

# **Dafny Language Server**

## **Bachelor Thesis**

Department of Computer Science  
University of Applied Science Rapperswil

Spring Term 2020

Authors: Marcel Hess, Thomas Kistler  
Advisors: Thomas Corbat (lecturer), Fabian Hauser (project assistant)  
Expert: Guido Zraggen (Google)  
Counter 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 ([tcorbat@hsr.ch](mailto:tcorbat@hsr.ch)) and Fabian Hauser ([fhauser@hsr.ch](mailto:fhauser@hsr.ch)), HSR, IFS.

### 2. External Examiner

- Guido Zraggen ([zraggen@gmail.com](mailto:zraggen@gmail.com)) - Google

### 3. Students

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

- Marcel Hess ([mhess@hsr.ch](mailto:mhess@hsr.ch))
- Thomas Kistler ([tkistler@hsr.ch](mailto:tkistler@hsr.ch))

### 4. Introduction

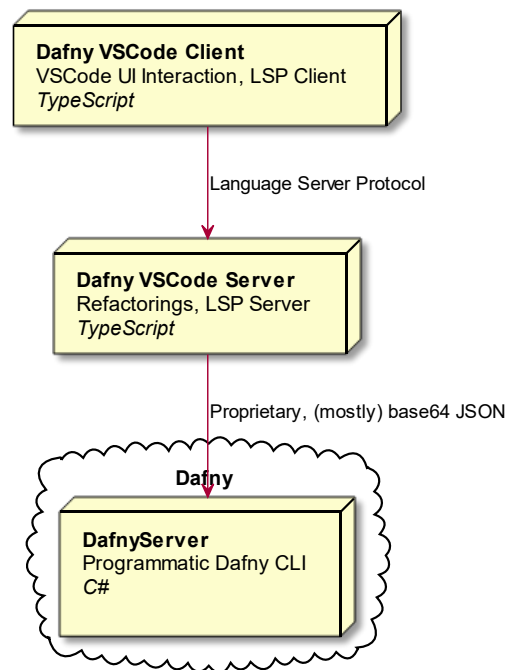
“Dafny is a programming language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs.

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 <a href="http://abstract.hsr.ch">abstract.hsr.ch</a>) Poster hand-in to SGI (until 10:00)</b>
<b>12.06.2020, 12.00</b>	<b>Final hand-in of the report through <a href="http://archiv-i.hsr.ch">archiv-i.hsr.ch</a></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 with preconditions, postconditions, loop invariants and loop variants. In a preceding bachelor thesis, a plugin for Visual Studio Code had been created to access Dafny-specific static analysis features. For example, if a postcondition is violated, the code will be highlighted and a counter example can be shown. Furthermore, it provides access to code compilation and auto completion suggestions.

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 a 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:

- Improvement of previously implemented features in usability, stability and reliability.
- Implementation of a symbol table to facilitate the development of symbol-oriented features like rename or auto completion.

The symbol table is required to contain information about each name segment in the code. It should allow direct access to a name segment's declaration, information about its scope and usage statistic. The symbol table opens a wide range for further development. For example, by knowing the depth of a symbol, indentation width can be determined for an auto formatting functionality.

## Result

Using the visitor pattern, the Dafny abstract syntax tree (AST) is visited to create the symbol table. Every symbol contains information about its parent, its children and its declaration. This makes navigation within the symbol hierarchy very simple. The implemented features go to definition, rename, code lens, hover information and auto completion can directly benefit by the symbol table and are no longer required to contain much logic. The offerings of the symbol table can be accessed for future extensions, such as auto formatting or code highlighting.

Aside features based on the symbol table, pre-existing functionality was revisited as well:

- The verification process was simplified by creating a dedicated Dafny translation unit.
- Unlike in prior versions of the plugin, warnings are now reported to the end-user as well.
- The verification results are buffered for efficient re-use upon compilation.
- Counter examples are simplified and better perceptive to the end user. They also remember their display setting per file.
- All features will now also work across multiple files and namespaces.

