the user only wants classes proposed, the following approach is followed. As with the proposals for an object instance, the `CompletionProvider` first looks for an access symbol as entry point at the cursor position. Then iteration is performed from the inner scope to the outermost scope and all symbols found are returned as completion suggestions. The list is filtered already during iteration by a passed predicate for class symbols only. This process is visualized in figure 63.

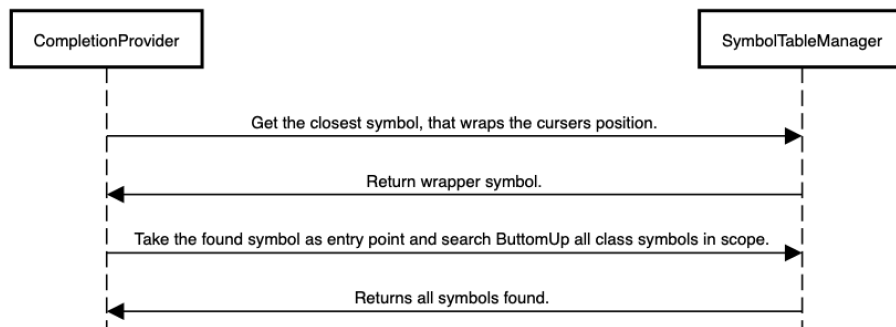


Figure 63: Sequence Diagram for Class Suggestions

### All Available Symbols as Default

If it is assumed that the user would like to have all symbols proposed in the current context, the procedure is basically the same as before. The only difference is that no filter is passed to the symbol navigator. Therefore, all types of symbols are suggested. This process is visualized in figure 64.

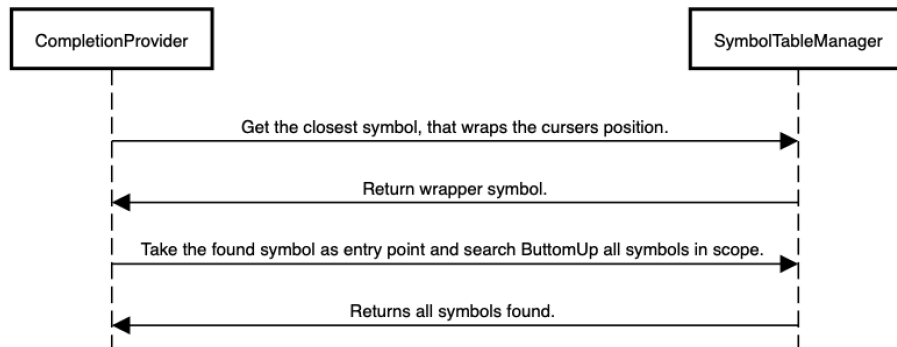


Figure 64: Sequence Diagram for all Symbols in Scope

### Insertion of Parameter Placeholders

Just like in other programming languages, Dafny invokes methods and constructors by round brackets. It was our goal to automatically propose those brackets. This goal could easily be achieved. Whenever a constructor or method is chosen as an *AutoCompletion* suggestion, an opening and closing bracket is added to the text to be inserted.

From certain IDE's such as XCode, you are used to face placeholders for the parameters and their types [3]. An example of that functionality is shown in figure 65 That is why we wanted to implement such a convenient feature for our *AutoCompletion* as well. Unfortunately, we had to realize that this feature is not yet supported natively by the VSCode API [23]. Neither does the LSP support a standard for the explosive insertion of parameters [9].

```
NotificationCenter.default.removeObserver(observer: Any, name: NSNotification.Name?, object: Any?)
```

Figure 65: AutoCompletion in XCode

That there is no simple way to implement this feature does not mean that it cannot be done. But the effort for the implementation in VSCode would be exceptionally high. Therefore we have decided not to support this feature. However, the symbol table stores a parameter list for every method, constructor, and function. Thus, the symbol table is ready to support the functionality. The complete idea how the feature could be implemented is described below.

As with *CodeLens*, you could send additional data to the client for the *AutoCompletion* objects - namely the parameters and their types for methods. Whenever a completion is inserted, a callback function can be called within the client. This function could then additionally insert the parameters as text in the code and register event listeners on keyboard keys like *TAB*. Using *TextEdits*, the registered function would then jump to the next placeholder parameter each time *TAB* is pressed and mark it, so that the user can easily replace it with his own code.

Also, just because VSCode does not yet offer an API to perform such *TAB* jumps between parameters, does not mean that other IDE's, for which Dafny support will be offered later, do not offer such a function. In the

future, VSCode might even add such a function to its API.

### Getting Information About Method Parameters

Usability test has shown that text information about which method receives which parameters would still be very helpful for users. The information provided by the symbol table could be used for that.

When constructing the completion elements, the field `CompletionItem` can contain additional details. Those details could hold the parameter list. This way, the user could be informed about them.

### Suggestions for More Complex Types

As soon as the visitor supports more symbols, *AutoCompletion* can also complete more complex data types. Especially for arrays. For example, it would be helpful if array objects were supported by *AutoCompletion*. In the example shown in figure 66, the property `Length` would then have to be suggested for the array `a`.

```
method Find(a: array<int>, key: int) returns (index: int)
  ensures 0 <= index ==> index < a.Length && a[index] == key
{
  a.
  // abc a
```

Figure 66: Missing Array Support

## 6.4.2 CodeLens

The *CodeLens* functionality is divided into two primary tasks. First, it is shown how often the method or class has already been used. Second, the user can click on the text and a popup window displays all symbol usages. This allows the developer to efficiently jump between different places within a workspace.

### Number of References

To get the reference count, no big effort is necessary thanks to the new symbol table. It contains a list of `Usages`, which can easily be counted. This way, it can also expose dead code, which has never been used as seen in 67.

```
2 references to ClassB
11 class ClassB {
12 |   constructor () { }
    3 references to myMethod
13 |   method myMethod() { /* do something */ }
14 }
15

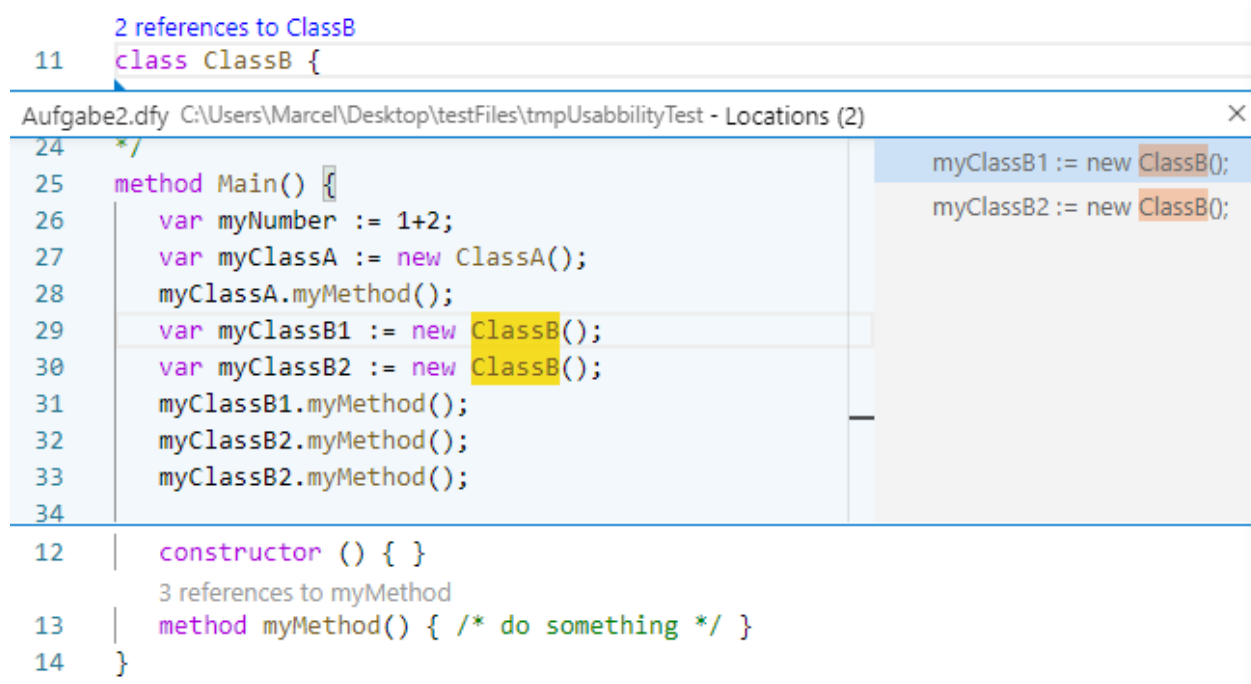
1 reference to ClassC
16 class ClassC {
17 |   var ABC: int;
18 |   constructor () { }
    Not used yet. Can you remove myMethod?
19 |   method myMethod() { /* do something */ }
20 }
--
```

Figure 67: *CodeLens* Shows Number of Counted References



### Popup Windows with Code Snippets

The popup that opens when you click on the grey text with the reference counter is a client feature. It is offered by VSCode [23]. Unfortunately, this is something that cannot be extracted completely to the server. The logic required is trivial, though. Only the function `VSCodeCommandStrings.ShowReferences`, offered by VSCode, is called [23]. The popup display is shown in figure 68.



```

11 class ClassB {
    2 references to ClassB
    24 */
    25 method Main() {
    26     var myNumber := 1+2;
    27     var myClassA := new ClassA();
    28     myClassA.myMethod();
    29     var myClassB1 := new ClassB();
    30     var myClassB2 := new ClassB();
    31     myClassB1.myMethod();
    32     myClassB2.myMethod();
    33     myClassB2.myMethod();
    34
    12 | constructor () { }
    13 | 3 references to myMethod
    13 | method myMethod() { /* do something */ }
    14 }
    
```

Figure 68: *CodeLens* shows Popup with Code Snippets

The preparation of the data is already done in the language server. For each symbol declaration, the *CodeLens* object is enriched with further information. The client just has to display them. This is shown in listing 32.

```

1 var args = new
2 {
3     Uri = _uri,
4     Position = position, // where to show the CodeLens popup
5     Locations = locations // what should be displayed inside the popup
6 };
7 Command command = new Command
8 {
9     Title = msgTitle,
10    Name = "dafny.showReferences",
11    Arguments = new JSONArray(JsonConvert.SerializeObject(args))
12 };
13 return new CodeLens
14 {
15     Data = _uri,
16     Range = range,
    
```

```
17     Command = command
18 };
```

---

Listing 32: LSP Handler Implementation

To each *CodeLens* element, a `Command` object is also appended. The command's name is the function called at the client, when the grey text is clicked.

In addition, the `Command` is given the positions it should display inside the popup.

If the function `dafny.showReferences` is called, the method `VSCoDeCommandStrings.ShowReferences` from the VSCode API is invoked and the arguments prepared in server are passed through.

Because the whole logic for *CodeLens* is in the server, integration tests and unit tests were written on the server side for this feature.

### Support for File Includes

*CodeLens* works very well over multiple files. But only if other files are included. In case the currently opened file is included by another file, *CodeLens* does not know that it gets used by a third Dafny program.

This problem should solve itself once a global symbol table for the complete workspace has been implemented, instead of a single table per file. Symbols will then be aware of usages across the complete workspace. More information about the global symbol table is described in section 5.4.3.

### 6.4.3 Compilation

Every time a file gets updated, the whole Dafny backend is triggered and the results are stored in a `FileRepository`. This includes the precompiled `dafnyProgram`, which is then available for further reuse. The compilation provider will take advantage of that and use the precompiled item, and just forward it to the Dafny compiler engine. Prior to the compilation, custom compilation arguments are installed, if the user provided any. The process is fully integrated into the Dafny backend. The compiler invocation can be seen in listing 33.

---

```
1 DafnyDriver.CompileDafnyProgram(dafnyProgram, filePath, otherFiles, true,
    textWriter);
```

---

Listing 33: Calling the Dafny Compiler

The provider will check if any errors occurred during the process and return the outcome within a wrapper class. The handler can then read the outcome and send a proper message to the user.

### 6.4.4 CounterExample

This feature is based on the model file, which is generated during verification. The model file is a key-value store generated by Boogie. It contains several *states*. Each state tracks the content of variables during different stages of the proof. Of interest is primarily the *initial* state, since this one contains the information how variables need to be set initially to achieve a *CounterExample*.

The *CounterExample* provider reads the model file and uses the Boogie backend to convert it into a useful format, which is called `ILanguageSpecificModel`. Afterwards, it extracts the initial state from the model. Out of the remaining state, all key-value-pairs are extracted, filtered, assembled and returned. Many values are internal Boogie references and cannot be resolved. These just look like `T@U!val!12`. Such values are replaced with the text `[Object Reference]`. Values providing no information at all are completely removed from the *CounterExample*. The component will also transform information into a more human readable format, e.g. `((- 12)) → -12`.

## 6.5 Mono Support for macOS and Linux

One of the core objectives was to provide support for multiple platforms. This means that in addition to Windows, macOS and Linux should be also supported.

In the preceding thesis, we had to switch from .NET Core to .NET Framework [3]. The reason for this was that although OmniSharp was compatible with .NET Core, the Dafny backend was not [3].

.NET Core has native support for Linux and macOS. With .NET Framework, however, a Windows executable is generated upon compilation. Therefore, it is necessary to rely on Mono for Unix based operating systems. Mono allows to execute `.exe` files on said systems [37].

Unfortunately, in the course of our thesis we discovered that the language server could not be started correctly with Mono. The problem already occurred when trying to start the OmniSharp language server component.

We took several measures to get even more specific details about the problem, after we found the specific point of failure.

- We have built a new try-catch around it - just in case. But the catch is not called because no exception was thrown.
- Because we use .NET Framework and not .NET Core, we could not debug step by step on macOS.
- Debugging the log output was not helpful since no log output was produced at all.
- We tried to start a very simple language server - without the Dafny project - and could not execute it correctly with `msbuild`.
- We tried different Mono versions: 6.6.0.166 (2019-08) and 6.8.0.105 (2020-02).
- We tested on macOS as well on Linux.

This made troubleshooting very difficult for us. Unfortunately, we could not find out why the OmniSharp server start-up does not work correctly with Mono. The OmniSharp community could not help us either [38].

While researching GitHub issues, it turned out that there are fundamental problems with Linux and Mac, as OmniSharp developers primarily develop and test on Windows [39].

In consultation with our supervisor, we then put the problem on hold after a certain amount of time had been used. Later on, Fabian Hauser noticed that instead of `msbuild` we would have to switch to `dotnet build`. Since `msbuild` is required for our Sonar scan and the limited time left, we decided not to take up the problem again in the last weeks of the project. However, this would certainly be a good approach to solve the problem.

## 6.6 Testing

This section provides a general overview of the testing. It is split into unit and integration tests. To read how to write tests or why we worked with interfaces for dependency injection, refer to the development document.

### 6.6.1 Unit Tests

Since all our core components are programmed against interfaces, testing them is not that hard. One can simply create a fake, mock or stub, implementing the interface, and use it for testing purposes.

#### Symbol Table Generation

To test the symbol table generation, dedicated Dafny files were written. First, the `DafnaTranslationUnit` is called to create a `DafnyProgram`. The `DafnyProgram` is then handed to the `SymbolTableGenerator`. The output is - once more - converted into a string representation. This outcome is then compared against an expectation file.

Technically, since the `TranslationUnit` is invoked as well, the test is not completely isolated. One would have to serialize the `DafnyProgram`, and use that one as a test input. However, to be able to quickly adjust the test file, this was not done.

Another flaw of this test is that the whole symbol table is tested at once. There is not a single test that checks if symbol `x` gets used 5 times - and checks only that. The string representation contains information about every symbol of the whole code at once, and the whole code is tested. While this is not best practice, it allowed for a very efficient testing. Even if a test fails, the string representation displays exactly why it failed, so there is not a big downside to this.

An example expectation file of this test is shown below:

```
[L0:C0] "_module" | P : [L0]_programRootNode | D : self | C : 2 | U : 0
[L1:C7] "MyClass" | P : [L0]_module | D : self | C : 4 | U : 0
[L3:C9] "field" | P : [L1]MyClass | D : self | C : | U : 2
[L5:C13] "add" | P : [L1]MyClass | D : self | C : 3 | U : 1
[L5:C17] "i" | P : [L5]add | D : self | C : | U : 1
[L5:C25] "j" | P : [L5]add | D : self | C : | U : 1
[L5:C42] "r" | P : [L5]add | D : self | C : | U : 2
[L6:C8] "r" | P : [L5]add | D : [L5]r | C : | U :
[L6:C13] "i" | P : [L5]add | D : [L5]i | C : | U :
[L6:C17] "j" | P : [L5]add | D : [L5]j | C : | U :
[L7:C15] "r" | P : [L5]add | D : [L5]r | C : | U :
[L10:C12] "aMethod" | P : [L1]MyClass | D : self | C : 1 | U : 0
[L10:C31] "this" | P : [L10]aMethod | D : [L1]MyClass | C : | U :
[L11:C13] "aLocalVar" | P : [L10]aMethod | D : self | C : | U : 2
[L12:C9] "field" | P : [L10]aMethod | D : [L3]field | C : | U :
[L12:C18] "aLocalVar" | P : [L10]aMethod | D : [L11]aLocalVar | C : | U :
[L13:C9] "aLocalVar" | P : [L10]aMethod | D : [L11]aLocalVar | C : | U :
[L13:C22] "add" | P : [L10]aMethod | D : [L5]add | C : | U :
[L13:C26] "field" | P : [L10]aMethod | D : [L3]field | C : | U :
[L16:C5] "_ctor" | P : [L1]MyClass | D : self | C : 0 | U : 0
[L0:C0] "_default" | P : [L0]_module | D : self | C : 0 | U : 0
```

The reader takes note that the test expectation is simplified. It is not exactly tested, if symbol `x` is used by symbol `y`. Instead, just some counts are checked. However, by testing if the parent is correct, and by testing the children count, we can be confident that the parent-child relationship is correct. The same applies

for the declaration-usage relation. You further note that there is no information given about base classes, parameter lists or such. But items like these are indirectly tested. If the base-class would not be attached properly, a symbols declaration would not be found. This would again be visible in the string representation.

### Symbol Table Navigation

To test the navigator and manager component, a fake symbol class was created. It allows to define the symbol position within the constructor. Afterwards, symbols can be attached to each other. This way, the top-down and bottom-up algorithms could be tested. An example is shown in listing 34.

---

```
1 public void BottomUpFirstChild()
2 {
3     SymbolInformationFake rootEntry =
4         new SymbolInformationFake(1, 0, 1, 0, 5, 0, defaultFile, "Parent");
5     SymbolInformationFake mySymbol =
6         new SymbolInformationFake(2, 0, 2, 0, 5, 0, defaultFile, "Child");
7     rootEntry.AddChild(mySymbol);
8     mySymbol.SetParent(rootEntry);
9     Predicate<ISymbolInformation> filter =(s => s.Name.Equals("Child"));
10    var symbol = nav.BottomUpFirst(mySymbol, filter);
11    Assert.AreEqual(mySymbol, symbol);
12 }
```

---

Listing 34: Navigator Unit Test

### Core Provider

To efficiently test the core providers, a `FakeSymbolTableManager` was created. It wraps a symbol table, in which a few fake symbols are attached to each other. Within the constructor of the fake-manager, it is told which symbol the manager will statically return. This way, the navigator component is not invoked.

This fake manager is used to test features like *Rename*, *GoToDefinition* or *HoverInformation*. It could just be injected to the providers, since the required interface `ISymbolTableManager` is implemented. By this construct, the correct assembly of *Rename* and *HoverInformation* could be tested. An example is shown in listing 35.

---

```
1 public void ReservedWord()
2 {
3     ISymbolTableManager manager =
4         new FakeSymbolManager(returnsDeclaration: false, returnsNull: false);
5     var provider = new RenameProvider(manager);
6     var result = provider.GetRenameChanges("method",
7         new Uri("file:///N:/u/1.1"), 2, 22);
8     Assert.IsTrue(provider.Outcome.Error, "error expected in rename-outcome");
9     Assert.AreEqual("method" + DafnyLanguageServer.Resources.LoggingMessages.
10         rename_reserved_word, provider.Outcome.Msg);
11 }
```

---

Listing 35: Core Provider Unit Test

### Core Provider not Depending on the Symbol Table

The features *Compile* and *CounterExample* are the only ones not depending on the symbol table. These were already tested in the prototype. Their tests were adjusted to meet the new implementation.

To test *CounterExample*, a custom `model.bvd` file can be injected into the provider. This way, predefined model files are analyzed instead of the model from the live Dafny code.

To test *Compile*, a `.dfy` file is defined in the test method. The test will again call the translation unit to create `TranslationResults`. Again, this is not optimal but practical. Out of the translation results, a file repository is created, which is then handed to the compilation provider. The final outcome can then be checked against an expectation.

### 6.6.2 Integration Tests

Unlike in the preceding semester thesis, integration tests were implemented using OmniSharp's language server client [40]. Each test starts a language server and a language client, then they connect to each other. Now, the client can send supported requests, such as the request for *CounterExample*. The result can be directly parsed into our `CounterExampleResults` data structure and be compared to the expectation. Thus, tests can be written easily and are very meaningful and highly relevant.

#### Dafny Test Files

Integration Tests usually run directly based on `.dfy` source files. Those test files need to be referenced from within the test. To keep the references organized, a dedicated project `TestCommons` was created. Each test project has access to these common items. Every test file is provided as a static variable and can thus be easily referenced.

---

```
1 public static readonly string cp_semiexpected = CreateTestfilePath("compile/  
   semi_expected_error.dfy");
```

---

Listing 36: Test File Reference

The class providing these references will also check if the test file actually exists, so that `FileNotFoundException` can be detected early.

#### String Converters

Many tests return results in complex data structures, such as `CounterExampleResults`. Comparing these against an expectation is not suitable, since many fields and lists have to be compared against each other.

To be able to easily compare the results against an expectation, a converter was written to translate the complex data structure into a simple list of strings. For example, each `CounterExample` will be converted into a unique string, containing all information about the `CounterExample`. All `CounterExamples` together are assembled within a list of strings. This way, they can be easily compared against each other. One could also overwrite `Equals(object other)`, but with the string representation, the developer has a direct visual feedback what caused the mismatch.

Since not only `CounterExamples`, but also other data structures such as `Diagnostic` were converted into lists of strings, the converters were held generic as far as possible. Listing 37 shows how this was realized. The method takes an enumerable of type `T` as an argument, and a converter which converts type `T` into a string. Each item in the enumerable is then selected in the converted variant.

---

```
1 private static List<string> GenericToStringList<T>(this IEnumerable<T> source,
    Func<T, string> converter)
2 {
3     return source?.Select(converter).ToList();
4 }
5
6 public static List<string> ToStringList(this List<CounterExample> source)
7 {
8     return GenericToStringList(source, ToCustomString);
9 }
10
11 public static string ToCustomString(this CounterExample ce)
12 {
13     if (ce == null)
14     {
15         return null;
16     }
17     string result = $"L{ce.Line} C{ce.Col}: ";
18     result = ce.Variables.Aggregate(result, (current, kvp) => current + $"{kvp
        .Key} = {kvp.Value}; ");
19     return result;
20 }
```

---

Listing 37: Converting *CounterExamples* to a List of Strings

Comparison of the results and the expectation is now very simple. The expectation can just be written by hand as follows in listing 38.

---

```
1 List<string> expectation = new List<string>()
2 {
3     "L3 C19: in1 = 2446; ",
4     "L9 C19: in2 = 891; "
5 };
```

---

Listing 38: Expectation

This is much more convenient than creating a custom `CounterExample`, which could then be used for comparison with an `Equals`-method. By taking advantage of the method `CollectionAssert.AreEqualent(expectation, actual)` from `nUnit`'s test framework, the two lists can be easily compared against each other [41].

### Test Architecture

Since every integration test starts the client and the server at first, as well as disposes them at the end, this functionality could be well extracted into a separate base class. This class is called `IntegrationTestBase` and just contains two methods, `Setup` and `TearDown`. These methods could be directly annotated with the proper `nUnit` tags like `[SetUp]` and `[TearDown]`, so that every test will at first setup the client-server infrastructure, and tear it down after the test has been completed.

A second base class exists for each feature. For testing compilation, this class is named `CompileBase` as an example. It inherits from the `IntegrationTestBase` class and provides the member `CompilerResults`, as well as two methods `RunCompilation(string file, string[] args)` and `VerifyResults(string expectation)`. The implementation of `RunCompilation` is shown in listing 39.

---

```
1 protected void RunCompilation(string testFile, string[] args)
2 {
3     CompilerParams compilerParams = new CompilerParams
4     {
5         FileToCompile = testFile,
6         CompilationArguments = args
7     };
8     Client.TextDocument.DidOpen(testFile, Resources.ConfigurationStrings.
9         DafnyFileEnding);
10    compilerResults = Client.SendRequest<CompilerResults>(compileKeyword,
11        compilerParams, CancellationSource.Token).Result;
12 }
```

---

Listing 39: Running a Compilation Integration Test

The test class itself inherits from its feature-specific base class. The tests itself are very simple. For example, if we want to test if the compiler reports a missing semicolon, we could create a test class `public class SyntaxErrors : CompileBase`. Note that we inherit from our case-specific base class. Thus, the methods `RunCompilation` and `Verify` are at our disposal. That means that the test is as simple as follows:

---

```
1 [Test]
2 public void FailureSyntaxErrorSemiExpected()
3 {
4     RunCompilation(Files.cp_semiexpected);
5     VerifyResults("Semicolon expected in line 7.");
6 }
```

---

Listing 40: Sample Test for Missing Semicolon

As you can see, the test contains only two lines of code. The first is stating the test file, the second one the test result expectation. Tests for other features look quite similar, just the expectation format may be different. Thus, the integration test architecture could be created in a way so that the creation of tests is extremely simple as you see in figure 69. The code is kept very clean and contains no duplicated parts. Tests can easily be organized into classes – considering compilation, this could for example be the separation into logical errors, syntax errors, wrong file types and so on.