## Table of Contents

<b>1 Abstract</b>	<b>4</b>
<b>2 Management Summary</b>	<b>8</b>
2.1 Dafny	8
2.2 Language Server Protocol	8
2.3 Initial Solution	9
2.4 Goals	9
2.5 Results	10
2.6 Outlook	11
<b>3 Introduction</b>	<b>12</b>
3.1 Dafny	12
3.2 Language Server Protocol	13
3.3 Initial Solution	13
3.4 Goals	13
<b>4 Analysis</b>	<b>15</b>
4.1 Language Server Protocol	15
4.1.1 Message Types	15
4.1.2 Communication Example	16
4.1.3 Message Example	16
4.1.4 LSP Standard and Individual Extensions	17
4.2 OmniSharp	17
4.2.1 Basic OmniSharp Usage	17
4.2.2 Custom LSP Messages	18
4.3 Visual Studio Code Plugin	18
4.4 Dafny Language Features	18
4.4.1 Modules	19
4.4.2 Functions and Methods	20
4.4.3 Hiding	20
4.4.4 Overloading	21
4.4.5 Shadowing	21
4.5 Symbol Table	22
4.5.1 Requirements for the symbol table	23
4.5.2 Dafny Symbols	24
4.6 Visitor	25
4.7 Dafny Expression and Statement Types	25
4.7.1 Expressions	26
4.7.2 Statements	26
4.7.3 Declarations	27
4.8 Dafny AST Implementation	27
4.9 Continuous Integration (CI)	27
4.9.1 Initial Situation	28
4.9.2 Targeted Solution - Achieved Solution	28
4.9.3 Docker	28
4.9.4 2do - Kapitelaufteilung komisch	29
4.10 notizen von marcel	29
4.11 notizen von tom	29

<b>5</b>	<b>Design</b>	<b>30</b>
5.1	Technologies	30
5.2	Client	30
5.2.1	Initial Situation	30
5.2.2	New Architecture	31
5.2.3	Components	32
5.2.4	Logic	34
5.2.5	Types in TypeScript	35
5.3	Server	36
5.3.1	Dafny Translation Unit	37
5.3.2	Workspace Manager	37
5.3.3	SymbolTableManager	38
5.3.4	Core	39
5.3.5	Ressources	39
5.3.6	Tools	39
5.3.7	Overview	40
5.4	Integration Tests	41
5.4.1	Dafny Test Files	41
5.4.2	String Converters	41
5.4.3	Test Architecture	42
5.4.4	Performance Measurement	44
<b>6</b>	<b>Implementation</b>	<b>45</b>
6.1	Client	45
6.1.1	Client (Code Review)	45
6.2	Server	47
6.2.1	Server Launch	47
6.2.2	Tools	47
6.2.3	Handler	48
6.2.4	Workspace	49
6.2.5	DafnyAccess	50
6.3	Symbol Table	51
6.3.1	Core	56
6.4	Testing	58
6.4.1	Unit Tests	58
6.4.2	Integration Tests	59
6.5	Usability Test	60
<b>7</b>	<b>Results</b>	<b>61</b>
7.1	Accessibility	61
7.2	Intro	62
7.3	Sntax Highlighting	62
7.4	Verification	63
7.5	Compile	63
7.6	Counter Example	64
7.7	Hover Information	65
7.8	Goto Definition	65
7.9	Rename	66
7.10	CodeLens	66
7.11	Automatic Code Completion	66
7.12	Symbol Table	66
7.13	Platform Independence / Mono Support for macOS and Linux	67

7.14 Server Architectural Improvements . . . . .	67
<b>8 Conclusion</b>	<b>69</b>
8.1 Project Summary . . . . .	69
8.2 Deployment . . . . .	69
8.3 Outlook . . . . .	69
8.3.1 Completion of the Visitor . . . . .	69
8.3.2 LSP Extensions . . . . .	70
8.3.3 More Refactorings . . . . .	70
8.3.4 Dafny Specific . . . . .	71
8.3.5 More IDE . . . . .	71
8.4 Achieved Improvements compared to the preceding semester thesis . . . . .	71
8.5 Metrics . . . . .	71
<b>9 Project Management</b>	<b>72</b>
<b>10 Designation of chapters taken from the preexisting term project</b>	<b>73</b>
<b>Glossar</b>	<b>74</b>
<b>References</b>	<b>75</b>
<b>Anhang</b>	<b>76</b>



## 2 Management Summary

*non-technical summary von allem mit results* This chapter gives a brief overview of the bachelor thesis at hand. It is held in a non-technical language to be comprehensive for anyone.