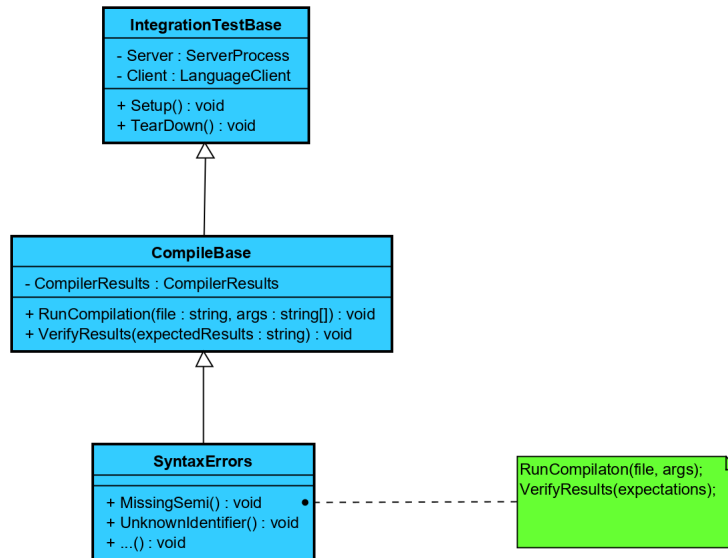


Figure 69: Test Architecture on the Basis of Compilation

## 6.7 Usability Test

Since we have been developing and using our plugin for many months, it is sometimes difficult to differentiate the results.

Therefore we have designed a usability test and found a test person to perform this test [42]. The test person, Remo Herzog, was chosen because he is programming in an environment based on VSCode and therefore he is well integrated into the corresponding VSCode universe.

The goal of this test was primarily to find out if the features we implemented were implemented as a VSCode Plugin user would expect them to be, and which features were expected to be different in terms of display and triggering.

The general usefulness would also be discussed and feedback for further improvements was collected. These improvements are discussed in this section.

### 6.7.1 Code Documentation for Classes and Methods

If you drive over symbols and the hover information is displayed, and if the *AutoCompletion* is listing symbols, a code documentation is missing in the corresponding info boxes. It would be desirable to have a code documentation for methods and classes, which is displayed additionally with the mentioned features.

This feature could be realized with the new symbol table. The approach would be the following: Through the feature *CodeLens* all declarations of methods and classes are already known. For each of these symbols the correct position in the Dafny source code could be read out and with a regular expression found code documentation comments could be read out. The documentation text, read out by the pattern matching, can then be stored in the corresponding symbol as code documentation. This information can then be sent to the client via the LSP and displayed, for example in the case of *AutoCompletion* or *HoverInformation*, as a `documentation` element [23].

### 6.7.2 Cleaner HoverInformation

The information which is displayed in a symbol during a hover event was felt to be somewhat confusing. Sometimes the terms like `kind` were not clear. By using markdown instead of plaintext, *HoverInformation* could be improved significantly, as shown in figure 70

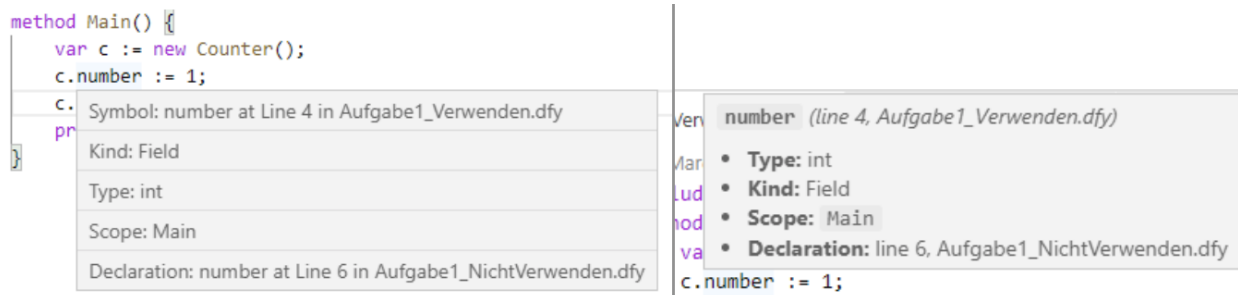


Figure 70: Hover Before (Left) and Cleaner Version (Right)

### 6.7.3 Automatic Triggering of AutoCompletion

If a dot is typed while typing, *AutoCompletion* suggestions are automatically expected. You should not have to press `CTRL+Space` as trigger. This issue was anyway pending and was implemented shortly after the usability test.

In addition, the suggested methods lacked to inform about the supported parameters. With regard to the parameters, this has already been discussed in section 6.4.1.

### 6.7.4 Error Message Without Brackets

Error messages are reporting the code symbol at which the error occurs. For example, when the error starts at the body of a method, the error message contains `Assertion Violation at [ ] in line 5..` The square brackets were confusing the test person. Thus it was decided to leave them out, and instead show the code token on a new line. This should be less confusing. The issue is shown in figure 71

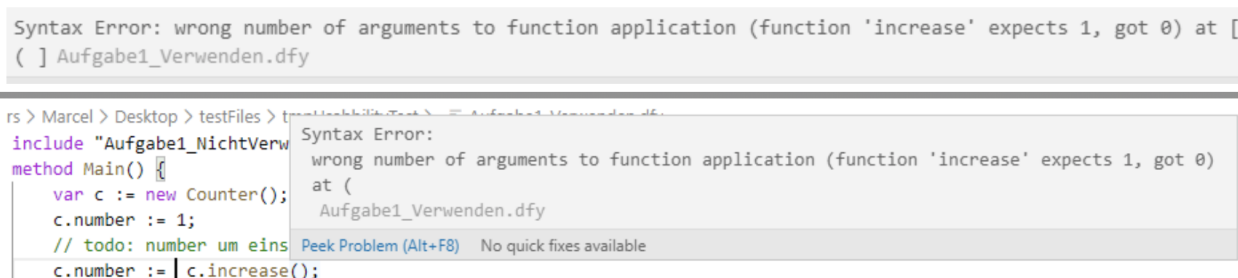


Figure 71: Error Message Before (Top) and After (Bottom)

### 6.7.5 Compile Improvements

Some plugins have a small green play icon in the upper right corner of the IDE for compilation, which can be pressed as shown in figure 72. The test person would have wished such a play button [42].



Figure 72: Play Button to Run Python Code

It is possible to also add this icon for Dafny. If the button is clicked, *CompileAndRun* is triggered. Similar to the UI elements for the Dafny status bar, a button can be created by the VSCode API [23]. It can be assigned an icon and a click action. Due to the lack of time, the feature could not be implemented anymore, though.

When running the Dafny program, a lot of text messages are displayed, which distract from the relevant output. To shorten the message length, full paths could be omitted and instead, just file names are shown. This is illustrated in figure 73.

```

1  include "Aufgabe1_NichtVerwenden.dfy"
2  method Main() {
3      var c := new Counter();
4      c.number := 1;
5      print(c.number);
6  }
    
```

---

DEBUG CONSOLE   PROBLEMS 4   OUTPUT   TERMINAL   1: Run c:\Users\Marcel\

Windows PowerShell  
 Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

Lernen Sie das neue plattformübergreifende PowerShell kennen - <https://aka.ms/pscore6>

```

PS C:\Users\Marcel\Desktop\BA\dafny-language-server\Test\LanguageServerTest\DafnyFiles\codeLens> & "
c:\Users\Marcel\Desktop\BA\dafny-vscode-plugin\testFiles\tmpUsabilityTest\Aufgabe1_Verwenden.exe"
1
PS C:\Users\Marcel\Desktop\BA\dafny-language-server\Test\LanguageServerTest\DafnyFiles\codeLens>
    
```

Figure 73: Console Output That Could be Improved

## 6.8 Continuous Integration (CI)

Optimizing the CI process is an important part of our thesis. Since we expect that our language server as well as the Dafny client plugin will be further developed by other developers, the CI is a primary component of automated quality assurance.

This includes that the CI pipeline will fail if certain quality attributes are not met. These include, in particular, successful completion of automated tests, static code analysis and formatting control of the TypeScript code.

More details on the quality aspects are described in chapter 9 – Project Management. This section describes how the planned improvements for the CI process from chapter 4.10 were implemented.

### 6.8.1 SonarQube

According to our research, a major problem was that the scanner for SonarQube can only analyze one language at a time [43]. This means, that the TypeScript code in the client and the C# code in the server cannot be analyzed simultaneously. Furthermore, in the pre-existing Dafny project, single Java files appear, too. This led to further conflicts in the Sonar analysis [3].

As a simple solution, we decided to separate the client (VSCode plugin) and server (Dafny Language Server) into two separate git repositories. This not only simplifies the CI process but also ensures a generally better and clearer separation.

As a result, the client could still be easily analyzed with the previous Sonar scanner. Regarding the server, a special Sonar scanner for *MSBuild* had to be installed, which publishes the analysis in a dedicated Sonar-Cloud project [27]. The available statistics, like the finding of bugs as shown in figure 74, are very helpful for code reviews.

As a little extra, not only our Dafny language server part is analyzed by Sonar, but the whole Dafny project. From the bugs, vulnerabilities, code duplications and code smells revealed by this analysis, the whole GitHub team working on the Dafny project can benefit.

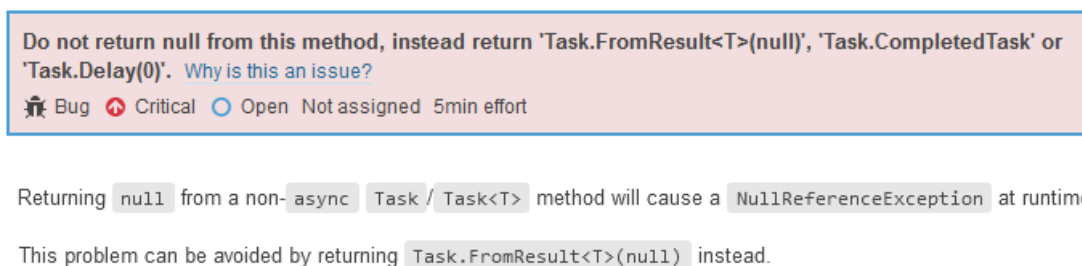


Figure 74: Example of a Useful Sonar Finding

Unfortunately, the code coverage by tests is not analyzed. Searching for an alternative, *OpenCover* was found as a very common tool for code coverage analysis in C#. Unfortunately, it only runs under Windows [44]. The CI server bases on Linux, though. During our research we came across *monocov* [45]. This tool would run under Linux and analyze .NET Framework projects. Unfortunately, this project was archived and has not been maintained for almost 10 years [45].

Since we would not gain much value with Sonar code coverage, we decided not to pursue this approach any further. The coverage information is provided by the ReSharper extension *dotCover* [46] to the developers.

### 6.8.2 Client End-to-End Tests

The end-to-end tests base on a lot of dependencies, such as a headless instance of Visual Studio Code. In consultation with our supervisors, we have removed these tests from the client project and replaced them with own specially written integration tests on the server side. This can be justified with the client only containing a minimal amount of logic that is required for the visual representation. Any other logic was moved to the server.

### 6.8.3 Static Program Analysis and Formatting for TypeScript

During development it became apparent that it would be useful to check the correct formatting for the TypeScript code from the client in the CI. Various tools are used locally, which automatically format the syntax correctly when saved. These tools are described in the developer documentation [27]. However, if someone does not install these tools, and the code is not formatted according to the definition, the CI pipeline will fail.

Prettier was selected for this function [27]. Alternatively, ESLint would have been an option [47]. This would also have automatically integrated a static code analysis in the CI. However, we decided to use a combination of SonarLint and Prettier, so that the local analysis tool is matched to SonarQube [27].

### 6.8.4 Docker

For an easier testability of the CI, we installed Docker locally. This allowed us to resolve CI issues locally and platform-independently (through the Docker Client) in case of problems. More details are stated in the developer documentation [27]. The documentation found there helps future developers to easily and efficiently test changes to the docker container locally.

## 7 Results

As described in the objectives, our work has three basic stakeholders.

On the one hand, there are the students of the HSR who want to get familiar with Dafny. For these students, a useful plugin should be offered to make the development with Dafny easier.

On the other hand, there are the developers who want to extend the language server in the future. This category also includes developers who want to integrate other IDE's by additional clients.

Last but not least, anyone willing to program with Dafny may download the plugin from the VSCode marketplace. Since we wanted to make a contribution to the Dafny community with our project, it was always a major concern for us to achieve a quality level so that it can be published.

This chapter describes the results achieved and further prospects for each stakeholder.

### 7.1 Features for Dafny Developers

In this section, the achieved feature set of the project is described and critically reflected. The current version of the plugin supports the following functionality:

1. *SyntaxHighlighting*
2. *CodeVerification*
3. *Compilation*
4. *CounterExample*
5. *HoverInformation*
6. *GoToDefinition*
7. *Rename*
8. *CodeLens*
9. *AutoCompletion*

#### 7.1.1 Syntax Highlighting

Syntax highlighting is realized by a given Dafny grammar file. The file contains regex expressions defining the highlights. It is provided by Dafny [22] and could simply be downloaded. The feature was already implemented in the prototype, thus no further actions aside updating the grammar file had to be done. The following screenshot shows how syntax highlighting looks inside Visual Studio Code.

```
1  method MoreAndLess(baseNumber: int, toAddOrSubtract: int)
2      returns (more: int, less: int)
3      requires toAddOrSubtract > 0
4      ensures less < baseNumber < more
5  {
6      more := baseNumber + toAddOrSubtract; //This will be more than baseNumber
7      less := baseNumber - toAddOrSubtract; //This will be less than baseNumber
8  }
```

Figure 75: Syntax Highlighting

As you can see, keywords like *method*, *returns*, *requires* and *ensures* are marked in purple. Types like `int` are printed in blue and comments become green. Symbols, such as classes and methods, are displayed in a brownish color. Just these simple rules increase the readability significantly.

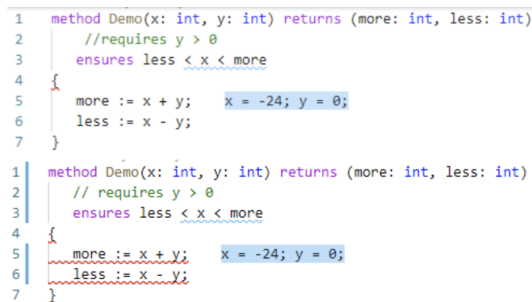
### 7.1.2 Verification

Verification was already implemented in the prototype, too. At the start of the project, the feature held some major flaws.

- It only reported logical errors.
- Syntax errors were not announced.
- Warnings were ignored.

The code just invoked the `DafnyTranslationUnit`, taken over by the pre-existing project.

Verification was reworked completely during this project. First of all, the Dafny implementation itself was analyzed to get a better understanding of how Dafny compiles its code and how errors are reported. Those errors could finally directly be extracted out of the Dafny error reporting engine. Thus, the user is now informed not only about logical errors, but also about syntax errors. Furthermore, warnings and information diagnostics are now also reported. Reporting warnings is something the pre-existing project did not do and was actually already issued on the official Dafny git repository [48]. With the symbol table, it is now also possible to underline complete code blocks, instead of just single symbols which were barely visible at the beginning of the project as you can see in figure 76. This shows well how the quality of the language server could be improved.



```
1 method Demo(x: int, y: int) returns (more: int, less: int)
2 //requires y > 0
3 ensures less < x < more
4 {
5   more := x + y;   x = -24; y = 0;
6   less := x - y;
7 }

1 method Demo(x: int, y: int) returns (more: int, less: int)
2 // requires y > 0
3 ensures less < x < more
4 {
5   more := x + y;   x = -24; y = 0;
6   less := x - y;
7 }
```

Figure 76: Underlining Before (Top) and After (Bottom)

The feature directly invokes Dafny's compile engine, thus works quite solidly and scales automatically with future Dafny features.

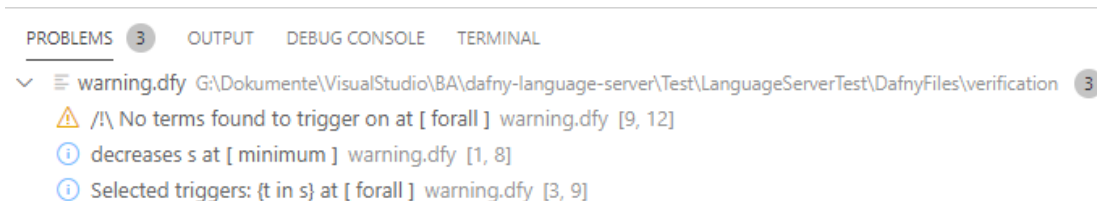


Figure 77: Reporting of Warnings and Information Objects

### 7.1.3 Compile

The compile feature is strongly connected to the verification process. Prior to compilation, the whole Dafny project has to be verified anyway. Thus, since verification yields a precompiled `DafnyProgram`, the buffered result can be used to invoke the Dafny compiler. This makes the compilation process very snappy and responsive.

If the code contains errors, the verification process already failed and compilation can instantly be denied. However, if the code is fine, the precompiled `DafnyProgram` just has to be translated, which can be done quickly. The user also has the option to apply custom compilation arguments. These can be directly set within VSCode.

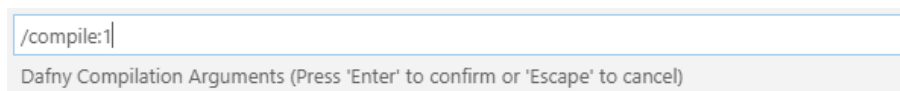


Figure 78: Custom Compilation Arguments

Custom arguments are directly handed to the Dafny options parser and are applied within the Dafny engine. Since compilation uses the precompiled Dafny program, compilation arguments affecting the verification process have no effect. This is something that could be resolved by just restarting the verification process if custom arguments were given.

In the prototype, compilation was implemented by just starting a sub-process, launching `Dafny.exe` with custom arguments given. To obtain compilation results, the console output of the sub-process was parsed and reported to the user. That solution was obviously not integrated at all. The current implementation improved that significantly, is now completely integrated, both, in terms of invocation and result reporting.

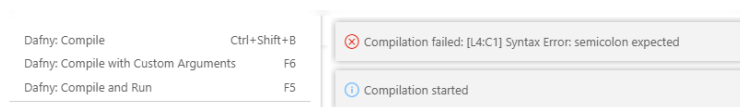


Figure 79: Compilation Context Menu and User Feedback

If the user chooses the option to *CompileAndRun*, the actual launch of the executable takes quite a long time - up to about 20 seconds. However, this delay is not related to the language server or the compilation process. If the executable is launched manually within another console outside of Visual Studio Code, it also takes long until the program starts. The effect just occurs the first time an executable is launched. The reason for that could not be investigated.

### 7.1.4 CounterExample

Providing counter examples was already possible in the prototype. A major flaw was that the representation of the counter examples was quite complex and not intuitively readable. Thus, it was a goal to ease their representation.

For this, the related `model.bvd` file was studied. It is quite a cryptic file and getting a comprehensive understanding of it would be very complex. However, an `inital state` was located which seemed exactly that part of information, that the user is interested in. Thus, unlike previous in versions, only that `initial state` is considered in counter examples. Furthermore, any unreadable representations such as `**myVar` or `TU!Val23` were omitted. To allow the user to catch the information at first sight, obsolete brackets are



also removed, and the minus sign is directly moved to the number. The expression  $(( - 23 ))$  is therefore reformatted into  $-23$ , making the term much more perceivable.

```

1 | method Demo(x: int, y: int) returns (more: int, less: int)
2 |     //requires y > 0
3 |     ensures less < x < more
4 | {
5 |     more := x + y; less = (**less#0); more = ((- 24))'1; x = ((- 24)); y = 0;
6 |     less := x - y; less = ((- 24))'2; more = ((- 24))'1; x = ((- 24)); y = 0;
7 | }

```

---

```

1 | method Demo(x: int, y: int) returns (more: int, less: int)
2 |     //requires y > 0
3 |     ensures less < x < more
4 | {
5 |     more := x + y;     x = -24; y = 0;
6 |     less := x - y;
7 | }

```

Figure 80: CounterExample Representation Before (Top) and After (Bottom)

Figure 80 shows a comparison of the *CounterExample* feature between the initial state and at the end of the project. The representation is much cleaner and easier to immerse. Also note that more room is given so that it does not have a clumsy effect on the user's eye.

This feature was also improved on the client side. The user has the option to configure the color scheme of the counter example representation, if he does not like the default colors. The default colors are chosen with respect to the user's base color theme of VSCode (dark or light mode). As in figure 80 the light theme is shown, the following figure 81 shows the dark mode colors.

```

1 | method Demo(x: int, y: int) returns (more: int, less: int)
2 |     // requires y > 0
3 |     ensures less < x < more
4 | {
5 |     more := x + y;     x = -24; y = 0;
6 |     less := x - y;
7 | }

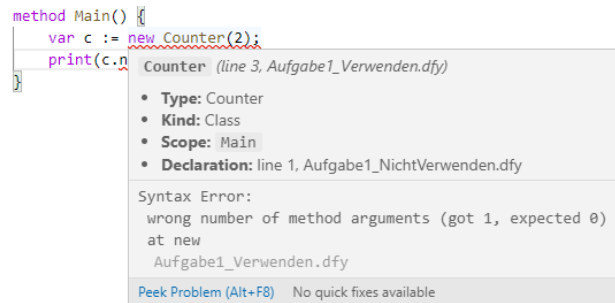
```

Figure 81: CounterExample Dark Theme

The counter example will correctly adjust if the user continues to work on the code and vanish, once the problem is resolved. If the user switches between windows, the visibility state is buffered, so that once the user switches back to the original window, the counter example will be shown again.

### 7.1.5 HoverInformation

*HoverInformation* displays a set of information, whenever the user hovers with the mouse cursor over a code symbol. The feature itself was very simple to implement, since it receives all necessary information from the symbol table. It was additionally implemented as an exemplary feature, to show off how easy some LSP functionality can now be implemented using the new symbol table. Similar to *HoverInformation*, other features like text highlighting can be added as well in the future, which is discussed in chapter 8. *HoverInformation* may not provide much useful information to the user, but it is still a nice gimmick. For example, in figure 82 the user can actually find out, where his `field` is declared if he is unsure.

Figure 82: *HoverInformation* Example

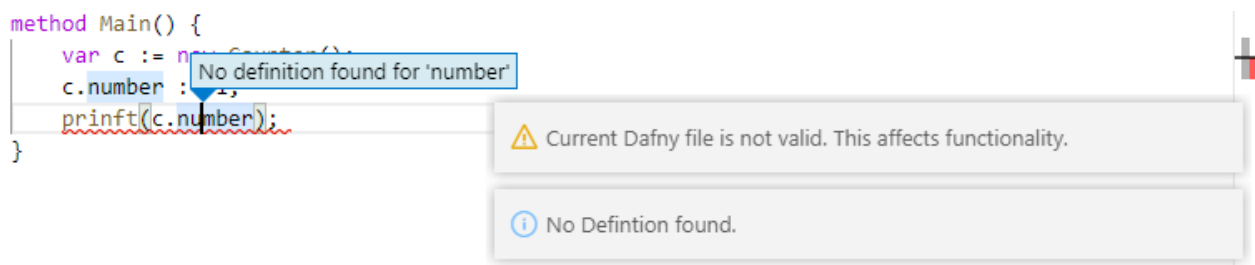
### 7.1.6 GoToDefinition

Compared to the prototype, *GoToDefinition* has been significantly improved. Prior to this bachelor thesis, the feature just scanned all code for the first name-match and reported it as the definition. This did neither work if multiple symbols with the same name occurred, nor if the declaration was placed after the first usage in the code. Both cases are now handled well by the symbol table engine.

Runtime could also be improved, since the symbol table generator scans for possible declarations scope by scope, and does not just iterate over all symbols. Last but not least, in the prototype, the cursor had to be at the beginning of a symbol to recognize the symbol at all. This major flaw has of course been overcome. The cursor can now be at any character within a symbol.

Technically, the feature is misnamed. A better name would actually be *GoToDeclaration*, since it jumps to the symbol declaration, not definition. LSP would even offer a dedicated handler for this, `textDocument/declaration` [9]. However, we kept the name to maintain the consistency of our work. But perhaps a rename would be appropriate.

To increase the user experience with this feature, notifications are sent to the user as shown in figure 83 if an error occurred or the requested symbol was already a declaration.

Figure 83: *GoToDefinition* Error Reporting

Jumps can also be done across included files. If the targeted file is not opened, VSCode will just open it inside the workspace. It is also possible to go to the definition of `this`-Expressions, which will just jump to the class definition. The feature works also across code that is not within a block statement, for example for expressions occurring inside an `ensures`-clause.