### 2.1 Dafny

Dafny is formal programming language. Within the 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 proof postconditions. This means, given the precondition holds, the code will manipulate data only thus far, so that also the postcondition is valid.

The following code snippet 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 and computational assistance is welcome. The main advantage of pre- and postconditions is, that a programmer can be sure that the program is correct.

### 2.2 Language Server Protocol

The language server protocol (LSP) is a specification for communication. It regulates data exchange between an editor and a so called language server. The editor is used by the programmer to write code. The code is then transferred to the language server, using a format specified by the LSP. The language server is now responsible to analyze 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.

The separation between the editor and the server has the advantage, that multiple editors can use the same language server. Support for a programming language like Dafny can thus be achieved for multiple editors by just implementing the language server once.

The most important requests the LSP supports incorporate:

- Transfer code to server
- Show errors and warnings
- Rename a variable
- Go to the definition of a variable

- Show auto completion suggestions
- Perform a refactoring
- Show usages of a code item

These are features that are commonly used by programmers, independent of the language. Aside these, Dafny has the option to show a counter example, if a postcondition is violated. This request is not natively supported by Dafny. However, custom requests 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.

## 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 pre-existing solution 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.

In the preceding semester project [3], the language server was migrated to C# and could be integrated into the Dafny backend. Any Dafny functionality was directly accessible and the proprietary JSON-interface, as well as console parsing, could be omitted. All features were reimplemented to satisfy the new architectural layout.

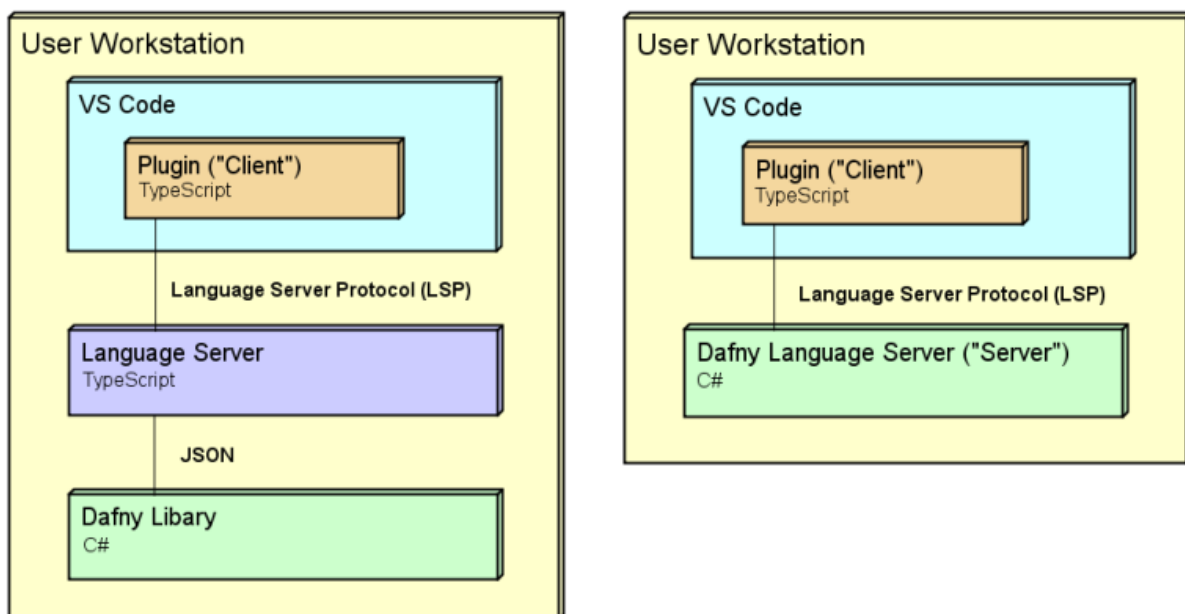


Figure 1: Architecture before (left) and after (right) the preceding term project

## 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

- Error highlighting
- Compile and run
- Auto completion

On the other hand, the plugin shall also be used in a professional environment. Thus, more advanced features have to be supported. To facilitate the development of further features, the code shall be refactored to achieve a clean state. This especially involves a clear architectural layout with dependencies that only point downwards.