### 7.1.7 Rename

*Rename* was newly added to the language server. It is a feature that could only be realized, since the symbol table provides all necessary information. If the user wants to execute a rename, all dependent symbols are deduced correctly as you see in figure 84.

```
1 | method Demo(x: int, y: int) returns (more: int, less: int)
2 |   requires y > 0
3 |   ensures less < x < more
4 | {
5 |   more := x + y;
6 |   less := x - y;
7 | }
```

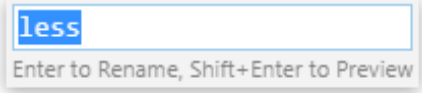


Figure 84: Renaming of a Symbol

Providing as much user experience as possible for this feature was embraced, even if it is relatively simple. In particular, this means if there is no renamable symbol at the cursor, the user is informed by a notification. The algorithm will also check whether the new symbol name is valid. This means, it must not start with an underscore or number, and it must not be any reserved Dafny word. Also, we limited the allowed new name to alphanumerical characters, although wild Unicode names would be allowed by Dafny.

This could even be driven further. For example, it could be checked whether a symbol with the same scope already exists. Functionality for this is already available by the symbol table, since something similar is needed by autocompletion. That is also something that could easily be added.

Since Dafny reserved words may change in the future, the wordlist is configurable for the end-user by adjusting a config file. The process targets more advanced users and is described in the readme file.

Because the symbol table reports occurrences across imported files as well, the rename feature works well for included files. However, if a file gets included elsewhere, it does not know about it. This is, since no global symbol table is currently used for the entire workspace. This issue and possible solutions were described in section 5.4.3.

### 7.1.8 CodeLens

*CodeLens* is an excellent way for a developer to see which classes and methods were used how often - and where. In order to be able to assign the references reliably and also to consider the references across included files, the symbol table was necessary.

As shown in figure 85, not only the references to the symbols are counted, but also dead code is lifted by a special text.

```

    2 references to ClassB
11  class ClassB {
12  |   constructor () { }
    3 references to myMethod
13  |   method myMethod() { /* do something */ }
14  }
15
    1 reference to ClassC
16  class ClassC {
17  |   var ABC: int;
18  |   constructor () { }
    Not used yet. Can you remove myMethod?
19  |   method myMethod() { /* do something */ }
20  }
..
    
```

 Figure 85: *CodeLens* Shows the Number of Uses

If the user clicks on one of the reference texts, a popup window opens, as figure 86 shows. On the right hand side of the popup window all references are listed. If the user clicks on them, the corresponding code location in the right file opens on the left side - at the location within the popup. This is very convenient for developers, because they no longer have to jump back and forth between files and they do not have to scroll unnecessarily.

```

    2 references to ClassB
11  class ClassB {
    
```

Aufgabe2.dfy C:\Users\Marcel\Desktop\testFiles\tmpUsabilityTest - Locations (2) ✕

```

24  */
25  method Main() {
26  |   var myNumber := 1+2;
27  |   var myClassA := new ClassA();
28  |   myClassA.myMethod();
29  |   var myClassB1 := new ClassB();
30  |   var myClassB2 := new ClassB();
31  |   myClassB1.myMethod();
32  |   myClassB2.myMethod();
33  |   myClassB2.myMethod();
34
12  |   constructor () { }
    3 references to myMethod
13  |   method myMethod() { /* do something */ }
14  }
    
```

 Figure 86: *CodeLens* Shows References in Pop-up

Just like *Rename*, the feature does not work for code that includes the current file due to the missing global symbol table as described in section 5.4.3.

### 7.1.9 Automatic Code Completion

*AutoCompletion* lists the available symbols in the current context on request. Compared to our previous thesis, we could significantly improve this function. Instead of displaying all found symbols in the opened file as before, we can now use the new symbol table to consider the current scope for suggestions.

Our *AutoCompletion* supports three basic types of proposals:

- Completion after a dot was written.
- Completion after `new` was written.
- General suggestions.

If a dot was written, only declarations of the class in front of the dot are suggested. If a `new` was written, only available classes are listed. When none of the above is the case, all available symbols in the current scope will be displayed. The three cases are shown in figure 87.

```

1 reference to A
class A {
  var inA : int;
  constructor() {}
}

Not used yet. Can you remove B?
class B {
  var inB : int;
  constructor() {}
}

Not used yet. Can you remove foo?
method foo() {
  var a := new A();
  var b : int;

```



The screenshot shows three examples of auto-completion in an IDE:

- Variable Declaration:** The code is `var b : int;`. The completion list shows `A`, `a`, `B`, `b`, and `foo`. The selected item is `a`. The tooltip for `a` shows `Kind: Variable` and `Parent: foo`.
- 'new' Statement:** The code is `var c := new`. The completion list shows `A` and `B`. The selected item is `A`. The tooltip for `A` shows `Kind: Class` and `Parent: _module`.
- Field Access:** The code is `var d := a.`. The completion list shows `inA`. The selected item is `inA`. The tooltip for `inA` shows `Kind: Field` and `Parent: A`.

Figure 87: AutoCompletion

### Additional Usability Sugar

As a little extra to increase the user experience, when inserting a suggested function or method after a dot or a class after a `new`, opening and closing brackets are automatically added.

However, the parameter list of methods is not yet proposed. Technically it would be possible, as described in chapter 6.4.1 – AutoCompletion. This would further enhance the user-friendliness as proven in the usability test.

## 7.2 Simplicity of Further Development

This section describes architectural improvements, which increase maintainability and facilitate future extensions of the project.

### 7.2.1 Symbol Table

Within this project, a new symbol table was implemented. The prototype was already working with a primitive approach of a symbol table, that worked only with strings. This is no comparison to the new symbol table, which works with proper objects. Related information, such as the parent symbol, is now directly accessible. Any string parsing could be completely omitted compared to the prototype. This leads to better performance and, above all, to reliable symbol references.

All of the previous features, with the exception of *AutoCompletion*, do not contain much logic themselves. They just request information from the symbol table and report it back to the client. The symbol table allows the implementation of new features, such as *CodeHighlight* and even *AutoFormat* with ease. This is a major gain for the future of the language server.

Because information about the symbols can be made available already prepared, the feature development has been greatly simplified. For example, before we had the symbol table, a feature like *CodeLens* required us to find and count all references for a symbol in the following way, as shown in listing 41. Comments have been added to understand the code without context. The code has been simplified for illustration purposes.

---

```
1 // for each symbol in the current opened Dafny file...
2 foreach (var symbol in fileSymboltable.GetFullList())
3 {
4   var symbolReferenceCounter = 0;
5   // ...search through all symbols in all buffered files
6   // to see if the current symbol is used.
7   foreach (var fileBuffers in _bufferManager.GetAllFiles().Values)
8   {
9     foreach (var symbolInformation in fileBuffers.Symboltable.GetFullList())
10    {
11      // if the name matches, count this as an usage.
12      if (symbolInformation.Name == symbol.Name)
13        symbolReferenceCounter++;
14    }
15  }
16  // create CodeLens with symbolReferenceCounter
17 }
```

---

Listing 41: Example for *CodeLens* in the Prototype

Newly this works via a property as a one-liner as shown in listing 42. The symbol table simplifies the development enormously.

```
1 foreach (var symbol in _manager.GetAllSymbolDeclarations())
2 {
3   var symbolReferenceCounter = symbol.Usages.Count;
4   // create CodeLens with symbolReferenceCounter
5 }
```

Listing 42: Example for *CodeLens* With the new Symbol Table

While the features are implemented quite robust, they only work as long as the symbol table provides proper information. Whenever the symbol table fails, the underlying features will also produce nonsense. Thus, a correct symbol table is crucial for correct functionality.

Dafny is a programming language offering a lot of features. Aside common object oriented features, also functional programmatic features are present. Figure 88 shows all classes occurring inside the Dafny AST. The writers are well aware, that the text in this figure is too small to be read. The figure should indicate the big amount of AST-elements present in Dafny. Many of them are not just inheriting from `Expression` or `Statement`, but from individual base classes. Thus, implementing the visitor for all of them turned out to be massively time consuming. Since the bachelor thesis is bounded by a limited time frame, it was necessary to limit the amount of Dafny language features we support. We decided to lay our focus on the symbols needed by the Dafny tutorial, which is done within the software engineering at HSR. This way, the basic concepts of Dafny are supported. We could support about 50% of the available AST-nodes. However, as soon as the user starts to use more advanced language features, a symbol may be introduced within a scope that is not visited by the visitor. If the user is using that symbol later on inside a common method body, the symbol table generator will be unable to locate the symbol's declaration and thus fail. An overview of the supported expressions and statements was given in section 4.7.

It is subject of further development to complete the visitor. Primarily, it lacks support for generics and custom datatypes. Finishing the visitor should have very high priority in the future development of this project, to provide the best possible user experience.

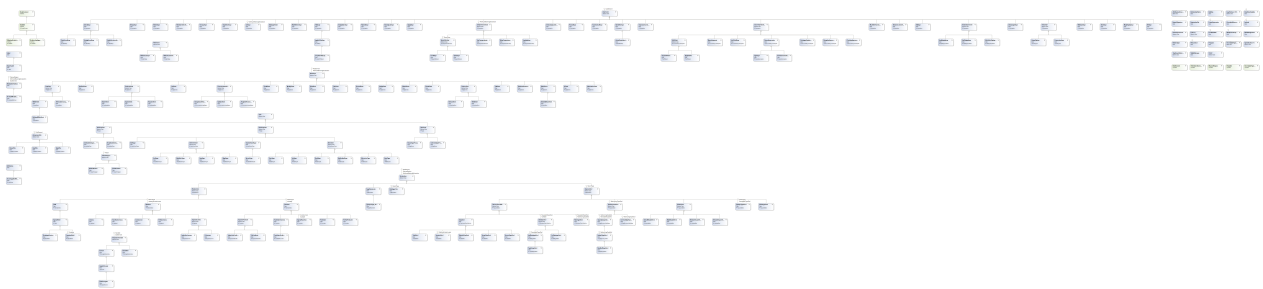


Figure 88: All Dafny AST Classes

### 7.2.2 Server Architectural Improvements

At the beginning of the thesis, the server's dependency graph was quite a mess. During development, the code was constantly cleaned up and dependencies were resolved.

While the project is quite large and has a lot of dependencies, they could still be well organized. As seen in figure 89, all dependency are now pointing strictly downwards only, just as it should be. Within the picture, dependencies to the `Utilities` layer are excluded for more overview.

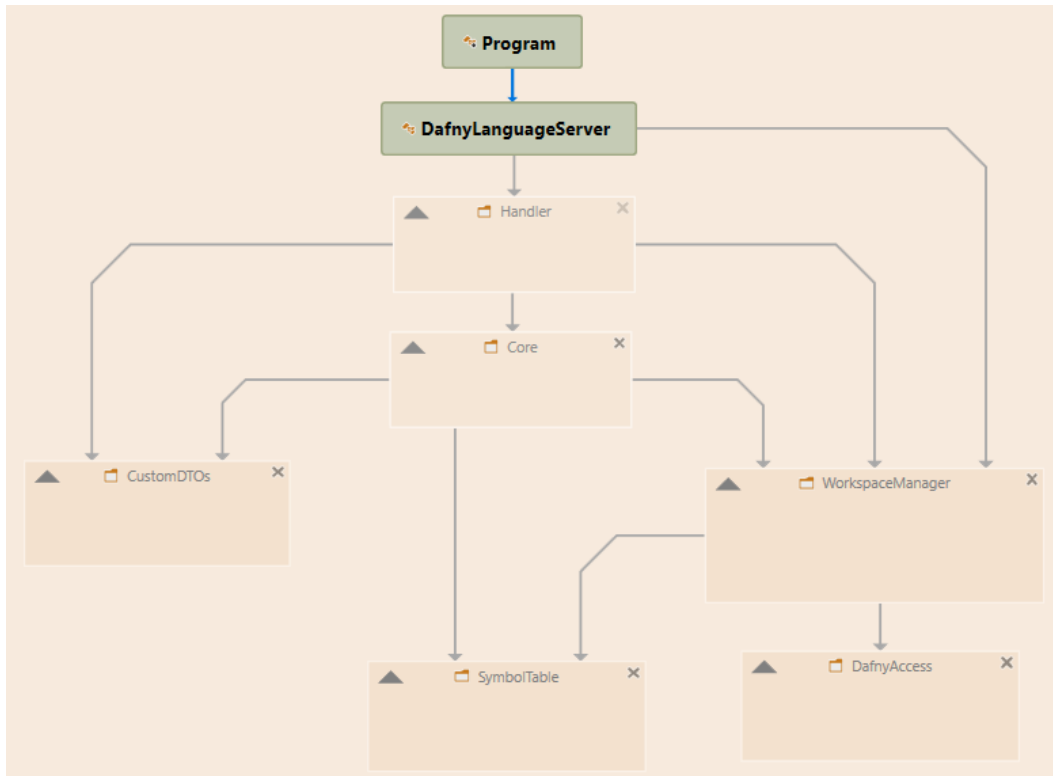


Figure 89: Server Package Dependencies Generated by Visual Studio

Aside from the broad architectural layout, many small refactorings were made to keep the code clean. This includes the creation of smaller classes and smaller methods with single responsibilities. A good example for this is the config initialization, which was one single large class at first. After refactoring, it was split into small, well organized sub-classes.

Any important classes, such as handlers, providers or classes related to the symbol table are programmed against interfaces. This allows for optimal testing, for example by implementing the same interface with a fake instance.

### 7.2.3 Client Architectural Improvements

The client was completely restructured. Isolated, small, preferably encapsulated components were programmed. Components are now no longer directly dependent on each other but program against interfaces. Modules were created and these export only the most necessary interfaces and components to the outside. If complex has be implemented for the client at a later date due to new features, this encapsulation by interfaces is ideal for writing tests.

This resulted in less encapsulation and higher coherence. These two motivations also encouraged the complete isolation of VSCode modules. We isolated the VSCode components in a separate module if possible. This minimizes the effort to migrate the plugin to other IDEs.



## 7.3 Deployment

The final plugin has been published to the Visual Studio Code marketplace as a preview version [49]. The installation is very easy and runs automatically. Any Windows user can install it – try it out and leave feedback.

This chapter describes further aspects related to the release.

### 7.3.1 Pull Request to Dafny

Our Dafny language server has reached a point where it technically can be merged back into the original Dafny project [50]. Unfortunately, they released a new version shortly before the end of the thesis. Thus, a rebase is necessary first. Since Dafny changed the underlying .NET Framework version, the rebase was executed as a proof-of-concept, but was not carried out to avoid bugs.

There is also still a problem with platform independence as described in in chapter 6.5 – Mono Support for macOS and Linux. Although this was a major goal, it could not be achieved. Before the pull request, those issues may be resolved.

### 7.3.2 Publication to the VSCode Marketplace

We merged our GitLab repository to a dedicated GitHub repository. From there, it is uploaded to the VSCode marketplace during the CI process, which is shown in figure 90. This process uses the GitHub pipeline `release-marketplace`, which was adapted by Fabian Hauser [51].

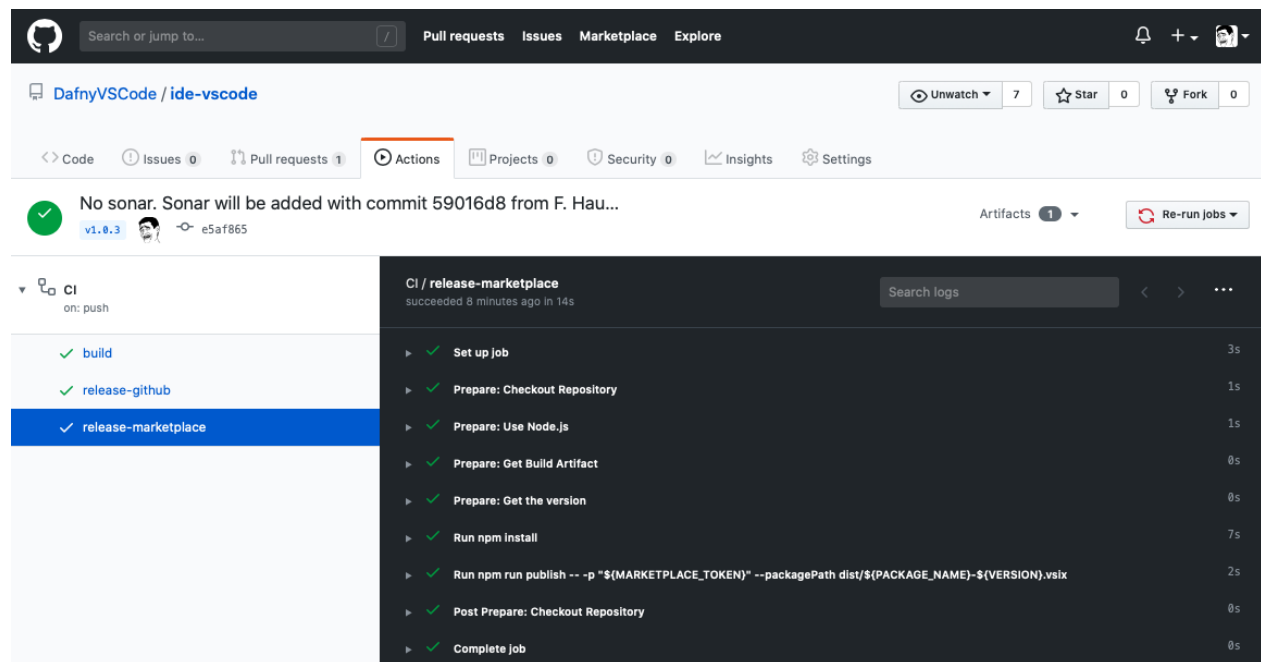


Figure 90: GitHub CI

Once the pull request to the Dafny project to integrate the Dafny language server is complete, the preview released plugin can completely replace the original plugin.

### 7.3.3 Download of the Dafny Language Server

In the pre-existing project, Dafny was downloaded from GitHub [32]. As discussed in section 6.1.4 – Download of the Dafny Language Server, we have written a custom interface for this original process.

Thanks to the interface, we can currently download the Dafny language server from a private server [52]. Once the pull request to Dafny is accepted, the download process can then be easily adjusted to download everything from Dafny directly.

We decided on this simple variant in order to keep the effort for this temporary solution as low as possible. Alternatively, this process could have been automated with the GitLab CI using the GitLab API [53] to download the CI artifacts. The artifacts could also be made available through the CI process as a static website with GitLab Pages [54].

### 7.3.4 Easy Installation

As soon as the user opens a Dafny file, he receives a message as shown in figure 91, that appropriate extensions are available for Dafny files in the extension marketplace.

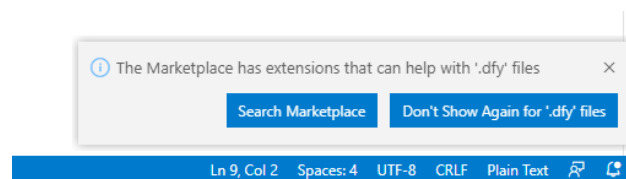


Figure 91: Message "Marketplace has Extensions for Dafny Files"

In the marketplace we search for extensions for Dafny and find our plugin as shown in figure 92.

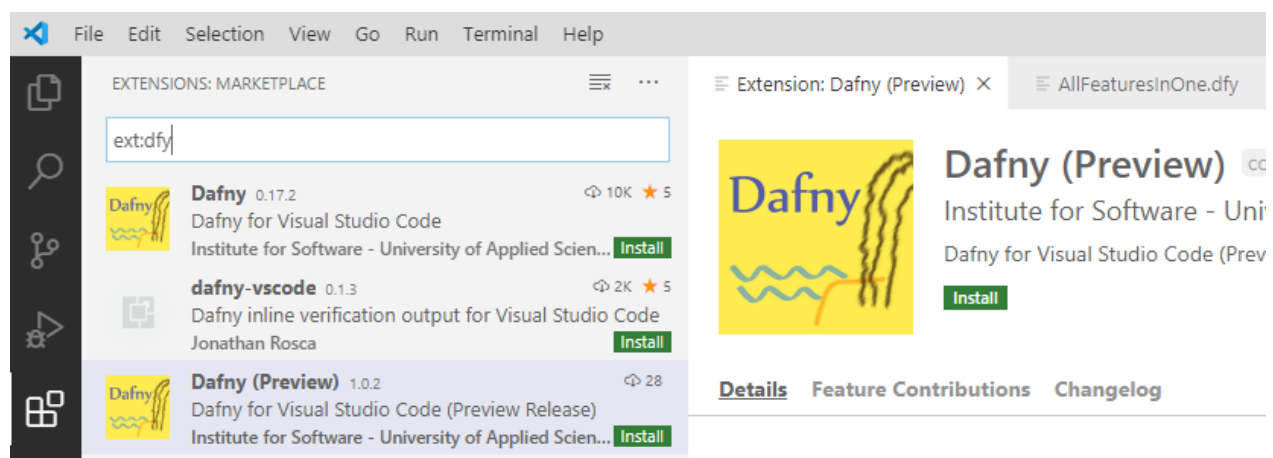


Figure 92: Searching Dafny Extensions in the Marketplace

Besides the plugin description [49], which contains a description of the plugin and some example screenshots, the plugin store automatically lists an overview of all shortcuts and possible settings for the user, as shown in figure 93.

▼ Settings (6)		
Name	Description	Default
dafny.languageServerExePath	Relative path to the DafnyServer.exe	../../../../dafny-language-server/Binaries/DafnyLanguageServer.exe
dafny.compilationArgs	Optional array of strings as Dafny compilation arguments	/compile:1,/nologo
dafny.useMono	Only applicable to _Windows_! Requires _NET_ 4.5 or higher when set to 'false'.	false
dafny.monoPath	Mono binary with absolute path. Only necessary if mono is not in system PATH. Ignored on Windows when useMono is set to false.	
dafny.monoExecutable	Mono executable with absolute path. Only necessary if mono is not in system PATH (you'll get an error if that's the case). Ignored on Windows when useMono is false.	
dafny.colorCounterExamples	Customize the color (HEX) of Counter Examples. There are two default colors: for dark theme (#0d47a1, #e3f2fd) and light theme (#bbdefb, #102027). This color setting will override both defaults.	[object Object]

▼ Commands (6)			
Name	Description	Keyboard Shortcuts	Menu Contexts
dafny.restartDafnyServer	Dafny: Restart DafnyServer	F9	editor/context
dafny.compile	Dafny: Compile	Ctrl + Shift + B	editor/context
dafny.compileCustomArgs	Dafny: Compile with Custom Arguments	F6	editor/context
dafny.compileAndRun	Dafny: Compile and Run	F5	editor/context
dafny.showCounterExample	Dafny: Show CounterExample	F7	editor/context
dafny.hideCounterExample	Dafny: Hide CounterExample	F8	editor/context

Figure 93: Settings, Commands and Keyboard Shortcuts

After the plugin is installed and the Dafny file is opened again, the plugin automatically starts the download of the Dafny language server and starts it afterwards as shown in figure 94.

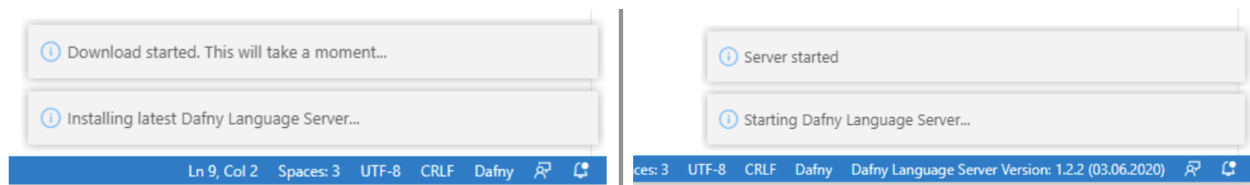


Figure 94: Popup Messages for Installation (Left) and Starting (Right)

That is it, the user has the plugin and the Dafny language server ready to use. He can start programming in Dafny.

## 7.4 Performance of the Language Server

To measure performance, a little algorithm was written that creates a pseudorandom Dafny file. Within a method, code lines are generated, either containing

- A variable definition: `var v142 := v16;`
- A variable access: `print v142;`
- A block scope: `while (true) { ...`
- Ending a blockscope: `}`

The chance to create a variable or to create print statement is 90%, thus the generated files will contain about 90% variable name segments, that have to be resolved. This challenges the symbol table quite a bit. A tree is still grown due to the while-blocks, so not all symbols are within the same tree branch. Since `textDocument/didOpen`, which triggers the symbol table instantiation, is not available, a random LSP request was sent after the opening to wait until all actions are finished.

The test has shown the following results:

- 100 LOC: 500ms
- 1000 LOC: 1200ms
- 10'000 LOC: 18'000ms

A resulting trend line is shown in figure 95.

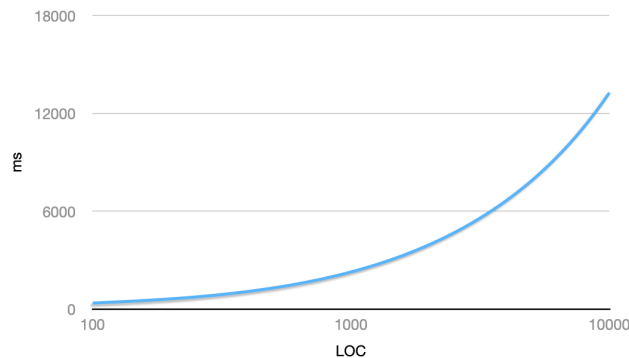


Figure 95: Runtime Trend Line

This distribution follows a  $O(n * \log(n))$  scheme, which was expected. While 18 seconds seem quite high, consider that 10'000 lines of code is huge. While we were working on the `DafnyAST.cs` file, which also contains 10'000 LOC, even Visual Studio struggled. Practical tests have shown that everyday usage of the plugin is pleasant. It generally reacts fast enough. All methods are `async`, so that the user is never blocked in his activity.

## 7.5 Metrics and Quality Measures

This chapter describes the most important metrics and quality measures for the server and the client. The results are also compared to the prototype.

### 7.5.1 Server

The language server consists of 5'000 lines of code. The following report is generated by Visual Studio.

Hierarchy	Maintainability index	Cyclomatic Complexity	Depth of inheritance	Class Coupling	Lines of Source code	Lines of Executable code
LanguageServer\DafnyLanguageServer (Debug)	85	898	3	375	5'049	1'197
DafnyLanguageServer	80	10	1	33	131	32
DafnyLanguageServer.Commons	87	57	1	29	270	68
DafnyLanguageServer.Core	86	171	1	138	1'038	298
DafnyLanguageServer.CustomDTOs	100	18	1	8	44	2
DafnyLanguageServer.DafnyAccess	83	49	1	52	324	79
DafnyLanguageServer.Handler	82	60	2	132	608	147
DafnyLanguageServer.SymbolTable	86	439	3	76	1'999	449
DafnyLanguageServer.Tools	74	17	1	28	158	25
DafnyLanguageServer.Tools.ConfigInitialization	83	45	1	21	281	62
DafnyLanguageServer.WorkspaceManager	90	32	1	22	196	35

Figure 96: Server Package Dependencies, generated by Visual Studio

Contrary to the Visual Studio report, Sonar will report 4'000 LOC. Compared to the prototype, this is an increase of about 200%. Aside the increased functionality, this is mainly to explain by the extensive and repetitive code necessary for the visitor within the `SymbolTable` namespace. Just to create a single symbol, about 20 LOC are necessary as seen in chapter 6.

For comparison: The original language server written in TypeScript contained 3'100 LOC, rounded off [55]. But this only includes the component forming the TypeScript language server – the additional backend they had to write in C# is not included. The table shown in figure 97 shows the ten largest components of the old language server.

Path	LOC
<b>server/src/vscodeFunctions/documentDecorator.ts</b>	287
<b>server/src/server.ts</b>	252
<b>server/src/backend/dafnyInstaller.ts</b>	227
<b>server/src/backend/dafnyServer.ts</b>	204
<b>server/src/backend/features/symbols.ts</b>	160
<b>server/src/backend/features/symbolService.ts</b>	150
<b>server/src/strings/stringResources.ts</b>	133
<b>server/src/backend/verificationResults.ts</b>	125
<b>server/src/vscodeFunctions/documentIterator.ts</b>	99
<b>server/src/backend/environment.ts</b>	94

Figure 97: 10 Biggest Component in the old TypeScript Language Server

Interesting is the comparison of the code base required for a feature like *GoToDefinition*. The new `DefinitionsProvider` just uses 38 lines of code. However, only about five are used for the core logic, all others are just error handling and user messaging. This is a major improvement compared to the pre-existing project.

The Sonar report for the server only detects about fifteen issues. The security aspects were ensured to be no problem, since the server is running on a local machine. The remaining smells would lower the code quality. Thus, we decided to ignore these issues. For example, Sonar would like us to put all `Visit` methods next to each other, and also all `Leave` methods. However, when they belong to the same AST node, we prefer to keep the `Visit` and `Leave` of that AST-node together.

To Sonar report is shown in figure 98.

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
📁 DafnyLanguageServer	4,035	0	0	14	4	—	0.0%
📁 Commons	209	0	0	0	0	—	0.0%
📁 Core	912	0	0	2	0	—	0.0%
📁 CustomDTOs	44	0	0	0	0	—	0.0%
📁 DafnyAccess	223	0	0	1	0	—	0.0%
📁 Handler	561	0	0	0	0	—	0.0%
📁 SymbolTable	1,478	0	0	10	0	—	0.0%
📁 Tools	351	0	0	1	1	—	0.0%
📁 WorkspaceManager	137	0	0	0	0	—	0.0%
📄 DafnyLanguageServer.cs	108	0	0	0	2	—	0.0%
📄 Program.cs	12	0	0	0	1	—	0.0%

Figure 98: Sonar Server Report

### 7.5.2 Client

In the client, a direct comparison between the pre-existing plugin and the new plugin can be made thanks to SonarCloud.

Compared to the original 900 LOC [56] in the client part of the old plugin, the new plugin reaches 1'400 LOC [57]. The numbers are displayed in figure 99.

Lines of Code	Lines of Code
📁 src 909	📁 src 1,433
📁 localExecutionHelpers 64	📁 dafnyLanguageServerStartup 379
📁 server 250	📁 ideApi —
📁 stringResources 85	📁 localExecution 110
📁 typeInterfaces 24	📁 serverRequests 174
📁 ui 442	📁 stringResources 141
📄 extension.ts 44	📁 typeInterfaces 56
	📁 ui 545
	📄 extension.ts 28

Figure 99: LOC Client Old (Left) vs Client New (Right)

This may be surprising at first. But the following factors must be taken into account:

- The new client implements own interfaces and contains more code due to the encapsulations. The typeInterfaces directory is almost three times as large.

- Strings are stored centrally in the resource package. As a result, the package is now twice as large.
- We added new configuration options such as dark and light mode support for *CounterExample*.
- The size of the original server requests in `server` could even be reduced by almost 100 LOC in the new version `serverRequests`.

Lines of Code	909	Lines of Code	1,433
Lines	1,115	Lines	1,795
Statements	237	Statements	374
Functions	67	Functions	102
Classes	21	Classes	26
Files	20	Files	44
Comment Lines	87	Comment Lines	122
Comments (%)	8.7%	Comments (%)	7.8%

Figure 100: Project Size Client Old (Left) vs Client New (Right)

As shown in figure 100, we have over twice as many files due to the division of components and interfaces. If we calculate the LOC divided by the number of files, we get for the old client on average 45 LOC per file, and in the new client 33 LOC per file. This is certainly a good result.

## 8 Conclusion

This chapter concludes the thesis and the puts the results into context. It ends up with some outlooks for future improvements.

### 8.1 Project Summary

The project is a success. The features of the prototype could be improved significantly, and new, major features were added to create a comprehensive user experience.

In comparison to the initial solution, any pre-existing features were improved.

- *Compilation* will no longer start a dedicated Dafny process for compilation, but instead use the Dafny backend directly.
- The representation of *CounterExamples* is now much cleaner and easier to perceive.
- *Verification* will also show syntax errors, not only logical errors.
- *Verification* will now also show warnings.
- *Verification* will no longer just highlight single characters.
- *CodeLens* can now actively be used to preview symbol usages.
- *AutoCompletion* no longer performs simple pattern matching, but provides exact results.

*Rename* and *HoverInformation* could newly be implemented due to the offerings of the symbol table.

The plugin is quite robust and will recover from exceptional states. If something goes completely wrong, an option to restart the language server is available.

Due to timely reasons, the visitor could not be implemented completely. When a project time frame is strictly given, the scope has to be variable [58], thus we needed to cut out some Dafny language features from the visitor. However, completing it is not necessarily complex, but rather a diligence task. The foundations for continue to work on the visitor are set. Thus, we are confident that this will be done within further development of the plugin.

For us, it was important that the plugin will somehow stay alive whenever it encounters language features that are not supported. It was targeted to just do nothing as a fault tolerant design, whenever an exceptional state occurs, which worked out quite well.

Although the visitor lacks support for a few advanced language features, the plugin provides a great experience for someone taking first steps in Dafny. The target persona was the student, completing the HSR-internal exercises on Dafny, for which the plugin is more than sufficient. Our user experience tests have shown that the plugin accommodates this requirement.

The existing features which base on the symbol table will automatically scale, once the visitor is completely implemented. The plugin will then automatically gain a lot of quality and can also target more advanced users that use more advanced language features.

The `DafnyServer` component, which originates from the pre-existing project, could be be completely superseded.

The codebase of the project is well organized, split into components, uses interfaces where ever possible and has a nice maintainability.



## 8.2 Deployment

The plugin has been uploaded to the VSCode marketplace and is available for public. This is a huge success for the project and makes us quite proud, since the result of our bachelor thesis is a product that actually gets used by other people.

## 8.3 Outlook

In this chapter, the further development of the project is described. This includes the complete implementation of the visitor, more LSP extensions, as well as more refactorings and features.

### 8.3.1 Completion of the Visitor

As mentioned previously in this chapter, the visitor should be completed to also target more advanced users. Once this is done, the project instantly gains a lot of quality, since all targeted features scale automatically with the symbol table. While it was said that this is rather a diligence task, some problems may arise as a result. For example, when inheritance was implemented, a whole new challenge appeared. Declarations in base classes were not found, since we could no longer just move up the symbol tree. Possible inherited symbols had to be treated separately. Thus, the task of completing the visitor should not be underestimate.

### 8.3.2 LSP Extensions

The language server protocol [9] offers a lot more than what is currently implemented. Many of these features can now easily be implemented, since the symbol table offers all information required. Indeed, many of the following features can be implemented in just a few minutes to hours.

For example, the foundations for `textDocument/documentHighlight` [9] are already completely supported by the symbol table. The feature highlights all occurrences of a symbol, given a certain text position. The developer just has to request the symbol at the text position, which is passed by the client. Then, all occurrences have to be requested, by just calling `symbol.GetAllOccurrences()`. Finally, these occurrences have to be assembled into the target response format `DocumentHighlight[]` and can be sent back to the client.

`textDocument/documentHighlight` is actually very similar to `textDocument/rename` and would be a good first step for a future development team, for example if this project is further developed within another bachelor thesis.

Besides highlights, the following LSP requests are also of interest for future extensions:

- `workspace/didChangeWatchedFiles`: Actually handle the request (for example remove the file from the buffer if it was deleted) instead of relying on `textDocument/didChange`.
- `workspace/symbol`: A request that sends all, project-wide symbols to the client. This could be done relatively easy with the symbol table.
- `textDocument/didClose`: Currently, nothing is done. The proper file could actually be removed from the buffer in this case.
- Differentiation between `textDocument/definition` and `textDocument/declaration`.
- `textDocument/implementation`: Go to the implementation of a declared symbol.
- `textDocument/formatting`: Auto formatting would actually be simpler to implement than one may think. Since the symbol table keeps track of scopes, the scope depth is well known. Indentation could just be done according to the scope depth.

### 8.3.3 Refactorings

The language server protocol also offers a code action request. It can be used for quick fixes, which would be a nice addendum to the plugin. For example, if Dafny reports that a semicolon is missing, a quick fix would just be to insert a semicolon at the target position.

Alongside quick fixes, the LSP also provides support for refactorings like extractions (extract method, extract variable...) [9]. The result can directly be transferred via the LSP within a `CodeAction` container, that contains all text edits which have to be done. Support for these refactorings would increase the quality of the plugin even further. The symbol table is also providing assistance for these features. For example, for *extract method*, the symbol table could provide information about what symbols have to be passed as an argument, and which symbols are local variables and get extracted, too. This example shows how far the benefits of the implemented symbol table reach. A whole new set of features became possible.

### 8.3.4 Dafny Specific Functionality

Further extension points would be to work on Dafny-specific features. An optional goal of this bachelor thesis was the automated generation of contracts, which could not be implemented due to timely reasons. Thus, it remains available as a possible extension point for the future. Here, even topics like AI come into play and the project would involve completely different fields of work.

A feature that was discussed already back at the preceding bachelor thesis [1] was the support for debugging. However, this is a very complex topic and may not even fit into the scope of a bachelor thesis.

### 8.3.5 More IDEs

Currently, only Visual Studio Code is supported. The idea of the language server protocol is, that the server is isolated and can connect to various clients. Thus, a suitable extension point would be to implement support for other IDE's, like Eclipse or Atom. Within this bachelor thesis, the VSCode client was refactored completely and the code was cleaned up. Thus, it is even more lightweight than before and adaption for other IDE's would not cost much effort. Support for the native LSP features come out of the box. The client developer would just have to implement the language server connection establishment and provide support for the additional features, namely *Compile* and *CounterExample*.

## 8.4 Further Achievements

At the beginning of the bachelor thesis, we were given a list of goals. The goals targeting the feature set, the symbol table or the architecture were already discussed. Also many minor goals were achieved:

- LaTeX was used instead of Microsoft Word.
- Client and server are split into separate repositories.
- The language server was rebased multiple times to be on par with the newest Dafny upstream.
- No local paths occur in the code.
- General code cleanup was done.
- Clean folder structure was implemented, especially for tests, too.
- Tests run clean, not every test has to be added manually to the CI.
- SonarQube is now running.

- Logging is injected properly and a common logging framework is used.
- The Dafny library is directly integrated everywhere by now.
- Client was completely refactored and cleaned up.
- Symbol Table was implemented.
- Previous features were improved.
- Small feature branches were used on git with short commits and commit messages were slightly improved compared to the prototype development.
- No todo remain in the code.
- Integration tests run properly with OmniSharp's client.

## 9 Project Management

This chapter reflects organizational aspects. We will compare relevant propositions from the project plan to the actual state [59]. First, we talk about time management. Then, quality aspects will be discussed.

### 9.1 Time Management

This chapter shows the time spent and how well we were able to stick to the planned schedule. The times are taken from Redmine [60].

#### 9.1.1 Time Spent From Each Student

The total working hours for the bachelor thesis was defined to be 12 ECTS or 360 hours per student [59]. The project lasted 17 weeks, which averages out in a nominal working time of 21.2 hours per week per student [59]. During the last two weeks, no other classes took place and more time resources were available [59]. The submission of the thesis was brought forward to Wednesday, the 10th of June 2020 instead of the following Friday.

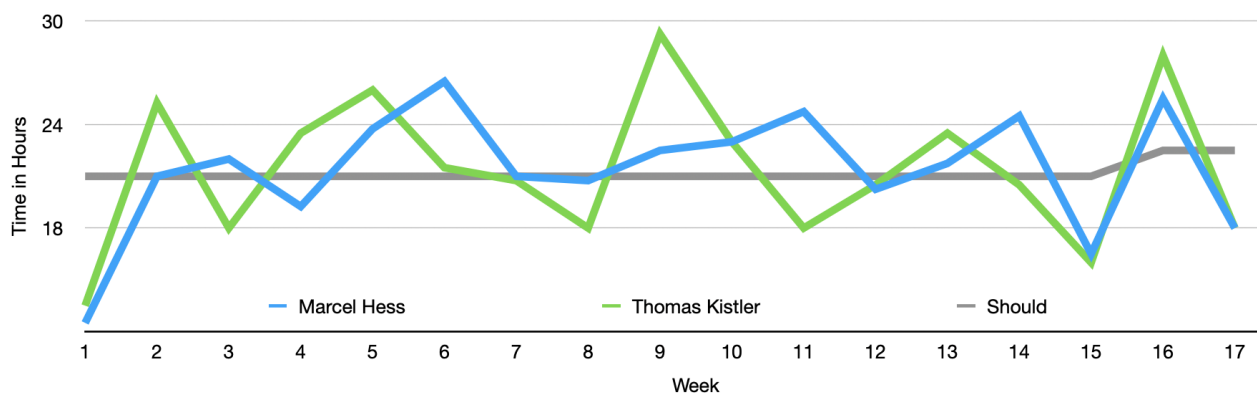


Figure 101: Time Tracking

As shown in figure 101, each student spent about 363 hours in total. In the last two weeks, a significant increase in workload can be seen.

Figure 102 shows the invested time by activity. The largest part was development, followed by documentation and meetings. We rate this basic time split as well taken. For testing, a relatively small percentage was spent according to the diagram. Since the boundary between implementation and testing is not so clear for test driven development, time was sometimes rather reported for development than for testing.

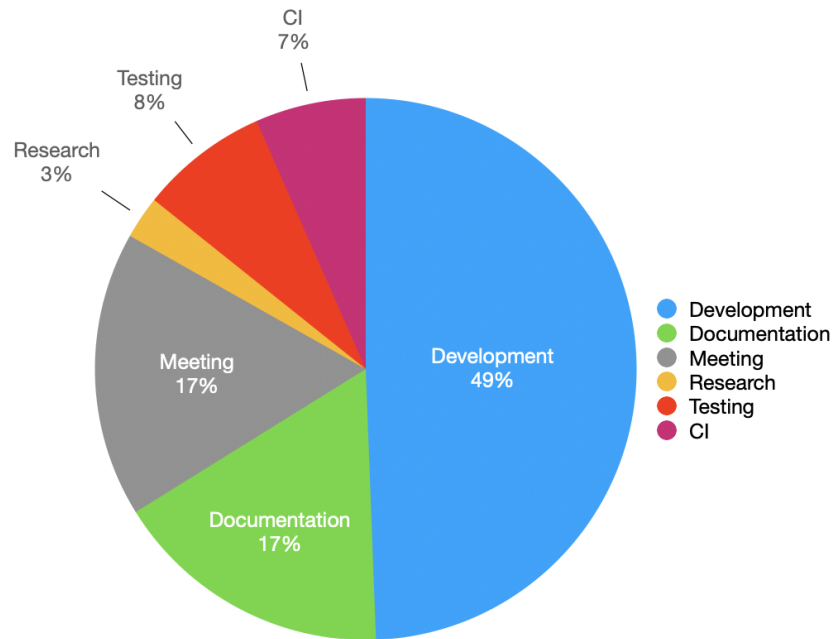


Figure 102: Time by Activity

Figure 103 shows the accumulated time per milestone. The individual milestones are described in more detail in the following section. It is interesting how little time the transition took and how time-consuming the symbol table was.

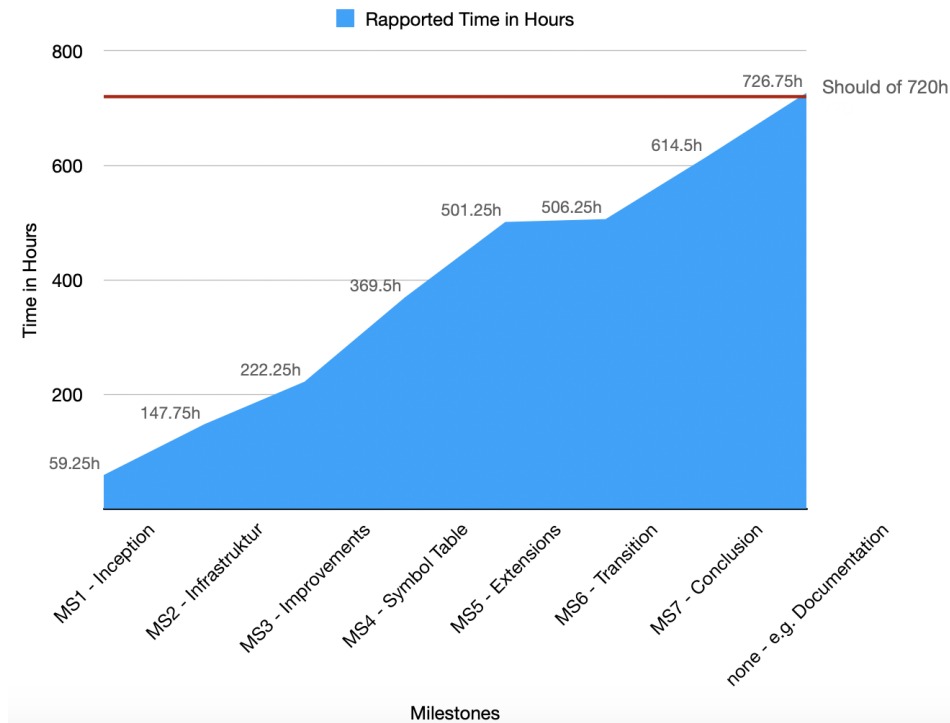


Figure 103: Time by Milestone

Apart from the time-consuming meetings and documentation, the symbol table was the most intensive ticket. This is followed by the newly implemented client. Figure 104 shows the six most time consuming tickets.

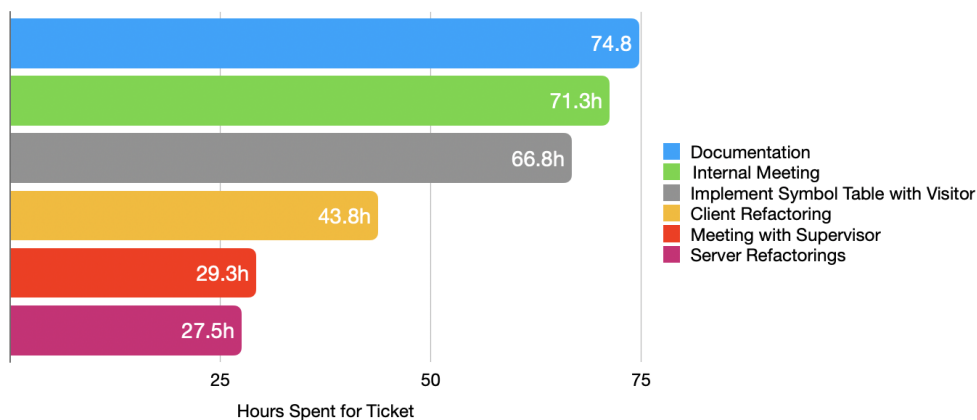


Figure 104: Top 6 Time Consuming Tickets

Figure 105 shows the reported time per core feature. The numbers are to be enjoyed with caution. A lot of time spent on features was also reported on dedicated tickets for testing or refactoring. Nevertheless, the graphic gives an overview of which features were more time-consuming and which were easier to implement.

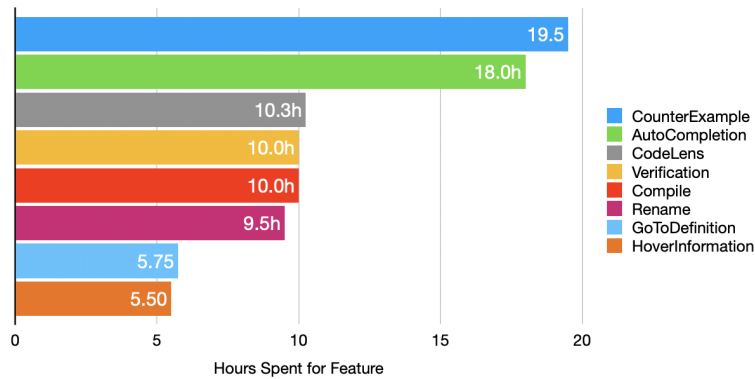


Figure 105: Time for Each Feature

### 9.1.2 Schedule and Scope

As planned in the project plan, we used a mix between unified process and SCRUM. The project was split into an inception, elaboration, construction and a transition stage as you can see in figure 106.

W 01	W 02	W 03	W 04	W 05	W 06	W 07	W 08	W 09	W 10	W 11	W 12	W 13	W 14	W 15	W 16	W 17
Inception		Infrastructure		Construction										Transition	Conclusion	
				Improvements		Symbol Table				Extensions						

Figure 106: Project Staging

Each of the stages was split into sprints lasting one week. This is shown in figure 107.

	W 01	W 02	W 03	W 04	W 05	W 06	W 07	W 08	W 09	W 10	W 11	W 12	W 13	W 14	W 15	W 16	W 17
Inception	█	█															
Infrastructure			█	█													
Improvements					█	█											
Symbol Table							█	█	█	█							
Extensions											█	█	█	█			
Transition															█		
Conclusion																█	█

Figure 107: Planned Milestones

In general, we were able to stick to this plan pretty well as you can see in figure 108.

	W 01	W 02	W 03	W 04	W 05	W 06	W 07	W 08	W 09	W 10	W 11	W 12	W 13	W 14	W 15	W 16	W 17
Inception	Green	Green															
Infrastructure		Orange	Green	Green	Orange												
Improvements					Green	Green											
Symbol Table							Green	Green	Green	Green	Orange			Orange			
Extensions										Orange	Green	Green	Green	Green	Orange		
Transition										Orange					Green	Orange	
Conclusion																Green	Green

Figure 108: Actual Milestones

### Inception

In the first two weeks, we found our way back into the project from the prototype and set up our environments accordingly. This included updating the IDEs, OmniSharp, Z3 and Boogie, as well as setting up a new Redmine ticket system and create the project plan.

### Infrastructure

Since the majority of the inception stage could be processed very quickly, we were able to start with parts of the infrastructure a week earlier. On the one hand, we got the old CI running again with the new updated versions of Boogie and Dafny, and on the other hand, the CI was extended by SonarQube for C#. This also included splitting the client and server projects into different git repositories.

In addition, we have revised the concept for integration tests and developed a working test-prototype. Completing the CI process had been somewhat delayed, but was then successfully finished.

### Improvements

In this milestone, we took care to improve the existing features. Only features that are not related to the symbol table were dealt with. This included:

- *Compilation*
- *CounterExample*
- *Verification*

Small improvements of the client were also done, such as a bug fix in the status bar. This milestone could be completed on time.

### Symbol Table

At the beginning of this milestone, we developed the concept and the basic implementation for the symbol table. A prototype with support for a few AST-Nodes could be achieved very quickly. However, implementing support for the massive amount of AST-Nodes took very long.

We started to already make use of the symbol table before the symbol table was complete. This was done to gain a break from the tedious visitor implementation.

### Extensions

As described above, this milestone was started a little earlier.

The content of this milestone was primarily to make the existing features - such as *AutoCompletion* and *CodeLens* - use the new symbol table. In addition to the improvement of the existing features, we also added new features like *HoverInformation* and *Rename*, which are also based on the new symbol table.