Aside a clean architecture, a major goal is the creation of a custom symbol table. This is necessary to make features like *go to definition* or *auto completion* possible at all. It also allows the development of further features. For example, if every the symbol table provides information about the scope-depth, the indentation width can be deduced for a feature like auto formatting.

The symbol table shall further contain a well organized tree structure, so that navigation can be done efficiently. Each symbol shall have direct links to it's 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.

## 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 chapter.

The following features are supported by the server:

- Syntax highlighting
- Verification: highlighting of errors and warnings
- Compilation
- Show Counter Example
- Code Lens
- Auto Completion Suggestion
- Renaming
- Hover Information

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

- Compilation will no longer start a dedicated Dafny process for compilation. Instead, it will now use a buffered pre-compile generated by verification and directly call the Dafny's compile method, to generate an executable.
- Counter example representation is now much cleaner and easier to perceive.
- Verification will now also show warnings.
- Verification will no longer just highlight single characters.
- Code Lens can now actively be used to preview symbol usages.

- Auto completion no longer performs simple pattern matching, but provides exact results.

Renaming and hover information could newly be implemented due to the offerings of the symbol table. The user can now benefit by a plugin, which gained a lot of user experience, but also robustness compared to the initial solution.

Aside the improvements in features, many internal aspects were also improved. 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. All of these are properly handled, but in exchange to this quality attribute, not all Dafny language features are supported by the symbol table. For example, custom datatypes<sup>1</sup>, as used in formal programming languages, are not supported.

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.

Aside 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
- Create clients for other IDE's.

Aside the widening of the feature range, it is definitely 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. This was due to the limited time frame of the bachelor thesis. Nevertheless, the plugin is of a nice quality and can be deployed into the VSCode market place. Thus, future 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 bases on *Boogie*, which uses the Z3 automated theorem prover for discharging proof obligations [4]. That means, 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 the precondition, the postcondition are both 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.

The following code snippet 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 2: 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. Since the proof is done with formal, mathematical methods, the 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 afterwards 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.

---

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

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

---

Listing 3: Lemmas

### 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 auto completion, rename or go to definition. If the user performs an action like rename, the IDE will send the proper request to the language server. The message format is specified by the LSP and bases on JSON.

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 coded. 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.

### 3.3 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. Its development was continued within the HSR under the lead of Fabian Hauser. In the following text, the existing project will be referred 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.

Marcel Hess and Thomas Kistler continued the development of the project within a semester project [3]. The language server was migrated to C# and could be integrated into the Dafny backend. Any Dafny functionality was directly accessible and the proprietary JSON-interface, as well as console parsing, could be omitted. All features were reimplemented to satisfy the new architectural layout. The result of this semester thesis will be referred as the *prototype* throughout this thesis.