### **Transition**

The transition stage was primarily based about preparing our plugin for the release. We noticed relatively early that there were problems with the support for Mono. Therefore, we already made rough clarifications in week ten and wrote posts in appropriate forums, namely the OmniSharp slack channel [38].

The transition took a little longer, since the the plugin had to be finalized before it could be published. Instead, we started a bit earlier with writing the documentation.

### **Conclusion**

The conclusion was basically used to write the documentation and as a general buffer time. The buffer time was not necessary, since we respected our plan and showed enough courage to leave out some visitor functionality in order to be on time. The buffer could then be used to do some finalization work and prepare the presentation.

## **9.2 Effects of the COVID-19 Pandemic**

The cooperation was only marginally affected by the Corona situation. Physical meetings were replaced by virtual video conferences and screen sharing. This was done for the meetings with the supervisors, but also for internal meetings of the two students.

### **9.2.1 Meetings**

The weekly meetings with our supervisors Thomas Corbat and Fabian Hauser were held in general each Thursday at 10:30am [59]. Thomas Kistler and Marcel Hess had a reserved time frame of 3x8h from Wednesday to Friday each week to work on the project [59]. Whenever necessary, internal meetings were held within this time frame [59].

### **9.2.2 Division of Labour**

Thomas Kistler was more involved in the realisation of the symbol table, while Marcel Hess was working a bit more on the client implementation. However, both were always involved in both concepts. Basic design implementations were therefore mostly implemented in pair programming.

In the implementation of core providers, both participated equally. To enable a precise tracking of the activities and also to facilitate collaboration, we worked with the version management git.

## **9.3 Quality Aspects**

This chapter highlights certain quality aspects that have been crucial for good code quality. In general, the requirements placed in the project plan were fulfilled [59].

### **9.3.1 Code Reviews**

There were three layers of non-automated code quality control.

1. Internal code reviews of the students. Critical and difficult parts of the code were always underlaying to a internal code review.
2. Reviews with the supervisors.
3. The four-eye principle using merge requests.

Since GitLab supports merge requests, merge requests for independent changes were sent to the project partner for control and approval. If no errors or improvement potential was found, the merge request was accepted. If there were minor issues, they were improved immediately. In the case of major discrepancies, a code review was convened.

### 9.3.2 Continuous Integration (CI)

According to the project plan, we wanted to resolve all CI-issues at the beginning of the bachelor thesis, so that we can then profit by a supportive workflow [59]. This goal could be achieved. By separating the client and server into two different repositories, two new CI pipelines were created. These are then discussed for the server and for the client.

#### Client

The client goes through three pipeline phases as shown in figure 109:

1. *Test*
2. *Build*
3. *Sonar*

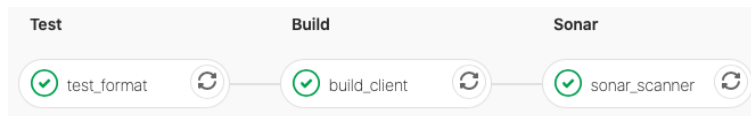


Figure 109: CI Pipelines for the Client

In the test phase, prettier [27] is used to check if the TypeScript code is formatted according to the style guidelines [59]. The build pipeline checks whether the TypeScript code can be compiled without errors. If there are errors in the TypeScript or warnings - such as an unused variable - the build will fail. Finally, the scanner for SonarQube is run and the report is sent to SonarCloud.

#### Server

The pipeline for the server was slightly modified. In order to create the SonarQube report, the build and Sonar scan processes had to be merged as shown in figure 110.



Figure 110: CI Pipelines for the Server

After the build has been created and the SonarQube report has been successfully submitted to SonarCloud, the test phase begins. Existing Dafny tests are run to verify that we have not inadvertently changed anything in the Dafny project, and our own unit and integration tests are executed in the `test_nunit` stage.

The automated CI processes were a great enrichment for our work. The tests were automatically run for checked-in changes. If these failed, no merge request was created. This automated control ensured that a new bug was not accidentally overlooked and that the quality standard was maintained.

### 9.3.3 Static Code Analysis

Aside SonarQube, ReSharper was used locally for the server to gain additional insights. Regarding the client, SonarLint was used to get a local, static code analysis [61]. The fact that a static code analysis could already be carried out locally meant that errors could be already corrected during development and were not only noticed after the CI had failed.

SonarQube was used for the server and for the client within the CI process. To integrate SonarQube into our CI process, we use the SonarCloud as a platform [62]. This is shown in figure 111.

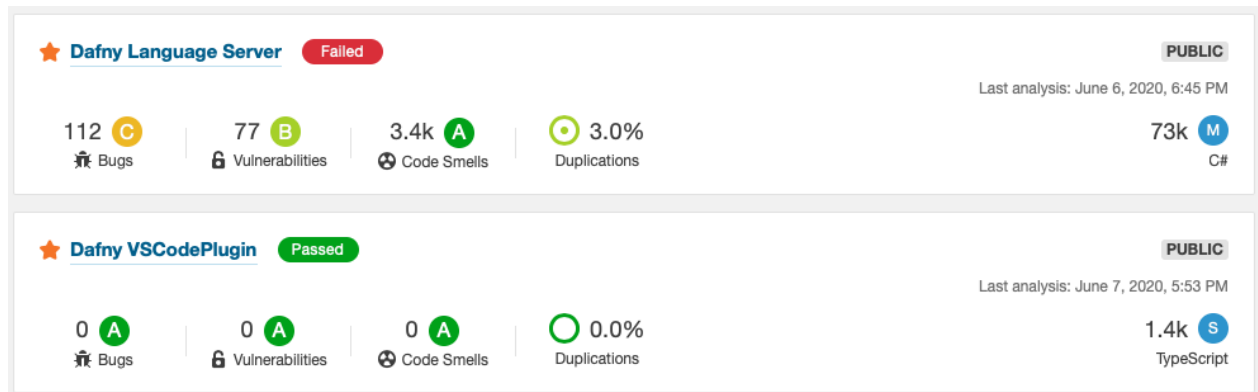


Figure 111: SonarQube Report in SonarCloud

It stands out that there is a high number of code smells and bugs. These high numbers are due to the fact that the entire Dafny project is analyzed by Sonar. The actual numbers for our project are of course much lower, as shown in figure 112. The `DafnyLanguageServer` is highlighted in yellow.

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
Source	72,971	112	77	3,411	9	—	3.0%
Dafny	67,150	111	76	3,356	4	—	3.3%
DafnyDriver	545	0	0	8	1	—	0.0%
DafnyLanguageServer	4,035	0	0	14	4	—	0.0%
DafnyRuntime	1,238	1	1	33	0	—	0.0%
version.cs	3	0	0	0	0	—	0.0%

Figure 112: SonarQube Report in SonarCloud for the Dafny project

Furthermore, local plugins for VSCode and Visual Studio were used to automatically format the code according to our styling guides [27]. Thus our code was always automatically formatted uniformly.

More detailed code metrics can be found in section 7.5.

### 9.3.4 Test Coverage

Since our integration tests are able to test all components of the language server and the client contains basically no logic, tests were written only for the Dafny language server.

Symbol ▲	Coverage (%)	Uncovered/Total Stmts.
▲ Total	87%	319/2375
▲ LanguageServer	87%	319/2375
▲ DafnyLanguageServer	87%	319/2375
▲ DafnyLanguageServer	87%	319/2375
▶ Commons	98%	2/132
▶ Core	91%	54/582
▶ CustomDTOs	100%	0/20
▶ DafnyAccess	92%	13/162
▶ Handler	80%	54/272
▶ SymbolTable	83%	146/853
▶ Tools	91%	20/217
▶ WorkspaceManager	73%	21/78
▶ DafnyLanguageServer	84%	9/55
▶ Program	100%	0/4

Figure 113: Language Server Test Coverage

The test coverage can be seen in figure 113. We did not achieve a full 100%, but still a very high percentage was reached. Components that contain no important logic remain untested. For example, code fragments that call Dafny itself and only pass information were not specifically tested, since no advantage would be gained. Listing 43 shows another code fragment that pulls down test coverage. It is taken from the DeclarationVisitor.

```

1 public override void Visit(ThisExpr e) {
2     throw new InvalidOperationException(Resources.ExceptionMessages.
3         visit_only_declarations);
4 }
5 public override void Leave(ThisExpr e) {
6     throw new InvalidOperationException(Resources.ExceptionMessages.
7         visit_only_declarations);
8 }
9 public override void Visit(DisplayExpression o) {
10    throw new InvalidOperationException(Resources.ExceptionMessages.
11        visit_only_declarations);
12 }
13 public override void Leave(DisplayExpression o) {
14    throw new InvalidOperationException(Resources.ExceptionMessages.
15        visit_only_declarations);
16 }
17 public override void Visit(ComprehensionExpr o) {
18    throw new InvalidOperationException(Resources.ExceptionMessages.
19        visit_only_declarations);

```

```
20
21 public override void Leave(ComprehensionExpr o) {
22     throw new InvalidOperationException(Resources.ExceptionMessages.
        visit_only_declarations);
23 }
24
25 ...
```

---

Listing 43: Untested Code

The `DeclarationVisitor` is not visiting any of these AST-nodes (the `DeepVisitor` is doing that), yet it has to provide the methods for them because it is inheriting from the base class `Visitor`. However, whenever core logic takes place, the code is tested. Thus, we are well satisfied with the test coverage of more than 85%.

## Glossary

**AST** Abstract Syntax Tree. 30, 82

**Boogie** An intermediate verification language. 14, 41

**CI** Continuous Integration. 19, 41, 43, 117, 133

**CodeLens** Preview of code usages. This word is a trademark by Microsoft. 11, 40, 91

**Docker** Tool for Container Virtualization. 41, 104

**GitLab** Project Integration Software, using git. 43, 117

**IDE** Integrated Development Environment. 9, 15

**JSON** JavaScript Object Notation. 15, 79

**JSON-RPC** JavaScript Object Notation Remote Procedure Call - to exchange data via JSON. 19

**lit** Test Runner. 41

**LOC** Lines of Code. 17, 119

**LSP** Language Server Protocol. 9, 15, 19

**Mono** Mono allows to execute `.exe` files on Unix based operating systems. 68, 94

**MSBuild** Compiler for .NET. 41, 94

**NuGet** Package Manager for .NET. 22

**nUnit** Testing Framework. 41, 98

**OmniSharp** Provider for LSP in C#. 19, 22

**PowerShell** Command line interface by Microsoft, combined with a script language. 37

**Pre-existing Project** Result of the Previous Bachelor Thesis by Markus Schaden and Rafael Krucker. 16

**Preceding Semester Thesis** Text of the Preceding Semester Thesis. 16

**Prototype** Result of the Preceding Semester Thesis. 16

**ReSharper** Visual Studio Extension. 43

**SonarQube** Tool for Static Code Analysis. 17, 41

**UI** User Interface. 62, 64, 69

**uri** Uniform Resource Identifier. 21, 52, 76

**Visitor** Programming Language Pattern. 59, 82

**VSCoDe** Visual Studio Code. 9

**Z3** A logical theorem prover.. 14, 41

## References

- [1] Rafael Krucker and Markus Schaden. *Visual Studio Code Integration for the Dafny Language and Program Verifier*. <https://eprints.hsr.ch/603/>. HSR Hochschule für Technik Rapperswil, 2017.
- [2] *HSR Correctness Lab*. URL: <https://www.correctness-lab.ch/>. (Accessed: 14.12.2019).
- [3] Marcel Hess and Thomas Kistler. *Dafny Language Server Redesign*. HSR Hochschule für Technik Rapperswil, 2019/20.
- [4] *Dafny, Wikipedia*. URL: <https://en.wikipedia.org/wiki/Dafny>. (Accessed: 14.05.2020).
- [5] *Boogie*. URL: <https://github.com/boogie-org/boogie>. (Accessed: 01.06.2020).
- [6] *Z3*. URL: <https://github.com/Z3Prover/z3>. (Accessed: 01.06.2020).
- [7] *JSON*. URL: <https://en.wikipedia.org/wiki/JSON>. (Accessed: 04.06.2020).
- [8] *Language Server Protocol Wikipedia*. URL: [https://en.wikipedia.org/w/index.php?title=Language\\_Server\\_Protocol](https://en.wikipedia.org/w/index.php?title=Language_Server_Protocol). (Accessed: 26.05.2020).
- [9] *Language Server Protocol Specification*. URL: <https://microsoft.github.io/language-server-protocol/specification>. (Accessed: 26.05.2020).
- [10] *Language Server Extension Guide*. URL: <https://code.visualstudio.com/api/language-extensions/language-server-extension-guide>. (Accessed: 15.12.2019).
- [11] *Langserver.org*. URL: <https://langserver.org/>. (Accessed: 05.12.2019).
- [12] *NuGet*. URL: <https://en.wikipedia.org/wiki/NuGet>. (Accessed: 04.06.2020).
- [13] *Martin Björkström - Creating a language server using .NET*. URL: <http://martinbjorkstrom.com/posts/2018-11-29-creating-a-language-server>. (Accessed: 14.10.2019).
- [14] *OmniSharp/csharp-language-server-protocol*. URL: <https://github.com/OmniSharp/csharp-language-server-protocol>. (Accessed: 05.12.2019).
- [15] *Martin Björkström - Creating a language server using .NET*. URL: <https://app.slack.com/client/T0RE90CRF/C804W8JHE>. (Accessed: 05.12.2019).
- [16] *Your First Extension*. URL: <https://code.visualstudio.com/api/get-started/your-first-extension>. (Accessed: 28.05.2020).
- [17] *Extension Guides*. URL: <https://code.visualstudio.com/api/extension-guides/overview>. (Accessed: 28.05.2020).
- [18] K. Rustan M. Leino Richard L. Ford. *Dafny Reference Manual*. Available at <https://homepage.cs.uiowa.edu/~tinelli/classes/181/Papers/dafny-reference.pdf>. 2017.
- [19] *Functions vs. Methods*. URL: <https://www.engr.mun.ca/~theo/Courses/AlgCoCo/6892-downloads/dafny-notes-010.pdf>. (Accessed: 15.04.2020).
- [20] *Coco/R*. URL: <http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/>. (Accessed: 28.05.2020).
- [21] *Haskell*. URL: <https://www.haskell.org/>. (Accessed: 28.05.2020).
- [22] *Merge pull request from robin-aws/update-syntax*. URL: <https://github.com/DafnyVSCode/Dafny-VSCode/commit/039695d0e9bf7f7ba4f6f3b465d1cb7c07e987fe>. (Accessed: 28.05.2020).
- [23] *VSCode Extension API*. URL: <https://code.visualstudio.com/api>. (Accessed: 22.05.2020).
- [24] *nUnit*. URL: <https://nunit.org/>. (Accessed: 02.06.2020).
- [25] *LLVM lit runner*. URL: <https://llvm.org/docs/CommandGuide/lit.html>. (Accessed: 02.06.2020).

- [26] *DevOps Concepts: Pets vs Cattle*. URL: <https://medium.com/@Joachim8675309/devops-concepts-pets-vs-cattle-2380b5aab313>. (Accessed: 30.05.2020).
- [27] Marcel Hess and Thomas Kistler. *Developer Documentation*. HSR Hochschule für Technik Rapperswil, 2020.
- [28] *Data Types in TypeScript*. URL: <https://www.geeksforgeeks.org/data-types-in-typescript/>. (Accessed: 15.04.2020).
- [29] Gamma et al. *Design Patterns*. Addison-Wesley, 1995. ISBN: 0-201-63361-2.
- [30] *Atom Package*. URL: <https://blog.eleven-labs.com/en/create-atom-package/>. (Accessed: 04.06.2020).
- [31] *Eclipse IDE Plug-in*. URL: <https://www.codeproject.com/Tips/893547/How-to-Create-Your-Own-Eclipse-IDE-Plug-in>. (Accessed: 04.06.2020).
- [32] *Dafny lang, Releases*. URL: <https://github.com/dafny-lang/dafny/releases>. (Accessed: 30.05.2020).
- [33] *TypeScript Dafny Installer*. URL: <https://github.com/DafnyVSCode/Dafny-VSCode/blob/develop/server/src/backend/dafnyInstaller.ts>. (Accessed: 02.06.2020).
- [34] *Newtonsoft Json.NET*. URL: <https://www.newtonsoft.com/json>. (Accessed: 22.05.2020).
- [35] *CommandLineParser*. URL: <https://github.com/commandlineparser/commandline>. (Accessed: 22.05.2020).
- [36] *Serilog*. URL: <https://serilog.net/>. (Accessed: 22.05.2020).
- [37] *Mono*. URL: <https://www.mono-project.com>. (Accessed: 30.05.2020).
- [38] *OmniSharp Slack - Mono*. URL: <https://omnisharp.slack.com/archives/C804W8JHE/p1587578976071500>. (Accessed: 24.04.2020).
- [39] *OmniSharp GitHub Issue - Mono*. URL: <https://github.com/OmniSharp/csharp-language-server-protocol/issues/179>. (Accessed: 24.04.2020).
- [40] *OmniSharp Language Client*. URL: <https://www.nuget.org/packages/OmniSharp.Extensions.LanguageClient/>. (Accessed: 15.04.2020).
- [41] *Nunit CollectionAssert*. URL: <https://github.com/nunit/docs/wiki/Collection-Assert>. (Accessed: 15.04.2020).
- [42] Marcel Hess and Thomas Kistler. *Usability Test with Remo Herzog*. HSR Hochschule für Technik Rapperswil, 2020.
- [43] *SonarCloud for C# Framework Project*. URL: <https://community.sonarsource.com/t/sonarcloud-for-c-framework-project/17132>. (Accessed: 23.03.2020).
- [44] *OpenCover*. URL: <https://github.com/OpenCover/opencover>. (Accessed: 23.03.2020).
- [45] *monocov*. URL: <https://github.com/mono/monocov>. (Accessed: 23.03.2020).
- [46] *dotCover*. URL: <https://www.jetbrains.com/de-de/dotcover/>. (Accessed: 23.09.2019).
- [47] *ESLint*. URL: <https://eslint.org>. (Accessed: 30.05.2020).
- [48] *Dafny, Warnings Issue*. URL: <https://github.com/dafny-lang/dafny/issues/168>. (Accessed: 26.05.2020).
- [49] *Dafny for Visual Studio Code (Preview Release)*. URL: <https://marketplace.visualstudio.com/items?itemName=correctnesslab.dafny-vscode-preview&ssr=false#overview>. (Accessed: 31.05.2020).
- [50] *Dafny lang, GitHub*. URL: <https://github.com/dafny-lang/dafny>. (Accessed: 30.05.2020).
- [51] *Dafny for Visual Studio Code (Preview Release), GitHub CI*. URL: <https://github.com/DafnyVSCode/ide-vscode/blob/master/.github/workflows/ci.yaml>. (Accessed: 31.05.2020).



- [52] *GitLab Client, Language Server String Resources*. URL: <https://gitlab.dev.ifs.hsr.ch/dafny-ba/dafny-vscode-plugin/-/blob/master/src/stringRessources/languageServer.ts>. (Accessed: 30.05.2020).
- [53] *GitLab API, Get Job Artifacts*. URL: <https://gitlab.dev.ifs.hsr.ch/dafny-ba/dafny-vscode-plugin/-/blob/master/src/stringRessources/languageServer.ts>. (Accessed: 30.05.2020).
- [54] *GitLab Pages*. URL: <https://docs.gitlab.com/ee/user/project/pages/>. (Accessed: 30.05.2020).
- [55] *SonarCloud for Old TypeScript Language Server*. URL: [https://sonarcloud.io/code?id=dafny-vscode\\_1337&selected=dafny-vscode\\_1337%3Aserver%2Fsrc](https://sonarcloud.io/code?id=dafny-vscode_1337&selected=dafny-vscode_1337%3Aserver%2Fsrc). (Accessed: 09.04.2020).
- [56] *SonarCloud for Old Client*. URL: <https://sonarcloud.io/code?id=ide-vscode&selected=ide-vscode%3Asrc>. (Accessed: 07.06.2020).
- [57] *SonarCloud for New Client*. URL: <https://sonarcloud.io/code?id=dafny-plugin&selected=dafny-plugin%3Asrc>. (Accessed: 07.06.2020).
- [58] Daniel Keller. *Software Engineering Lecture, HSR*. HSR Hochschule für Technik Rapperswil, 2018.
- [59] Marcel Hess and Thomas Kistler. *Project Plan*. HSR Hochschule für Technik Rapperswil, 2020.
- [60] *Redmine*. URL: <https://redmine.wuza.ch>. (Accessed: 05.06.2020).
- [61] *SonarLint for Visual Studio Code*. URL: <https://marketplace.visualstudio.com/items?itemName=SonarSource.sonarlint-vscode>. (Accessed: 27.05.2020).
- [62] *SonarCloud*. URL: <https://sonarcloud.io/organizations/hsr/projects>. (Accessed: 02.06.2020).



## Project: Enhancing Dafny Support in Visual Studio Code

### Developer Documentation

Marcel Hess  
Thomas Kistler

Supervisors:  
Thomas Corbat  
Fabian Hauser

## Table of Contents

1. Overview.....	4
2. Introductory Tutorials .....	5
2.1 VSCode Extensions.....	5
2.2 OmniSharp .....	5
3. Setup of the Development Environment .....	6
3.1 IDEs .....	6
3.1.1 Automatic Code Formatting on Saving.....	6
3.2 Repositories .....	7
3.3 Client.....	7
3.3.1 Packages Dependencies .....	7
3.3.2 Updating Packages .....	7
3.3.3 Starting the Client.....	7
3.3.4 Selecting the Proper Output View.....	7
3.4 Server .....	8
3.4.1 Microsoft Boogie .....	8
3.4.2 Z3 .....	9
3.4.3 Visual Studio Solution Overview .....	10
3.4.4 Missing References.....	11
3.4.5 Testing your project setup.....	11
3.5 Folder Structure .....	12
3.6 Download of the Language Server.....	13
3.6.1 Correct Versioning.....	13
3.6.2 Create Folder Alias .....	13
4. Debugging.....	14
4.1 Client Side .....	14
4.2 Server Side .....	14
4.2.1 Using ReAttach .....	15
4.2.2 Using Debug.Launch().....	15
5. Relevant Code Information .....	16
5.1 Target Framework.....	16
5.2 Stream Redirection .....	16
6. Testing .....	17
6.1 Client Tests.....	17
6.2 Running Server Tests .....	17
6.2.1 From Inside Visual Studio .....	17
6.2.2 From Console.....	17
6.3 Creating Server Tests .....	18



---

6.4 Test Folder Structure .....	19
6.5 Concept of Writing Isolated Unit Tests.....	20
7. Continuous Integration(CI) .....	21
7.1 Local Testing With Docker .....	21
7.2 Updating Versions.....	22
7.3 Prebuild Stage .....	22
7.4 SonarScanner .....	22
7.5 Adjusting the Sonar Token.....	22
8. References.....	23

## 1. Overview

This document will help you getting started to work on the Visual Studio Code Dafny plugin and the Dafny language server.

The project consists of two main parts as you can see in Figure 1. On the one hand, there is the Visual Studio Code plugin, the so called “client”. It just contains a very basic level of logic and is mainly responsible for displaying information to the user.

On the other hand, there is the Visual Studio solution, called the “server”. It delivers the required information to the client using the language server protocol (LSP).

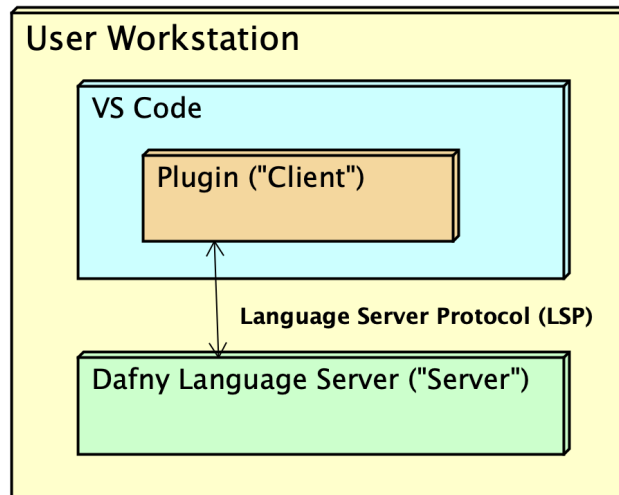


Figure 1 - Architecture

Although the plugin part is called “client” and the language server “server”, please note that both instances are run on the user's local workstation.

## 2. Introductory Tutorials

This chapter presents two rather simple, but extremely helpful tutorials to get familiar with the problem domain. The first tutorial is about the creation of Visual Studio Code extensions. The second one is from OmniSharp and shows how to use their implementation of the language server protocol. Both together are an optimal preparation for this project.

### 2.1 VSCode Extensions

To understand how one develops a Visual Studio Code extension, we can recommend the tutorials provided on the official site from Visual Studio Code [1]. The “Your First Extension” tutorial is very simple but you will get familiar with all important files, classes and concepts for developing an extension. Further on that site, you will find more advanced information like what kind of programmatic language features are possible with the Visual Studio Code API.

### 2.2 OmniSharp

OmniSharp offers the Language Server Protocol implementation for C#. Instead of starting with our project, you may first want to have a look at a more basic example of an OmniSharp implementation. “Creating a language server using .NET” is a very well-suited tutorial for this matter [2]. It gives a nice introduction on how LSP requests are handled. Our implementation follows the same style used in the tutorial. We think it is extraordinary helpful for your understanding.

If you need further help with LSP and OmniSharp, please visit their Slack community [3]. Questions asked in the slack channel are usually answered very quickly if kept concise.

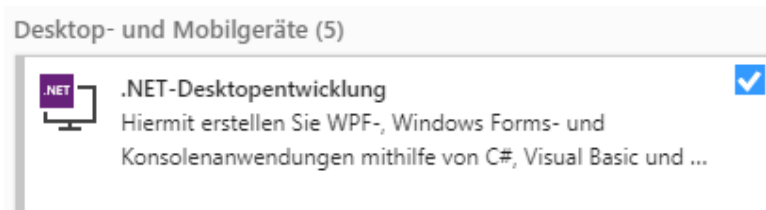
### 3. Setup of the Development Environment

This chapter will provide you with a detailed introduction how to set up all necessary repositories and files, so that you can start to contribute to the project. The chapter is separated into two parts. Firstly, we will have a look at the client side which is rather simple. To work on the client, one uses Visual Studio Code itself as an IDE. Secondly, the server side, which is quite strenuous to set up, is discussed. Some hints for CI your configuration are also given. Make sure to follow our instructions carefully. The recommended IDE for C# is Visual Studio.

#### 3.1 IDEs

You may use the following two IDEs for this project. To work on the client, you best use Visual Studio Code, which is free and easy to install. Make sure to set the "PATH" Option during install, so you easily launch it by typing "code ." into a console window. There are also options to add VSCode to the context menu on windows, which is also a nice to have.

For developing the server, use Visual Studio 2019. The community edition is sufficient. Make sure to have at least the following package installed:



Also ensure under "Single Components" that the following packages are installed:

- SDK for .NET Framework 4.6.1
- Package to compile into target version .NET 4.6.1.
- MSBuild

Nice to have and strongly recommended are also the following packages:

- NuGet Package Manager
- NuGet Target and Build Tasks
- Git Integration
- Code Analysis Tools

If you face an issue that some projects - namely the test projects - need to be migrated, don't do so but make sure the above packages are all installed.

As always with Visual Studio, we would recommend to work with ReSharper from JetBrains, too.

#### 3.1.1 Automatic Code Formatting on Saving

Once in a while, it may happen that a bracket accidentally gets a space too small or a line break too large. To avoid such formatting errors and to have a consistent formatted code, we recommend installing the following two plugins: Prettier[4] for VSCode and "Format document on Save"[5] for Visual Studio. Prettier will only format code. For static code analyze we recommend SonarLint [6].

Prettier uses the formatting role in the Prettier configuration file [4]. Therefore it formats correctly after installation out of the box. The visual Studio Plugin uses the project settings of visual studio itself.

## 3.2 Repositories

For a nice start, you best create a project folder and then clone the git repository for the client as well for the server into it [7]. The server should then be in a subfolder `dafny-language-server`, and the client in a subfolder `dafny-vscode-plugin`.

## 3.3 Client

This chapter describes the commissioning of the client development environment.

### 3.3.1 Packages Dependencies

Before you are able to run the client, you have to run the command `npm install` within a console inside the client folder manually. Of course one needs to have `npm` installed on his machine for that. This installs all packages the client requires. Dependencies are defined in the `package.json` file.

### 3.3.2 Updating Packages

From time to time it is appropriate to update packages to the latest version. We recommend to do this especially for each further development of the project. This is especially important to close security holes with the latest package versions. This is relatively easy to do. With the following command the packages are updated: `npm update` [8]. Afterwards you may be asked to run an `npm audit fix` to check the project for vulnerabilities [9].

### 3.3.3 Starting the Client

After you have cloned our repositories, open the folder `dafny-vscode-plugin` in the client repository with Visual Studio Code. To do so, you may just open a terminal and type `code .`. You can also create a batch file for a quick access to the client in Visual Studio Code [10] or use the context menu listing if that was installed.

Find and open a random `.ts` file, for example the main file `src/extension.ts` and press `F5`. This will launch the plugin in a virtual test environment. It can be used like it would be installed and you can set breakpoints and debug the code as well. Once your virtual environment is running you have to open a `.dfy` file. Otherwise the plugin will not start since the plugin listens only to Dafny files.

You should then see a notification that the server could not be found. This is because we have not built it yet, but the message ensures that the client is running properly so far. We will discuss the server building in chapter 3.4 - Server.

### 3.3.4 Selecting the Proper Output View

On the bottom of the screen, the console output is displayed. Often, a random window is shown. However, you are interested in the console output of "Dafny Language Server". You may have to manually choose this output window as shown in the figure below. Think of that whenever you get no console output.

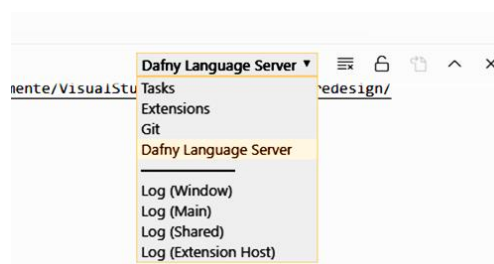


Figure 2 - Set VSCode Console Output

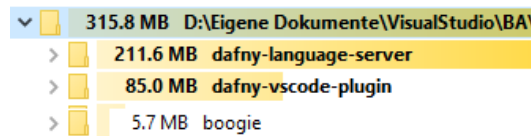


## 3.4 Server

Setting up the language server is not as easy as just cloning the repository. Dafny has a bunch of dependencies which you have to provide manually. This is rather inconvenient for developers and the issue has been reported to our supervisors. Hopefully, the setup can be made easier in the future. For now, please follow the upcoming steps carefully. Please note that the root directory in the following steps is always the created “parent folder” that contains both cloned git repositories and not the server subfolder.

### 3.4.1 Microsoft Boogie

First of all, you need to provide binaries from Microsoft Boogie [11]. Dafny references these, thus you need to provide them. They need to be inside a folder boogie/Binaries/ within the root folder of your project.



To acquire Boogie, you can either build it yourself, or what we'd recommend, just use the included Binaries within the original Dafny repository.

Currently, Dafny requires Version 2.4.2. Check yourself what the current required version is.

If you want to build them yourself, you have to clone the repository using the command  
`clone --branch v2.4.2 https://github.com/boogie-org/boogie.git`

This should clone the correct Boogie version onto your system. Now you have to open the Solution file from Boogie and then, you can build the project. This should create all binaries inside the folder “boogie/Binaries”. Make sure to hit “Rebuild”, not just “Build”.

A problem that may occur is that one file, “Main.resx”, is marked as not trustworthy under Windows. Visual Studio will give you a detailed error message. To resolve this, just locate the file and allow access to it.

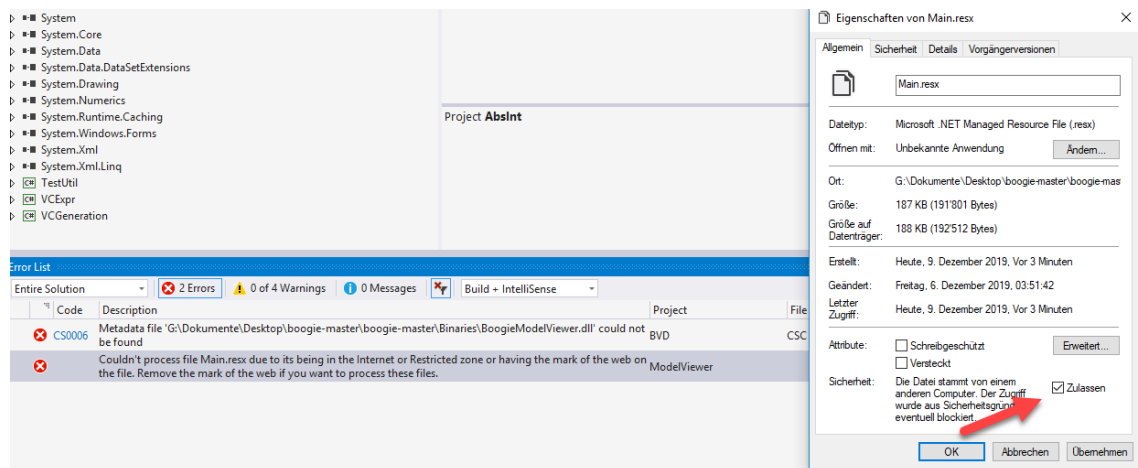


Figure 3 - Main.resx Permissions

The build should cover 21 projects and create about 40 files inside the binaries folder. Each project should have an associated .dll and .pdb file, as well as Boogie.exe itself. After building, all you need to keep is the content of the boogie/Binaries folder. Be aware that aside the .dll and .exe files, you have to keep the associated .pdb files. These refer to system libraries, such as System.Collections or such whenever a collection is used.

The other folders are technically no longer needed, but in case you want to rebuild Boogie, they may come in handy.

To provide Boogie inside your CI environment, you can as well clone the repository and build it. Our Docker file contained the following three self-explanatory lines for this purpose.

```
RUN git clone --branch ${BOOGIE_RELEASE} https://github.com/boogie-org/boogie.git &&\
    msbuild boogie/Source/Boogie.sln
ENV PATH=$PATH:/opt/boogie/Binaries
```

Whereas BOOGIE\_RELEASE is currently set to

```
ARG BOOGIE_RELEASE=v2.4.2
```

If you encounter any problems during this process, Boogie has a short readme in their repository which may help you [4].

### 3.4.2 Z3

Z3.exe is a prover from Microsoft. Dafny - to be more precise: Boogie - is making use of this prover for its purposes. Dafny expects Z3.exe to be inside the dafny/Binaries/ folder. However, this exe file will not get built during the process. It is an external dependency and you have to provide the file manually. Thus, make sure that Z3.exe is located inside dafny/Binaries.

While you could clone the Z3 repository [12] and build it yourself, the process is rather inconvenient if you are not familiar with the suggested build tools like nmake. You can also simply download the the release from the release subfolder in the repository [13]. Currently, Dafny requires Version 4.8.4. Again check yourself which Version is suggested by Dafny.

Take note that you also have to provide Z3.exe inside your CI environment. For example, we had to provide Z3 in our docker environment by downloading the file from the above repository and unzip it with the following series of commands:

```
RUN wget --no-verbose ${Z3_RELEASE} &&\
    unzip z3*.zip &&\
    rm *.zip &&\
    mv z3* z3
ENV PATH=$PATH:/opt/z3

ARG Z3_RELEASE=https://github.com/Z3Prover/z3/releases/download/z3-4.8.4/z3-4.8.4.d6df51951f4c-x64-ubuntu-14.04.zip
```

First, the z3 release is downloaded and then unzipped. Afterwards, the zip gets deleted and everything starting with z3 (z3.exe is what we want here) is moved into the z3 directory. This directory is then provided as an environment path variable.

### 3.4.3 Visual Studio Solution Overview

Once Boogie and z3 are installed, you are ready to open the solution “Dafny.sln”. It consists of two major folders, Dafny and LanguageServer. Projects you find inside the Dafny folder correspond to the existing Dafny solution [14]. Within the LanguageServer, you'll find our main project DafnyLanguageServer. Last, there are Test folders, which contains unit and integration tests.

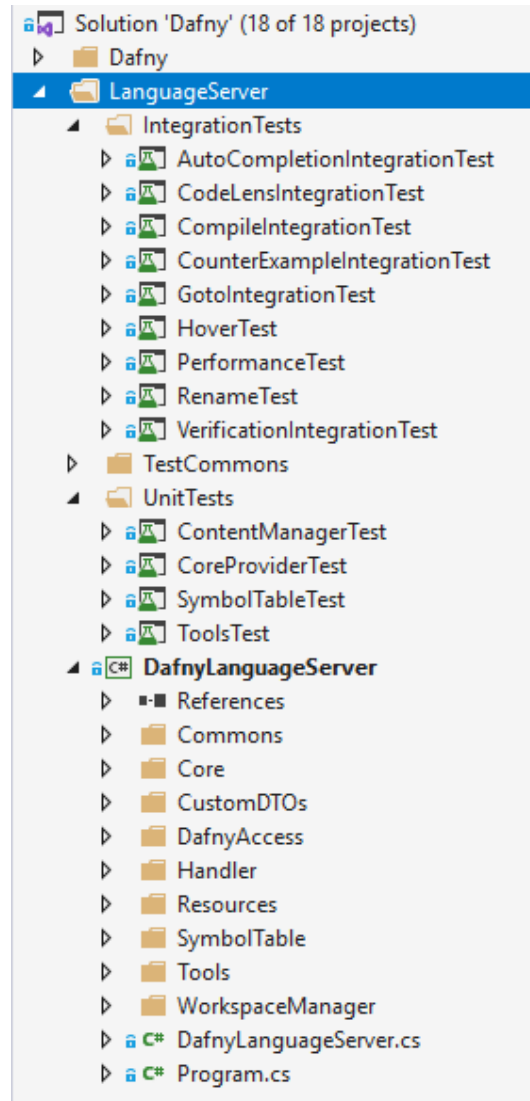


Figure 4 - Dafny Solution Overview

### 3.4.4 Missing References

If you have warnings containing missing references in your solution, make sure the folder structure is correct according to chapter 3.5 - Folder Structure. It is also possible to manually refer to the corresponding Boogie dlls that you have built in the prior chapter. To clean up a missing reference, right click on “References”, click “Browse” and then you can select the proper dlls, such as Provers.SMTLib.dll for example. This is shown in Figure 5 below.

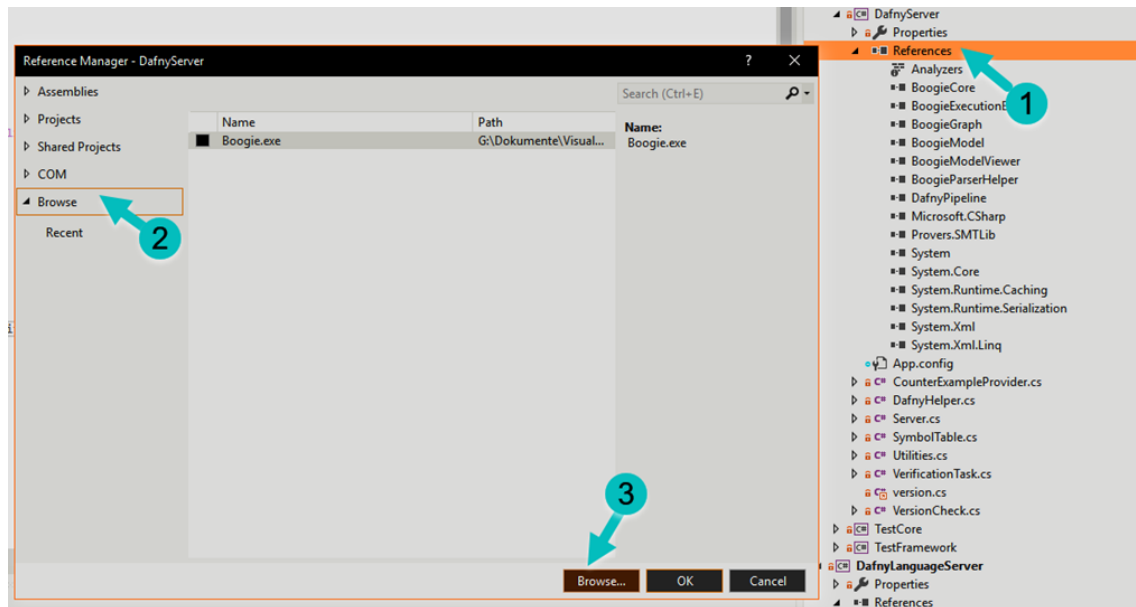


Figure 5 - Setting References

References are stored in each .csproj file. You may want to inspect these with a text editor if things go wrong. Once you have all references correctly set, the solution should build. Nuget Packages will be downloaded automatically if not available yet.

If Visual Studio complains about NUnit references, please refer to chapter 6.3 Creating Server Tests. This is just a capitalization issue and should cause no trouble.

### 3.4.5 Testing your project setup

If you want to perform an isolated test to check if your project setup is working correctly without using the client side, you may just want to run all provided tests inside the solution. Since there are integration tests using the complete infrastructure including Z3, they are a suitable indicator if your setup is working properly.

You can also just try to start the server yourself. Simply hit F5 inside Visual Studio. Make sure the language server is set as startup project. Maybe the launch causes already an exception, which for example happens on missing references.

### 3.5 Folder Structure

Make sure you provide the correct folder structure, so that you do not have to manually update any references. Inside your root folder, you should have a folder “boogie”, which contains its binaries in the subfolder “boogie/Binaries”. As well inside your root folder, you need to have a “dafny-language-server” folder containing the Dafny project with its own “Source” and “Binaries” folder as you can see in Figure 6 below.

The repository we are working on covers the necessary folder structure, but you may want to use newer distributions from Boogie. If things go wrong, you can also use the installation instructions from the “dafny-lang” [15] and “boogie” [11] repositories.

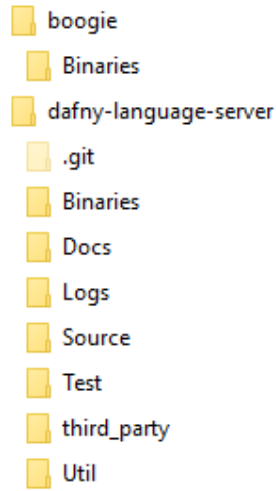


Figure 6 - Folder Structure

## 3.6 Download of the Language Server

The Visual Studio Code Marketplace contains only the client part when it is published.

When initializing the plugin the client automatically downloads the latest Dafny Language Server version if it has not already been downloaded. The following two things must be considered: Correct versioning to provide and adapt the local development environment for comfortable development.

### 3.6.1 Correct Versioning

There are basically two types of version numbering. One for the plugin itself, which is located in the client's `package.json` [16].

The server version, the relevant version for the download, is currently indicated in two parts. Please always adapt all versions. One is in the server in the resource files (`DafnyLanguageServer/Resources/VersionInformation.resx`). Secondly, in the string resources of the client for the server components (`src/stringResources/LanguageServer.ts`). The latter version is only a temporary solution.

Currently our changes have not yet been published on Github. Therefore we temporarily obtain the language server as a ZIP file from our own server.

Once our changes have been integrated into the official project, the client's temporary `languageServerInstaller` can be replaced by the original `languageServerInstaller` [17]. For the interchangeability we have written a separate interface. Only minimal adjustments to the newly given method names should be necessary.

### 3.6.2 Create Folder Alias

To avoid having to republish and download every change during the development of the server, you can create an alias to the binaries of the server.

The server is loaded into the out directory of the client (`dafny-vscode-plugin/out/dafnyServer`). The contained folders "Binaries" and "Config" have to be replaced to appropriate folder references (`dafny-language-server/Binaries` and `dafny-language-server/Config`). Under Windows, a softlink can be created using the following console command. The sourcefolder `out/dafnyServer` must be deleted first for the command to work:

```
mklink /J
  "X:[...]\dafny-vscode-plugin\out\dafnyLanguageServer"
  "X:[...]\dafny-language-server"
```

Please note that the corresponding directories only exist after a build of the Language Server on the server side and on the client side only after a first plugin start.

Alternatively, the path to the executable of the language server can be adjusted in the client settings [16]. This change should not be committed to git. Otherwise the server will no longer be downloaded into the local plugin directory during production.

## 4. Debugging

Debugging is of major importance for a quick and efficient development. Since we struggled with it at first, we decided to show you in a detailed fashion how you can debug a language server project. The client is hereby not the problem. The main difficulty is how to debug the server while the client is running. As we will see, one can simply attach the debugger to the executing process. Alternative methods that we tried are also presented, including the reasons why we do not recommend them.

### 4.1 Client Side

As mentioned in chapter 2.1 VSCode Extensions, your client will automatically be in debug mode once you start it with F5. Client debugging should be quite straightforward.

### 4.2 Server Side

The client will launch the language server. Once it has started, you can attach the Visual Studio debugger to the process that started the language server. Be aware that not Visual Studio Code is starting the server, but a dedicated launch process. In a dotnet core project, one would write “dotnet DafnyLanguageServer.dll” and thus, the executing process would be dotnet. If you are in a Linux environment and you want to launch DafnyLanguageServer.exe, you probably write “mono DafnyLanguageServer.exe”, and thus “mono” is the executing process. However, under a Windows environment you simply start the executable with “DafnyLanguageServer.exe”, and the executing process is then “DafnyLanguageServer”. How the server-process is started is defined inside the file VSCodePlugin/src/server/dafnyLanguageClient.ts. For now, we assume that we started DafnyLanguageServer.exe directly.

To be able to debug, your binaries must be up to date. You may want to always build the solution prior to debugging. Once the language server is running, select Debug -> Attach and choose the process as mentioned above and shown in Figure 7.

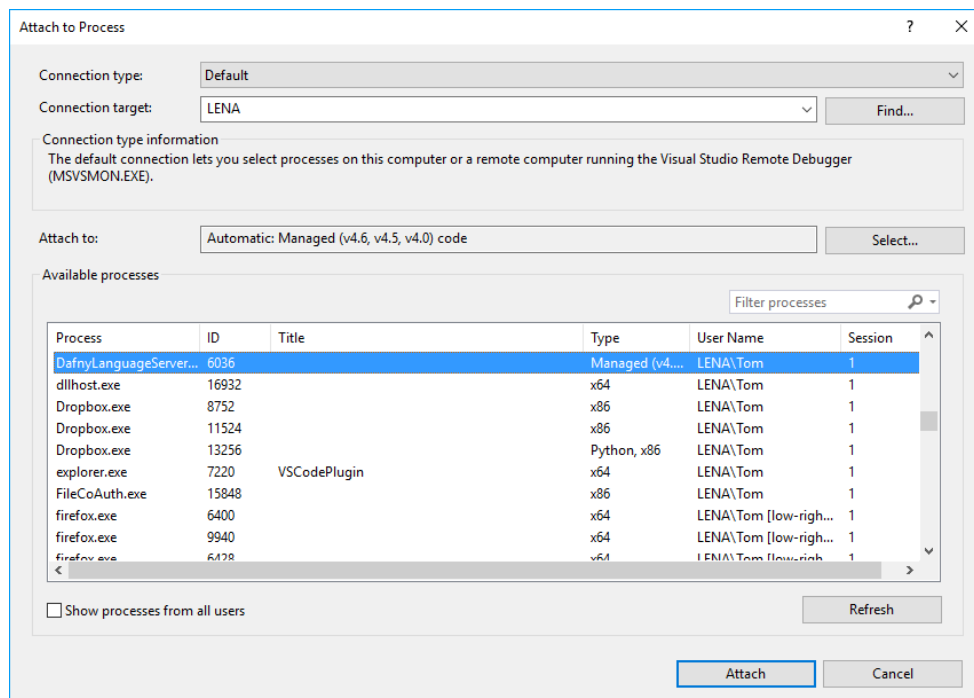


Figure 7 - Attach Debugger to a Process

Visual Studio will afterwards switch to debug mode and you can debug as usual.

### 4.2.1 Using ReAttach

You notice yourself that many clicks are necessary to attach the debugger every single time. Thus, we recommend to use the Visual Studio Code extension ReAttach [18].

It allows you to attach the debugger using a single click, or rather a single keyboard shortcut. If your client is not ready yet, it will even wait until the selected process has started. This is a very convenient feature.

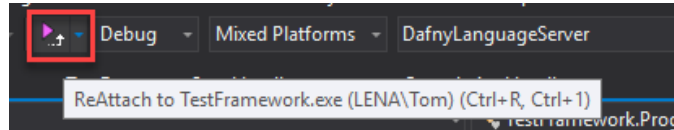


Figure 8 - ReAttach in Visual Studio

### 4.2.2 Using Debug.Launch()

At first, we just used `Debug.Launch()`, a .NET Core system method. This will also start the debugger but you will always be prompted if you want to launch a new Visual Studio instance. Afterwards, you have to wait until the project and the debugger have loaded, which takes quite some time. Thus, this method is strongly discouraged. The best way to debug is using ReAttach.



## 5. Relevant Code Information

This chapter states two important facts about the code, which are not obvious. The first subchapter is about why we have chosen to target .NET Framework instead .NET Core. The second subchapter is about the redirection of the console output stream.

### 5.1 Target Framework

Please note that all projects are targeting .NET Framework. While it would be favorable to use .NET Core for platform independence, not all used dependencies support the core framework. If you refer a .NET Framework project from inside a .NET Core project, system libraries will be unavailable. Thus, we decided to code in .NET Framework as well. The chosen version is .NET Framework 4.6.1. Note that Dafny was updated to .NET Framework 4.8 recently. You may adapt this change.

### 5.2 Stream Redirection

If you take a look into the source code, you will notice that we redirect the output stream into a file. This has two reasons. First and most importantly, deeper layers from Boogie sometimes print output. We do not want side effects like these. Such console outputs can ruin the client-server connection because those logs are not valid LSP formatted outputs. Secondly, the created log can be useful for debugging in case that you develop a new feature that gets no debug information dumped by the Visual Studio Code side.

## 6. Testing

This chapter gives useful hints on how to run all provided tests.

### 6.1 Client Tests

There are no client tests anymore. All end-to-end system tests have been removed and are completely replaced by our server side integration tests.

### 6.2 Running Server Tests

This chapter describes how to run server tests. Of interest is primarily the second subchapter, in which we describe how you can easily run our tests in a console-only environment, namely with your CI server.

#### 6.2.1 From Inside Visual Studio

As you most likely know, server tests can directly be run from within Visual Studio. You may use Visual Studio's own test runner or the one from ReSharper, whichever is more in your favor.