### 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

- Error highlighting
- Compile and run

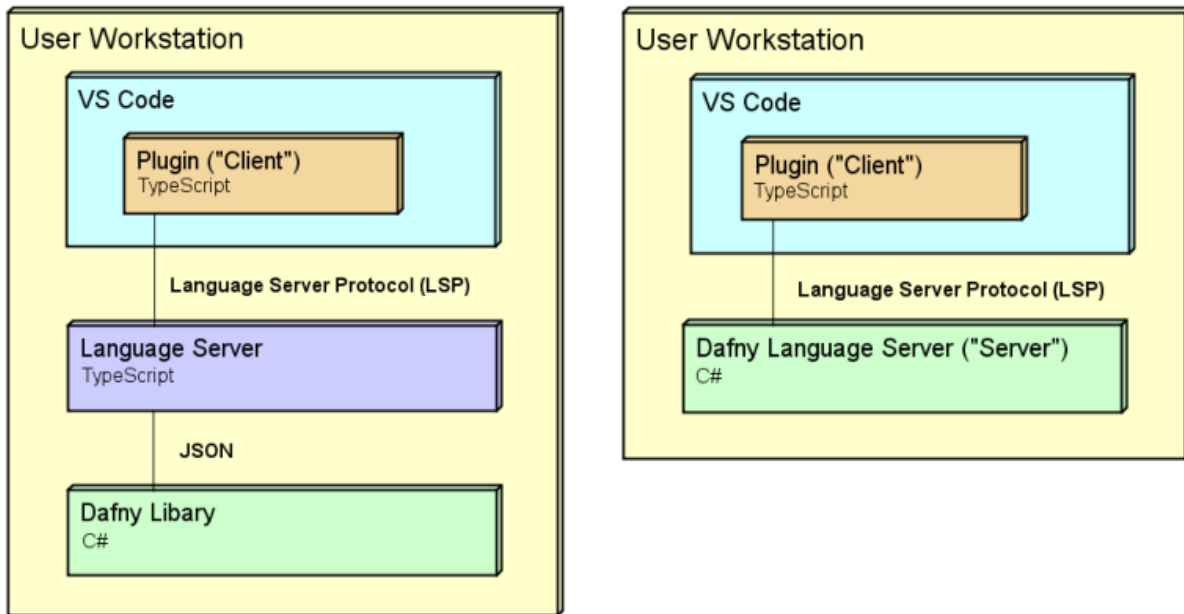


Figure 2: Architecture before (left) and after (right) the preceding term project

- Auto completion

On the other hand, the plugin shall also be used in a professional environment. Thus, more advanced features have to be supported. To facilitate the development of further features, the code shall be refactored to achieve a clean state. This especially involves a clear architectural layout with dependencies that only point downwards.

Aside a clean architecture, a major goal is the creation of a custom symbol table. This is necessary to make features like *go to definition* or *auto completion* possible at all. It also allows the development of further features. For example, if every the symbol table provides information about the scope-depth, the indentation width can be deduced for a feature like auto formatting.

The symbol table shall further contain a well organized tree structure, so that navigation can be done efficiently. Each symbol shall have direct links to it's 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.

## 4 Analysis

Since this thesis is a direct sequel of the preceding semester thesis, work could be directly continued. However, to provide the reader with a comprehensive knowledge base about Dafny and the language server protocol, some chapters out of the semester thesis will be repeated in the following subsections. To be able to create the symbol table, more detailed research about Dafny's language feature and its AST element had to be done, which is also described in this chapter.

### 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 [4]. In 2016, Microsoft started collaborating with Red Hat and Codenvy to standardize the protocol's specification [4]. 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 his 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 offers them. Those features can for example be auto completions, hover information, or go to definition. Custom message types, for example compile or counterExample can also be added to the LSP. [4] A big advantage of this is that the IDE specific plugin can be kept very simple. The relevant information is delivered by the language server, which is IDE and language independent. Figure 3 from the VSCode extension guide illustrates these benefits. [5]



Figure 3: Communication Benefit of LSP

#### 4.1.1 Message Types

The LSP supports three types of messages.

- Notification: One-way message, for example for a console log or 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.

#### 4.1.2 Communication Example

The basic concept of the lsp is, that the IDE tells the language server what the user is doing. These messages are pretty simple, namely `textDocument/didOpen` or `textDocument/didChange`. The language server on the other hand can now verify the opened or changed document and test it for errors. If errors are found, the server can send a `textDocument/publishDiagnostics` notification back to the client. The client may now underline the erroneous code range in red. [6]