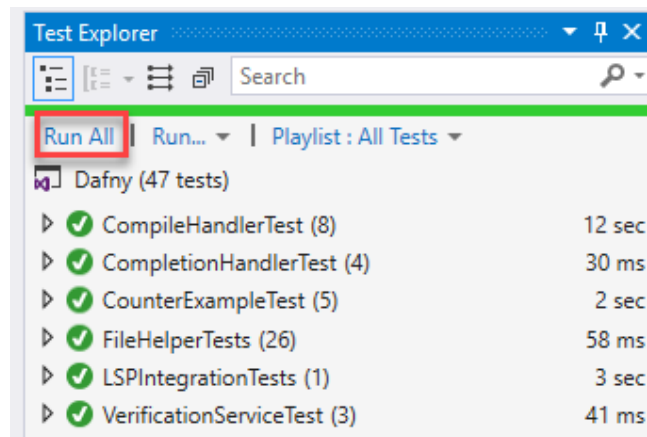


Figure 9 - Running Tests From Inside Visual Studio

You can access intergration as well as unit tests this way.

#### 6.2.2 From Console

If you have to start the tests from a console, you can use the NUnit Console Test Runner. It is available as a NuGet Package named "nunit.console.runner". Using NuGet, the installation is simple. The corresponding command is

```
nuget install nunit.console.runner
```

Afterwards, just launch the NUnit runner and provide the test-project-dll as an argument. The test dlls are as well located in `dafny/Binaries/`. A call could look like this:

```
$ ./NUnit.ConsoleRunner.3.11.0/tools/nunit3-console.exe MyTest.dll
```

Note that when working with Linux, you can launch the same runner using Mono.

## 6.3 Creating Server Tests

We decided to use nUnit as our testing framework. This way, we are independent of `mstest`, which is not as simple to install on a Linux CI environment. With nUnit, we can run the tests easily from within Visual Studio, but also from within the CI server.

However, when you create a new nUnit test project as shown in Figure 10, you may notice that these target only .NET Core by default.

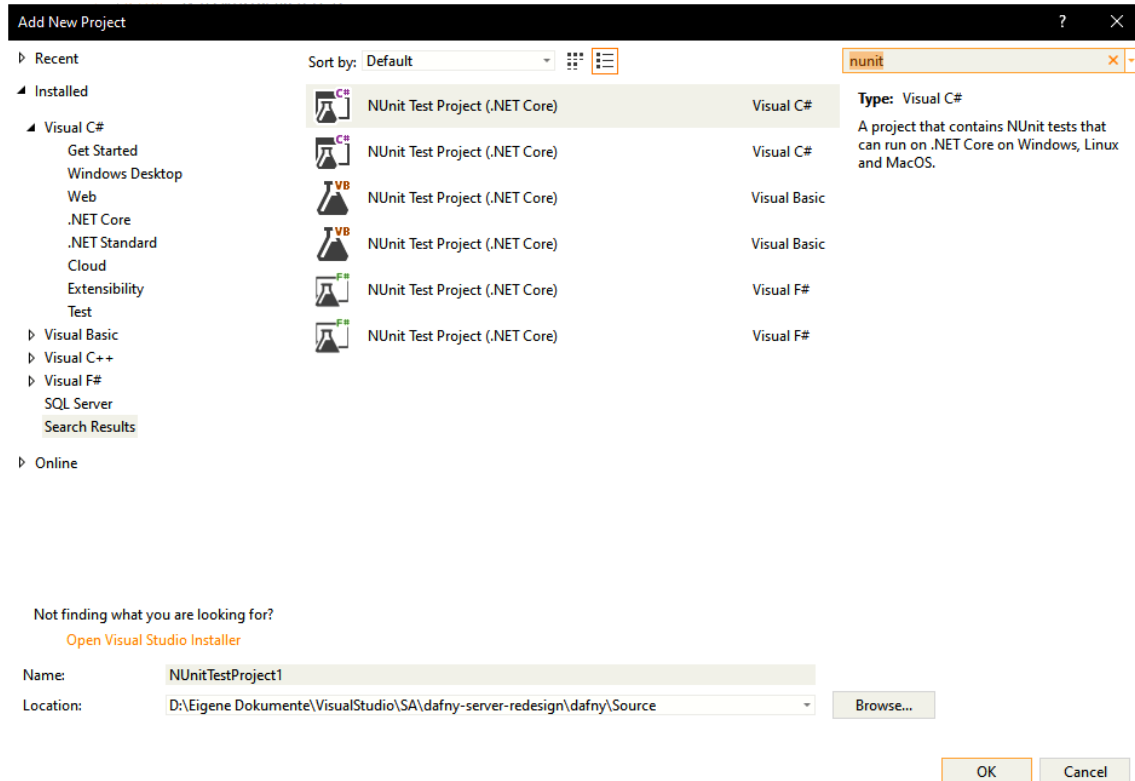


Figure 10 - NUnit Test Project Wizard

Just create the project as a .NET Core project. Locate the project inside the

*dafny-language-server/Test/LanguageServerTest/[Integration|Unit]Tests*

folder and follow the existing folder structure. Note that your Project must end on *\*Test.csproj* to be detected as a test project by the CI. After creating the nUnit project, close Visual Studio and navigate to the just created project file. Open it with a text editor and change the targeted framework to “net461”. This is a bit rigorous, but works totally fine. You may also want to adjust the output path of the binaries. An example .csproj file is shown below. You may just use it as a template.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net461</TargetFramework>
    <OutputPath>../../../../Binaries/</OutputPath>
    <AppendTargetFrameworkToOutputPath>>false</AppendTargetFrameworkToOutputPath>
    <AppendRuntimeIdentifierToOutputPath>>false</AppendRuntimeIdentifierToOutputPath>
    <IsPackable>>false</IsPackable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="nunit" Version="3.11.0" />
    <PackageReference Include="NUnit3TestAdapter" Version="3.11.0" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.9.0" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\..\..\Source\DafnyLanguageServer\DafnyLanguageServer.csproj" />
    <ProjectReference Include="..\..\TestCommons\TestCommons.csproj" />
  </ItemGroup>
</Project>
```

Sometimes, Visual Studio will complain that “NUnit” is not a valid package and that it cannot resolve this reference. This should not cause any trouble, but if in doubt try to rename the reference to “nUnit” with a lower n.

If you encounter a problem that the test runner is not executing test properly, you probably mixed up target frameworks within your test project. Make sure they are all equal, and especially don't mix .NET Core and .NET Framework as target frameworks.

## 6.4 Test Folder Structure

Our nUnit tests are placed within the *dafny-language-server/Test/LanguageServerTest* folder. There is one subfolder for integration test project, one for unit tests, one for TestCommons and one that contains Dafny-testfiles.

## 6.5 Concept of Writing Isolated Unit Tests

Most of our classes are programmed against an interface. This means that you can easily create fake-instances for testing. For example, to create a fake symbol table manager, you just have to implement the proper interface:

```
public class FakeSymbolManager : ISymbolTableManager
{
    ...
}
```

Now you can use your fake and inject it into everything depending on it, namely a core-provider. A complete example is shown below.

```
[Test]
public void ReservedWord()
{
    ISymbolTableManager manager = new FakeSymbolManager();
    var provider = new RenameProvider(manager);
    var result = provider.GetRenameChanges(...);

    Assert.IsTrue(provider.Outcome.Error, "error expected in rename-outcome");
}
```

If you cannot write simple unit tests for your code because of heavy dependencies, you can probably refactor your code and use dependency injection. Once you use proper interfaces, it should be easy to write a mock or a fake that allows writing isolated unit tests with ease.

## 7. Continuous Integration(CI)

Our CI is already linked to GitLab. In case one develops further changes, it works out of the box. In case you wish to reconfigure for your own GitLab repository, you will find helpful hints in this chapter. The CI for the client as well for the server are independent. The concepts and tips described below can be applied to both projects.

We do not have an continuous delivery (CD) for our plugin to the VSCode plugin marketplace. For a detailed description of the used GitLab stages, please follow the main document.

### 7.1 Local Testing With Docker

Whenever you change something essential in your Dockerfile or the `.gitlab-ci.yml`, you probably want to test it first on your local machine. To do so, you simply have to install Docker in case you have not done yet.

In case you would like to test if a changed Dockerfile would result in a successfully built container, you simply run this command in your root folder of the Dafny Language Server or Client root:  
`docker image build .`

In case you would like to connect to the already built container of the ci process, you can download the prepared docker container with the following `docker run` command.

Please note that you have to authenticate yourself the very first time. To do so, execute this the following login command (username is your GitLab username, token is your generated access token):  
`docker login <GitLabURL>: -u <username> -p <token>`

```
docker run
  -v /absolutePathToTheServerOnYourMachine:/mountedFolderInDockerContainer
  -ti --rm <GitLabURL>:latest /bin/bash
```

Whereas `<GitLabURL>` can be found out as follows: In GitLab go to the menu: Package - Container Registry. There is the URL to your built Docker image.  
E.g.: `gitlab.dev.ifs.hsr.ch:45023/dafny-ba/dafny-language-server/dafny-language-server-build`

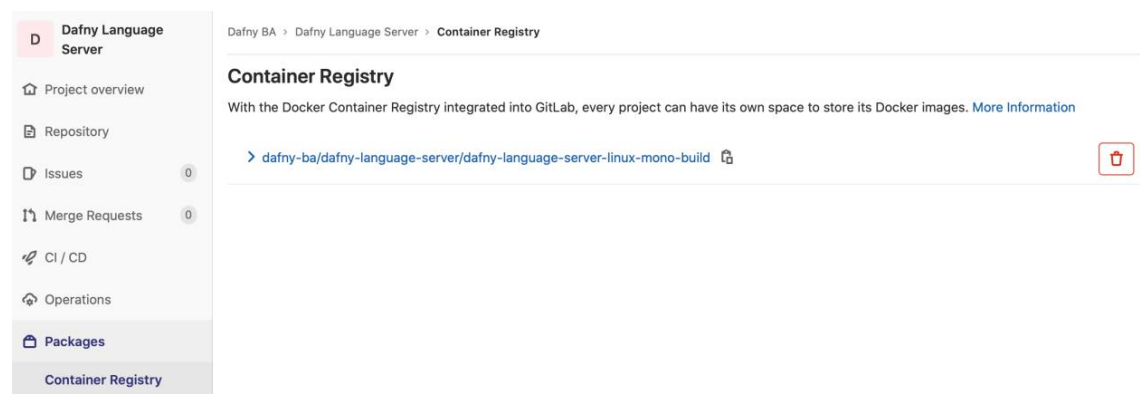


Figure 11 - Docker Container Registry in GitLab

Please note that the docker container does not contain any code from the git repository since this gets mounted on every CI process. To mount your project source code into the docker container, you have to add the `-v` option.

## 7.2 Updating Versions

The Docker image includes several third party dependencies like Z3 for the Dafny Language Server and Node for the client. Both projects contain an individual Sonar scanner. You can find them in the Dockerfile file. They are listed at the very beginning of the file as ARGs as shown in the following code snippet. The versions are clearly recognizable and easy to change.

```
ARG NODE_VERSION=10.16.3
ARG Z3_RELEASE=https://github.com/Z3Prover/z3/releases/download/z3-4.8.4/z3-4.8.4.d6df51951f4c-x64-ubuntu-14.04.zip
ARG GO_RELEASE=go1.10.3.linux-amd64.tar.gz
ARG SonarScanner_RELEASE=3.0.3.778
```

Please note that when the Dockerfile file changes, the next CI pipeline will take longer since the prebuild stage for building the docker container will be triggered again.

## 7.3 Prebuild Stage

As mentioned in the previous chapter, the prebuild stage runs every time the Dockerfile file is changed. The same applies whenever the yaml configuration file is edited. This is defined by the changes tag in the following snippet from `.gitlab-ci.yml`:

```
build_image:
  stage: prebuild
  image: docker:latest
  only:
    refs:
      - master
    changes:
      - Dockerfile.build
      - .gitlab-ci.yml
```

As you can notice, there is also a refs tag in the only section. It means that the docker image gets only built when the changes happen on the master branch. If you would like to change this behavior, you have to update the `.gitlab-ci.yml` file.

## 7.4 SonarScanner

Different types of sonar scanners are required for the Server and the Client. The reason for this is discussed in our main document. The scanner for TypeScript in the client can be run through independently as the last CI phase. Please note that this is not possible for the server. The server needs a Sonar scanner for MSBuild This will override the actual build process of msbuild. Accordingly, the analysis must not be outsourced to an independent CI process, as otherwise the necessary information cannot be collected by the scanner.

## 7.5 Adjusting the Sonar Token

If you would like to link the repository to another SonarCloud project for code metrics, you have to change the SONAR\_TOKEN in GitLab. To do so go to Settings > CI/CD and expand "Variables". Now you can add or change the environment variable as shown in Figure 12 below. To generate a new token, please follow the SonarCloud user guide [19].

The client and the server needs both a own, separated SonarCloud project.

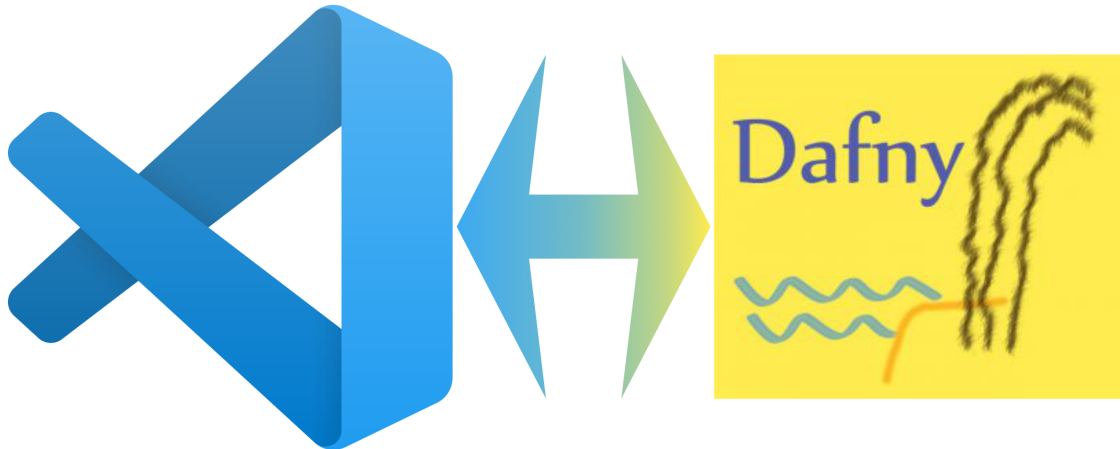


Figure 12 - Add SONAR\_TOKEN as Environment Variable

## 8. References

- [1] “Extension API.” <https://code.visualstudio.com/api/index> (accessed Oct. 14, 2019).
- [2] “Martin Björkström - Creating a language server using .NET.” <http://martinbjorkstrom.com/posts/2018-11-29-creating-a-language-server> (accessed Oct. 14, 2019).
- [3] “OmniSharp community on Slack.” <https://omnisharp.herokuapp.com/> (accessed Oct. 14, 2019).
- [4] “Prettier - Code formatter - Visual Studio Marketplace.” <https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode> (accessed May 13, 2020).
- [5] “Format document on Save - Visual Studio Marketplace.” <https://marketplace.visualstudio.com/items?itemName=mynkow.FormatdocumentonSave> (accessed May 13, 2020).
- [6] “SonarLint - Visual Studio Marketplace.” <https://marketplace.visualstudio.com/items?itemName=SonarSource.sonarlint-vscode> (accessed May 28, 2020).
- [7] “Dafny SA / Dafny\_Server\_Redesign,” *GitLab*. <https://gitlab.dev.ifs.hsr.ch/dafny-sa/dafny-server-redesign> (accessed Dec. 11, 2019).
- [8] “npm-update | npm Documentation.” <https://docs.npmjs.com/cli-commands/update.html> (accessed May 13, 2020).
- [9] “npm-audit | npm Documentation.” <https://docs.npmjs.com/cli/audit> (accessed May 13, 2020).
- [10] “launch\_VSCode.bat,” *GitLab*. [https://gitlab.dev.ifs.hsr.ch/dafny-sa/dafny-server-redesign/blob/master/VSCoDePlugin/\\_launch\\_VSCode.bat](https://gitlab.dev.ifs.hsr.ch/dafny-sa/dafny-server-redesign/blob/master/VSCoDePlugin/_launch_VSCode.bat) (accessed Dec. 11, 2019).
- [11] “boogie-org/boogie,” Oct. 20, 2019. <https://github.com/boogie-org/boogie> (accessed Oct. 22, 2019).
- [12] “Z3Prover/z3,” Dec. 11, 2019. <https://github.com/Z3Prover/z3> (accessed Dec. 11, 2019).
- [13] “Z3Prover Releases,” *GitHub*. <https://github.com/Z3Prover/z3> (accessed Dec. 11, 2019).
- [14] “dafny-lang/dafny,” Oct. 11, 2019. <https://github.com/dafny-lang/dafny> (accessed Oct. 14, 2019).
- [15] “dafny-lang installation,” *GitHub*. <https://github.com/dafny-lang/dafny/wiki/INSTALL> (accessed Oct. 28, 2019).
- [16] “package.json · master · Dafny BA / Dafny VSCode Plugin,” *GitLab*. <https://gitlab.dev.ifs.hsr.ch/dafny-ba/dafny-vscode-plugin/-/blob/master/package.json> (accessed May 26, 2020).
- [17] “Dafny VSCode Plugin / dafnyLanguageServerStartup,” *GitLab*. <https://gitlab.dev.ifs.hsr.ch/dafny-ba/dafny-vscode-plugin/-/tree/master/src/dafnyLanguageServerStartup> (accessed May 26, 2020).
- [18] “ReAttach - Visual Studio Marketplace.” <https://marketplace.visualstudio.com/items?itemName=ErlandR.ReAttach> (accessed Oct. 22, 2019).
- [19] “User Token | SonarCloud Docs.” <https://sonarcloud.io/documentation/user-guide/user-token/> (accessed Dec. 11, 2019).





## Project: Enhancing Dafny Support in Visual Studio Code

### Project Plan

Authors:  
Marcel Hess  
Thomas Kistler

Supervisors:  
Thomas Corbat  
Fabian Hauser

Third Reader:  
Olaf Zimmermann

Expert:  
Guido Zraggen

## Table of Contents

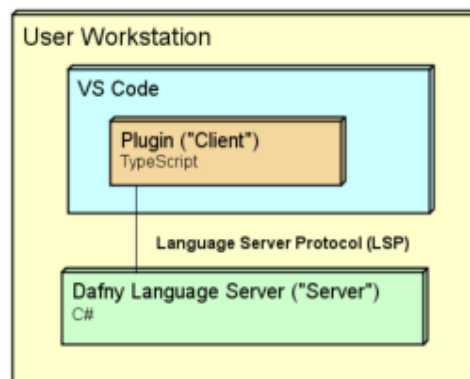
1. Project Overview .....	3
1.1 Initial Situation .....	3
1.2 Goal .....	4
1.3 Scope of Delivery .....	4
2. Organization .....	5
2.1 Time Budget .....	5
2.2 Process Management .....	5
2.3 Meetings .....	5
3. Schedule and Scope .....	6
3.1 Project Staging .....	7
3.2 Milestones .....	7
3.2.1 Inception .....	8
3.2.2 Infrastructure .....	8
3.2.3 Improvements .....	8
3.2.4 Symbol Table Manager .....	8
3.2.5 Extensions .....	9
3.2.6 Transition .....	9
3.2.7 Conclusion .....	9
3.3 Additional Potential Extensions .....	9
3.4 Work Packages .....	10
4. Infrastructure .....	11
4.1 CI .....	11
4.2 CD .....	11
5. Quality Aspects .....	12
5.1 Code Reviews .....	12
5.2 Code Style Guidelines .....	12
6. Testing .....	13
6.1.1 Unit Testing .....	13
6.1.2 Integration Testing .....	13
6.1.3 System Tests .....	13
7. References .....	14

## 1. Project Overview

During fall semester 2019/2020, Thomas Kistler and Marcel Hess worked on the term project «Dafny Server Redesign». [1] They will now continue this work by the bachelor thesis at hand. Aside the improvement of existing features, new and more complex features can be joined.

### 1.1 Initial Situation

The preceding project «Dafny Server Redesign» included the creation of a language server for Dafny, the integration with the Dafny library itself, as well as reworking the client for VSCode. The final architecture is shown in the following figure. [1]



**Figure 1 - Architectural Overview**

As one can perceive, there is only one component «Server» which includes the Dafny library as well as the language server.

The coded feature set of the language server included, but is not limited to:

- Syntax Highlighting
- Highlighting Logical Errors
- Underlining Violated Promises
- Providing Counter Examples
- Auto Complete Suggestions
- Code Compilation and Execution
- Displaying Status Bar Information

The preceding thesis itself based on the bachelor thesis by Markus Schaden and Rafael Krucker. [2]

## 1.2 Goal

The goal of this bachelor thesis is the enhancement of the preceding term project. During the first part of the project, existing features shall be improved in functionality, reliability, maintainability and usability. Some of the features imply flaws which shall be corrected. The second part includes coding new features. Due to our personal interests we would like to implement a component to manage the symbol table of the user's Dafny code. This component can then be used for easier handling of existing features, and they can then be further improved.

## 1.3 Scope of Delivery

The final report will at least consist out of the following items:

- Abstract
- Management summary
- Technical report
- Developer documentation
- Project plan
- Personal reports
- Poster
- Legal declarations
- Code in digital form

The thesis will be handed in according to the internal guidelines of HSR.

## 2. Organization

This chapter covers the most important organizational aspects of the project in a concise way. It lists the time budget, that SCRUM will be used and when ordinary meetings take place.

### 2.1 Time Budget

The total working hours for the bachelor thesis is defined to be 12 ECTS or 360 hours per student. The project lasts for 17 weeks, which averages out in a nominal working time of 21.2 hours per week per student. During the last two weeks, no other classes take place and more time resources will be available. Furthermore, this calculation does not include the one-week Easter break.

### 2.2 Process Management

During the construction phase, we will use SCRUM for an agile development. At the beginning of a sprint, the sprint will be planned and each of the two developers can choose tasks to work on. All occurring tasks will be tracked with tickets in Redmine. Redmine's ticket base forms the product backlog. After the sprint has ended, a sprint review will take place to review the achieved work. At the end of a sprint, a short retrospective will be held to revise procedural aspects, too. There won't be daily reviews, since there are only two team members which can exchange information at any time. The sprints will last for one week only to be as flexible as possible. They will start and end on Friday.

### 2.3 Meetings

The weekly meetings with our supervisors Thomas Corbat and Fabian Hauser will be held each Thursday at 10:30am.

Thomas Kistler and Marcel Hess have a reserved time frame of 3x8h from Wednesday to Friday each week to work on the project. Whenever necessary, internal meetings can be held within this time frame.

### 3. Schedule and Scope

The following list provides the reader with an overview of all features that are outlined in this project plan. A more detailed discussion will be held throughout this chapter.

#### Improvement of existing features

- Syntax Highlighting
- Verification
- Autocompletion
- Counter Example
- GoTo Definition
- Compilation

#### New but simple features

- Hover Information

#### New major features

- CodeLens
- Refactorings
- Automated Contract Generation

#### Functionality for publishing

- Easy installation
- Cross Platform Usage

A nice feature addendum would be debugging, too. Due to its complexity, it will not be considered and may be subject to a dedicated thesis project in the future.

During the preceding term project, first attempts to implement CodeLens have already been made. The implementation is rather inefficient and should be improved. [1] To be able to do so, we got the idea to implement a component that manages the symbol table. This component is strongly connected to the abstract syntax tree (AST) provided by existing Dafny code. The AST can be used to create the symbol table. This component would also ease the implementation of many other features, such as GoTo Definition and Autocompletion. Thus, it wouldn't make much sense to improve features such as GoTo Definition first, and then implement the symbol table afterwards. This circumstance will be considered during scheduling the work load.

The reader will notice that the following schedule is very tight. It may not be possible to realize every idea mentioned in this chapter. Since the time is fixed, the scope has to be variable. Thus, the students will dynamically adjust the scope according to agile project management methods.

To be able to supply a deployable product in the end, the existing code has to be improved, ways for testing have to be found, complex concepts such as a symbol table managing component have to be added. To be able to achieve all of that and not get lost in unimportant details, the project is split into multiple stages which are time boxed. This ensures that we can work on all fields we would like to, so that we can learn as much as possible within this bachelor thesis.

### 3.1 Project Staging

In this project, we will use a modified variant of the unified process. This means that broadly speaking, the project is split into the following stages:

- Inception
- Infrastructure
- Construction
- Transition
- Conclusion

The inception phase is very short and just contains a few preliminary tasks. During the infrastructure phase, CI, automated testing and automated quality measures will be worked on. The transition stage includes making the plugin ready to publish. The last phase, conclusion, is meant to finish documentation and any open tasks left over from the preceding stages. The major part of the project will take place within the construction phase. This rather large stage is further divided into the improvement of existing features and the addition of new functionality. Details about the individual work packages can be found in the following chapters.

W 01	W 02	W 03	W 04	W 05	W 06	W 07	W 08	W 09	W 10	W 11	W 12	W 13	W 14	W 15	W 16	W 17
Inception		Infrastructure		Construction										Transition	Conclusion	
				Improvements		Symbol Table			Extensions							

Figure 2 - Project Staging

### 3.2 Milestones

This subchapter will give a brief overview of the project's milestones. They are defined according to the project stages. There is a milestone at the end of every phase.

	W 01	W 02	W 03	W 04	W 05	W 06	W 07	W 08	W 09	W 10	W 11	W 12	W 13	W 14	W 15	W 16	W 17
Inception	■	■															
Infrastructure			■	■													
Improvements					■	■											
Symbol Table							■	■	■	■							
Extensions											■	■	■	■			
Transition															■		
Conclusion																■	■

Figure 3 - Milestones

Note that this project will contain no elaboration stage, since most elaborating work had already been completed during the preceding project. There is no need to create a prototype, nor to evaluate new technologies for most aspects. Nevertheless, during the infrastructure stage, problems such as automated testing can be taken care of. [1]

### 3.2.1 Inception

This stage will only last for two weeks and contains the initial set up. This includes:

- Getting familiar with the project after Christmas break
- Defining the goals for the bachelor thesis
- Authoring the project plan
- Updating IDE's and other software
- Update frameworks and libraries, such as OmniSharp if applicable
- Creating the intermediary presentation

Because the expert's availability is limited, the intermediary presentation will be held at the beginning of the project to present the achievements of the preceding term project and is held at February, 27<sup>th</sup> 2020.

### 3.2.2 Infrastructure

This milestone contains revising the existing CI/CD environment. There are some leftovers from the preceding thesis for which improvements are necessary. [1] Aside solid integration tests, system testing shall be considered and SonarQube needs to be installed for all software parts. Testing and quality measures are further discussed in chapters 4 and 5.

### 3.2.3 Improvements

During the term project in the fall semester 19/20, a solid amount of features was implemented. However, some of them show flaws which limit their usability. [1] To increase the acceptance of the product, these features shall be improved during this stage. Features that require symbol table component to be available are not listed. Aside those, the following ideas are present:

- Verification
  - Show syntactical errors, too, not only logical errors
  - Smarter underlining for better visibility
- Counter Example
  - Simplify presentation
  - Automatic update when the user continues to type
- Compilation
  - Support for compilation arguments
  - Direkt Dafny/Boogie nutzen
- Improve Status Bar

### 3.2.4 Symbol Table Manager

The goal in this milestone is to implement a component that builds and manages the symbol table for the Dafny code. This would make many other features possible, such as renaming. Existing features such as CodeLens and GoTo Definition will also profit by this component. To acquire the required knowledge base, the students will attend the lecture "Compilerbau" by Prof. Dr. Luc Bläser at HSR.



### 3.2.5 Extensions

In the follow-up milestone, the symbol table manager shall be used at its glance for the following features and improvements:

- Autocompletion
  - Smarter suggestions, for example show only classes after the keyword «new»
  - Place cursor according to user's desire
  - Show placeholder arguments for methods and constructors
- GoTo Definition
  - Be more flexible to where the cursor position is
- CodeLens
  - Finish all work in progress
- Refactorings

### 3.2.6 Transition

Before the code will be frozen, we will invest one week in final work such as:

- Ensure easy installation
- Ensure cross platform compatibility

### 3.2.7 Conclusion

At the very end of the project, the documentation will be authored and completed, the final presentation has to be prepared and a poster will be created. The last two weeks function also as a buffer for unforeseen consequences.

## 3.3 Additional Potential Extensions

Depending on how quick progress will be made during construction phase, the following features can also be implemented:

- Display hover information
- Support for multiple files
- Auto contract generation

### 3.4 Work Packages

To coordinate our tickets and for time reporting, we will use Redmine. The first tickets have already been created:

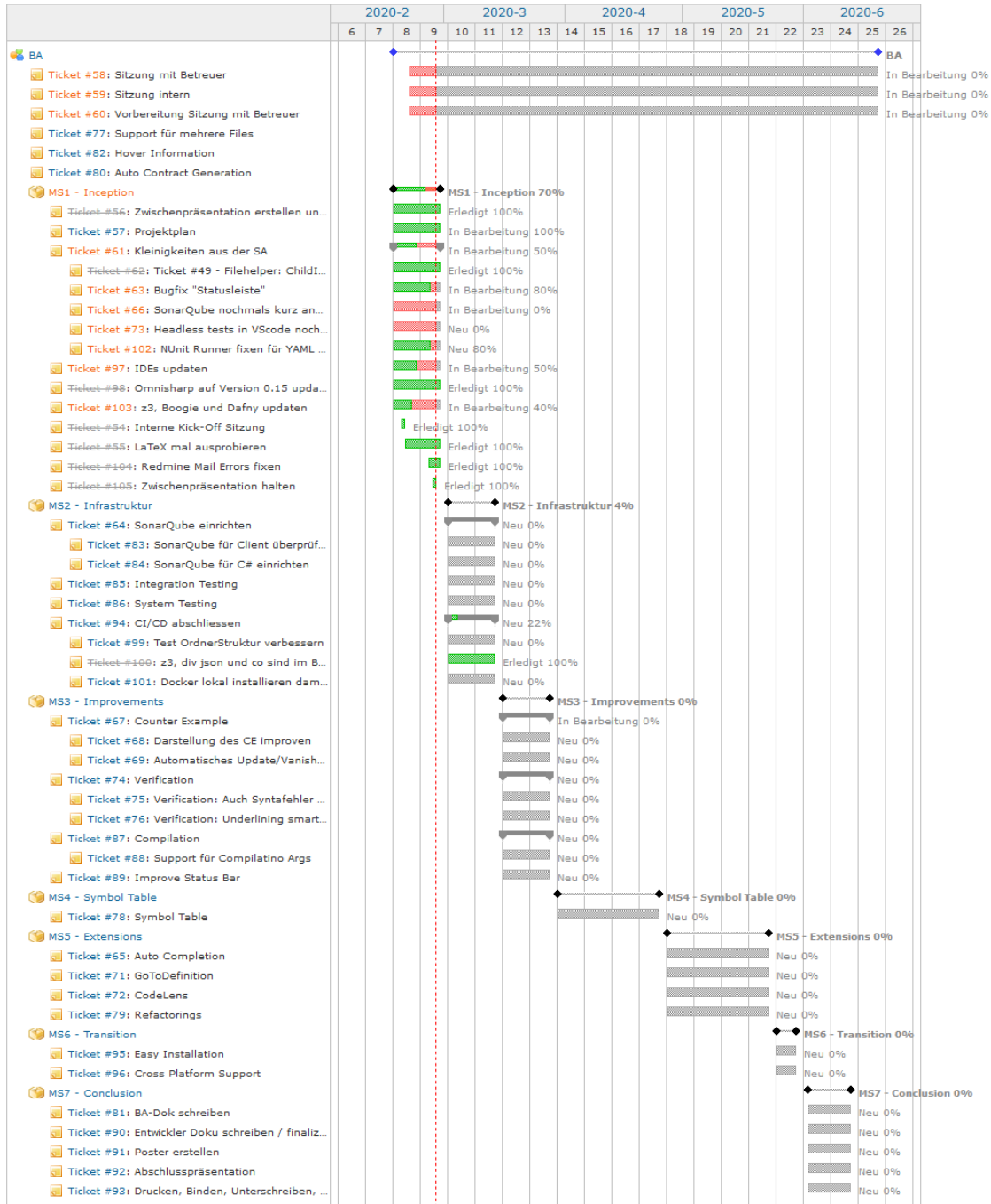


Figure 4 - Gantt Diagram

## 4. Infrastructure

Each member of the team will use his own notebook for the major part of the work.

The following technologies will be used during the research project:

Project planning and time reporting:	Redmine
Build server / CI server:	GitLab
Version control system:	Git
Host for Git:	GitLab
Runtime environment:	.NET Framework 4.6.1

IDEs:

- Microsoft Visual Studio
- Microsoft Visual Studio Code

Code quality and guidelines:

- ReSharper
- SonarQube

### 4.1 CI

The source code will be on GitLab. We will use GitLab's integrated pipeline system to compile the code, run the unit tests, and calculate quality metrics.

GitLab will notify all developers whenever a pipeline fails, for example if tests fail or the committed code is not compiling. Each member has to ensure that the pipelines run successfully.

### 4.2 CD

No continuous deployment will be used. The code base will be published only once at the end of development via the VSCode Marketplace.

## 5. Quality Aspects

This sub chapter covers quality aspects of this project, namely code reviews and style guidelines.

### 5.1 Code Reviews

Critical and difficult parts of the code shall underlie a code review. For this purpose, GitLab supports merge requests. We can use these each time a branch is merged into the master branch. We will especially keep in mind that not only one student works on a specific feature and the other student has no clue about it. Regular information and knowledge exchange will be enforced during the code reviews and sprint reviews.

### 5.2 Code Style Guidelines

We will apply the following coding guidelines:

- The naming of classes, methods and variables shall be in English only. Comments and the documentation shall also be in English.
- The code in C# shall basically fit the default guidelines of ReSharper and SonarQube. However, if we find that the readability of the code gets worse, exceptions and custom rules are allowed.

Generally, each developer has to ensure that the following guidelines are met:

- Good readability of the code
- Style guides as mentioned above redeemed
- Simplicity of the solution
- Documentation updated
- Proper tests written
- All tests are green

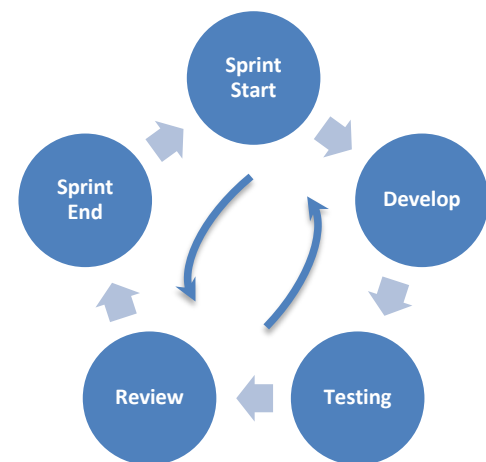


Figure 5 - Iterative Workflow

## 6. Testing

This chapter gives a brief overview of the planned testing infrastructure and guidelines. The chapter is divided into a section for each testing level.

### 6.1.1 Unit Testing

Simple classes, especially such that have none or few dependencies shall be properly unit tested. The tests shall be meaningful and their input shall cover different equivalence classes. We are going to use proper mocking to run automated unit tests on classes with dependencies, wherever possible. The targeted test coverage shall be at least 80%. However, if the goal is not met because trivial code paths such as constructors are not tested, no dummy tests for those code paths shall be added just to meet the goal.

The tests will be separated from the other parts of the project inside a specific project folder. The developers can run these unit tests directly inside Microsoft Visual Studio. The unit tests will also be automatically performed by the CI process each time a commit is made and pushed to GitLab using the existing NUnit runner setup from the previous thesis.

### 6.1.2 Integration Testing

It is planned to use Omnisharp's Language Client [3] to simulate communication between a client and the language server. The client can send all messages that the system under development supports and also receive the responses. Thus it is easy to test them for correctness.

This can be done directly inside Visual Studio in C#. Prior trials using this approach have shown that issues may arise, such as failures to parse the server's response. Those issues seem hard, or even unreal, to resolve since no obvious reason could be found. However, further investigation on this approach is worthwhile and shall take place. [1]

The advantage of using Omnisharp's client for testing is that it bases on the same technologies as the unit tests do. This would allow an easy integration into the CI system. Since they could be coded inside Visual Studio, we also had direct access to all methods and classes which makes writing tests very convenient. [1]

### 6.1.3 System Tests

The integration tests cover any code written in C#. The only code that is left to test is client code written in TypeScript. The preceding bachelor thesis used tests with a headless VSCode instance to test basic functionality of the client, which is very tedious, but necessary to test basic client functionality. [1]

## 7. References

- [1] Hess, Marcel and Kistler, Thomas, "Dafny Language Server Redesign," HSR Hochschule für Technik Rapperswil, 2019.
- [2] Krucker, Rafael and Schaden, Markus, "Visual Studio Code Integration for the Dafny Language and Program Verifier," HSR Hochschule für Technik Rapperswil, 2017.
- [3] "OmniSharp.Extensions.LanguageClient 0.15.0." [Online]. Available: <https://www.nuget.org/packages/OmniSharp.Extensions.LanguageClient/>. [Accessed: 26-Feb-2020].

Note: This document is based on the previous term project «Dafny Server Redesign».

# Angaben zur Person

Ort und Art der Durchführung: Vor Ort, Windows 10, Remetschwil  
Datum der Durchführung: 27.Mai.2020  
Test wird durchgeführt von: Marcel Hess

## Vorinterview

*Dient zur Einschätzung der Fähigkeiten des Probanden und klärt ihn über die Ziele und Absichten des Tests auf.*

### **Vorabinformationen:**

*Dem Probanden darüber informieren, dass dieser Test nicht ihn sondern unser Plugin testet. Er darf jederzeit unbeschwert allfällige Probleme äussern. Er kann in dem Sinne nichts falsch machen. Den Probanden grob mit unserer Arbeit vertraut machen und was das Ziel unseres Plugins ist. Grobinformation über Dafny selbst, und dass für den Test keine Dafny-Kenntnisse benötigt werden, da Codebeispiele zur Syntax bereitstehen. Für den Probanden werden grundsätzliche Programmierkenntnisse vorausgesetzt. In dieses Dokument hat nur der Testdurchführer Einsicht.*

Wie heisst du, wie alt bist du, programmierst du beruflich oder in deiner Freizeit?  
Remo Herzog, 25 Jahre, primär in der Freizeit.

Welche Programmier-/Skriptsprachen verwendest du am meisten?  
Ich verwende primär Python und selten JavaScript.

Weisst du was eine IDE ist? Kannst du mir grob dein Verständnis einer guten IDE schildern?  
Zum Beispiel VSCode. Eine Umgebung, in welcher man programmieren kann, und gleich alle nötigen Tools integriert sind.

Welche IDE verwendest du am häufigsten/liebsten?  
Ich arbeite fast ausschliesslich mit VSCodium auf Linux.  
Von der Bedienung ist es sozusagen dasselbe wie VSCode.

Welche Features einer IDE schätzt du besonders?  
Für mich ist das automatische Markieren von Fehlern eine essenzielle Funktion. Auch sehr wichtig für mich in Verbindung mit Python ist das direkte Ausführen eines Codeteils zum Prüfen, obs funktioniert wie gewollt. Dass die Entwicklungsumgebung ein modernes Aussehen hat ist auch kein unwesentlicher Bestandteil. Inklusive Darkmode versteht sich. Ein altmodischer Auftritt hat meiner Meinung nach beispielsweise Notepad++.

Hast du bisher schon mal etwas sehr störendes an einer IDE erlebt? Wenn ja, was?  
Bei Python gibt es eine Umgebung, mit welcher man Pakete installiert. Codium erkennt das leider nicht richtig. So kann ich als Entwickler nicht einfach «run» klicken und die Pakete werden installiert, sondern ich muss Manuel in der Konsole die Pakete installieren. Das empfinde ich als unnötigen Zusatzaufwand.



Hast du VSCode schon einmal verwendet? Für was/welche Sprachen? Positives Erlebnis?

Ja - sozusagen. Primär für Python.

Kennst du Dafny und hast allenfalls schon mal mit Dafny etwas programmiert?

Ich hab schonmal davon gehört, es aber selbst nie verwendet.

## Szenarien

**Vorbedingung:** VSCode und der Dafny Language Server sind gestartet und einsatzbereit. Die benötigte Dateien sind bereits im Workspace geöffnet. Andere Plugins von Dritten für VSCode sind deaktiviert.

### Aufgabe 1 – Zahl hochzählen

Du findest eine Datei «Aufgabe1\_Verwenden» und eine Datei «Aufgabe1\_NichtVerwenden». Bitte öffne die Datei «Aufgabe1\_Verwenden».

Darin findest du bereits ein kleines Dafny-Programm inklusive Einstiegspunkt (Main).

Im *Include* auf der ersten Zeile wird die Datei «Aufgabe1\_NichtVerwenden» eingebunden.

Bitte schau dir diese Datei NICHT an.

Finde heraus, was für ein Typ die Variable «number» hat und wo sie deklariert wurde.

*Ziel: Wird Hover Information genutzt?*

*«In Python hätte ich jetzt beispielsweise mit «type» gearbeitet und eine entsprechende Ausgabe getätigt.» Der Proband sucht nach Debugging. Ist etwas verwirrt. Kleine Hilfestellung, dass man nicht zu weit suchen muss.*

*Nun fährt er mit der Maus über die Klasse. Dies liefert zwar den Hover, aber nicht die gewünschten Informationen. Er erwartet hier eine Art **CodeDoc der Klasse**. Anschliessend fährt er über die Variable und er erhält die gesuchten Informationen.*

*Die vielen Informationen auf viel Raum verwirren etwas. **Weniger Text wäre wünschenswert**, eine bessere Gruppierung. Das Feld «Kind» verwirrt etwas. Ein Hinweis «was genau was macht bzw bedeutet» wäre hilfreich. Das liegt aber allenfalls auch an der fehlenden Vertrautheit zu Dafny. Etwas in Richtung einer Entwicklungsdokumentation wäre wünschenswert.*

*Ausserdem haben «Symbol» und «Declaration» zu viel Text. Weniger Text, höherer Informationsgehalt. Beispielsweise die Position in Klammern als reine Zahl*

Verwende die dir nicht weiter bekannte Klasse «Counter» des inkludierten Files um die Variable «Number» um eins hochzuzählen.

*Wird die Autocompletion verwendet? Ausreichende Hilfe? Bsp: `c.increase(c.number);`*

*Als erstes fährt der Proband wieder mit der Maus über «Counter» und «Number». Er erwartet eine Art Code Dokumentation, findet aber keine. Er fängt an zu tippen «c.». Nun ist er verwundert, dass keine Autocompletion auftaucht. **Nach einem Punkt erwartet er automatisch eine Autocompletion**. Er erhält den Hinweis, dass man aktuell Steuerung + Leerschlag drücken muss.*



Bei den vorgeschlagenen Methoden fehlen die Parameter. Das verwirrt etwas. Bei der Fehlermeldung (rot unterstrichen) würde die Information enthalten sein, dies fällt ihm aber nicht auf. Bessere Benutzerführung mit dem Augenfokus bei Fehlermeldungen wären allenfalls hilfreich.

Auch für die Behebung des fehlenden Semikolons wird ein Moment gebraucht. Er findet dies allerdings von alleine heraus und nicht mit Hilfe des Fehlertexts.

Bitte kompiliere dein Programm, führe es aus und überprüfe die Ausgabe.

*Wird «compile + run» entdeckt und verwendet?*

Oben rechts wird ein grüner «Play Button» gesucht. Er ist verwirrt, dass dieser nicht dort ist. (Gewohnheit von Python). Solch ein Button wäre hilfreich.

Als nächstes wird ein Rechtsklick versucht. Dort findet der Proband sofort die «Compile and Run» Option.

Die vielen Meldungen verwirren ihn etwas. «Was ist jetzt passiert.»  
Bei der Ausgabe wünscht er sich weniger Text. Der Pfad zum aktuellen Prozess sowie die Ausgabe der ausgeführten Executable sind zum Grossteil identisch. Das könnte man vereinfachen, dass nicht so viel Text ausgegeben wird. Sprich dass vom aktuellen Pfad ausgegangen wird. Das wäre ihm sympathischer.

### Szenario 2 – Codebereinigung

Öffne die Datei «Aufgabe2» und scroll in der Datei NICHT herunter. Du möchtest herausfinden, welche Klasse im oberen Teil der Datei wie oft verwendet wurde. Gibt es dead-code? Wo wurde die Klasse ClassA überall verwendet?

*Wird CodeLens verwendet? Wird die nicht verwendete Methode in ClassC entdeckt?*

CodeLens wird direkt verwendet, sofort gefunden. Der tote Code wird auch sofort erkannt.

Zusatzfragen im Gespräch:

Bei TypeScript werden die Referenzen aufgeteilt nach Scope. Dann gibt es beispielsweise die Auszeichnung «1 Reference | 1 Reference» wobei der erste Link die internen Referenzen (this) der Klasse und die zweiten Referenzen die Verwendungen von aussen (über Instanzvariablen) auszeichnen. Fändest du diese Aufteilung angenehmer?

*Nein. Ich will ja mit einem Klick gleich alle Referenzen sehen; egal über welchen Weg darauf zugegriffen wird. Im Popup sehe ich ja dann, was von wo kommt.*

Wenn eine Klasse oder Methode keine Referenz hat, wird diese mit dem Vermerk «wird noch nicht verwendet – kann dieser Code gelöscht werden?» ausgezeichnet. Bei Typescript wird beispielsweise einfach «0 Referenzen» ausgewiesen. Welche Variante bevorzugst du und warum?

*Ich finde das mit dem Text eigentlich besser. Es ist «mehr Text» und «anders» wodurch es sofort auffällt. Dead Code will man ja grundsätzlich nicht haben. Also darf das ruhig etwas ins Auge stechen.*



### Szenario 3 – Fehlerbehebung

Öffne die Datei «Aufgabe3». Jemand hat hier nicht sauber gearbeitet. Findest du den Fehler? Kannst du dir Beispiele anzeigen lassen, für welche konkreten Fälle die Funktion sich falsch verhalten wird?

*Wird CounterExample verwendet?*

Der Proband erkennt durch Überlegen sofort, dass «less» grösser sein könnte als «more». Dann verwendet er den Rechtsklick und geht Punkt für Punkt durch. «CounterExample» kennt er zwar nicht, hört sich aber passend an. Das gewünschte Ergebnis wurde erzielt.

Zusatzfrage im Gespräch:

Du versuchst jetzt grundsätzlich eher den Rechtsklick. Bei VSCode gäbe es ja noch die integrierte Eingabeaufforderung für Befehle. Dort hätten wir die Befehle auch hinterlegt. Verwendest du diese grundsätzlich weniger?

*Diese verwende ich eigentlich nie.*

Du möchtest nun die Variable «more» zu «bigger» umbenennen. Wie gehst du vor? Kannst du das automatisiert so umsetzen bitte?

*Wird Rename verwendet?*

Der Proband verwendet «Control + H» und ersetzt die Begriffe mit «Suchen und ersetzen». Zusätzlich aktiviert er die Option, nur ganze Wörter und nicht Wortbestandteile zu ersetzen. Nach dem Hinweis, dass dies in gewissen Fällen auch ungewollte Ersetzungen zur Folge haben kann, verwendet er erfolgreich den Rechtsklick und «Rename».

*Dannach verwendet er rechtsklick.*



## Nachinterview

Sind Probleme während der Tests aufgetreten?

Der Play Button oben rechts habe ich etwas vermisst. Bei der Completion erwarte ich Vorschläge sobald ich einen Punkt eingetippt habe.

Hast du etwas vermisst? Fehlte etwas bei den bisherigen Feature? Ein ganz anderes Feature?

Grundsätzlich die oberen beiden Funktionen. Ansonsten eigentlich nicht.

Hat dir etwas besonders gut gefallen?

Das Counter Example war sehr cool. Das ist sicher sehr praktisch.

Das Underlining ist sicher auch nützlich wenn was falsch ist. Falls die Meldungen kürzer und aussagekräftiger wären, würde ich es aber noch besser finden.

Empfandst du die Features als hilfreich? Würdest du sie verwenden? (Mehrwert)

Welches der Features findest du am nützlichsten? Warum?

Das Umbenennen ist noch überraschend praktisch (statt Suchen & Ersetzen).

Allgemeine Anmerkungen, Inputs was man sich noch wünschen würde:

Das Wichtigste fände ich das Autocompletion. Und falls Dokumentationsstrings für Klassen und Methoden hinterlegt werden könnten, fände ich das sehr nutzbringend.

---

## Usability Test Dafny Code

---

```
1 class Counter {
2     constructor(){}
3     function method increase(x: int): int {
4         x+1
5     }
6     var number : int;
7 }
8
9 method myUselessMethod() { /* do something */ }
```

---

Listing 44: Aufgabe1\_NichtVerwenden.dfy

---

```
1 include "Aufgabe1_NichtVerwenden.dfy"
2 method Main() {
3     var c := new Counter();
4     c.number := 1;
5     // todo: number um eins hochzaehlen
6     print(c.number);
7 }
```

---

Listing 45: Aufgabe1\_Verwenden.dfy

---

```
1 method MultipleReturns(inp1: int, inp2: int) returns (more: int, less: int)
2 {
3     more := inp1 + inp2;
4     less := inp1 - inp2;
5 }
6 class ClassA {
7     constructor () { }
8     method myMethod() { /* do something */ }
9 }
10
11 class ClassB {
12     constructor () { }
13     method myMethod() { /* do something */ }
14 }
15
16 class ClassC {
17     var ABC: int;
18     constructor () { }
19     method myMethod() { /* do something */ }
20 }
```

```
21
22 /*
23   Huge space. This part of the file the user must not see.
24 */
25 method Main() {
26   var myNumber := 1+2;
27   var myClassA := new ClassA();
28   myClassA.myMethod();
29   var myClassB1 := new ClassB();
30   var myClassB2 := new ClassB();
31   myClassB1.myMethod();
32   myClassB2.myMethod();
33   myClassB2.myMethod();
34
35   var more;
36   var less;
37   more, less := MultipleReturns(1,2);
38
39   var abc := new ClassC();
40   abc.ABC := 1;
41 }
```

---

Listing 46: Aufgabe2.dfy

```
1 method MultipleReturns(number1: int, number2: int) returns (more: int, less:
   int)
2   ensures less < number1 < more
3 {
4   more := number1 + number2;
5   less := number1 - number2;
6 }
```

---

Listing 47: Aufgabe3.dfy