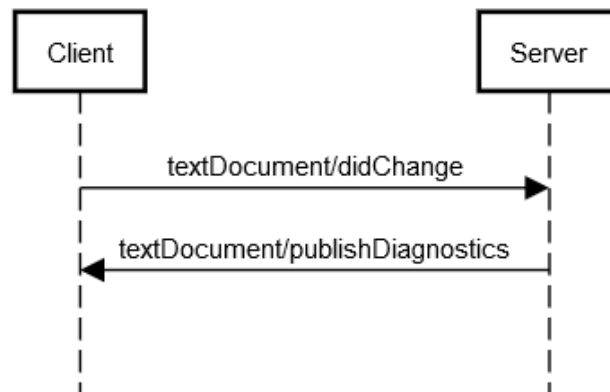


Figure 4: Example Communication

#### 4.1.3 Message Example

The following message is a `textDocument/publishDiagnostics` notification as it appears in the example above. It states that on line 4, from character 12 to 17, there is an assertion violation.

```
1 [12:45:29 DBG] Read response body
2 {
3   "jsonrpc":"2.0",
4   "method":"textDocument/publishDiagnostics",
5   "params":{
6     "uri":"file:///D:/[...]/fail1.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[\\dots]/fail1.dfy",
22         "message":"assertion violation"
23     }
24 ]
25 }
26 }
```

---

Listing 4: LSP Message Example

#### 4.1.4 LSP Standard and Individual Extensions

*Beschreibung von "Default LSP Sachen" und was wir erweitert haben. Olaf Zimmermann. Verweis auf Kapitel Custom LSP Messages*

## 4.2 OmniSharp

To work with the language server protocol, a proper LSP implementation was required. OmniSharp offers support for C#[7]. It could be simply installed as a NuGet package. OmniSharp also offers a language server client that can be used for testing.

### 4.2.1 Basic OmniSharp Usage

*rewritten* Mr. Martin Björkström published a comprehensible tutorial about Omnisharp's language server protocol implementation. 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         throw new NotImplementedException();
7     }
8 }
```

---

Listing 5: LSP Handler Implementation

Listing 5 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 auto completion, this is the file and the cursor position and some context information, how the auto completion event was triggered. The task of the language server is now to figure proper suggestions and return them in the form of a `CompletionList`.

Since OmniSharp is open source, we could find all available interfaces and thus all available handlers in their git repository [8]. 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 auto completions or diagnostics, 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 [9]. Mr. Björkström and his team were able to provide the solution for this issue.

The server can simply register custom handlers, too. 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 allow for maximal flexibility. The following code skeleton demonstrates how a custom request handler can be implemented:

```
1 public class CounterExampleParams : IRequest<CounterExampleResults> { [\dots]
2     }
3
4 [Serial, Method("counterExample")]
5 public interface ICounterExampleHandler : IJsonRpcRequestHandler<
6     CounterExampleParams, CounterExampleResults> { }
7
8 public class MyHandler : ICounterExampleHandler
9 {
10     public async Task<CounterExampleResults> Handle(CounterExampleParams
11         request, CancellationToken c)
12     {
13         CounterExampleResults r = await DoSomething(request);
14         return r;
15     }
16 }
```

Listing 6: LSP Handler Implementation

## 4.3 Visual Studio Code Plugin

todo. seite 16 SA.

## 4.4 Dafny Language Features

With regard to the symbol table, the Dafny language had to be studied more in detail. For example, shadowing describes the existence of multiple variables with the same name, but different visibility scopes. This is 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 Guide [10]. This chapter provides the reader with the most relevant concepts with regard to the symbol table.

#### 4.4.1 Modules

*C Sharp hat iwie nie einen Abstand... Latex fix? oben auch backslash dots nicht als drei dots im code snipped.*

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

---

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

---

Listing 7: Module Example

- The user writes 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, he 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. [11]

---

```
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 command `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 may consist of only one expression. For example, consider listing 9. Further, functions are not compiled and may only be used in specification context. That is, in contracts or assert statements to proof logical correctness. [11].
- The Function Method is just both at once. It also contains of a single expression with a single return, but is also compiled and thus also available in regular context. [11]

```
function method minFunctionMethod(a:int , b:int ):int
{
    if a<b then a else b
}
```

Listing 9: Function

Further concepts include:

- A predicate is just 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. An example can be found in listing 10. [10]

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

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. [10]

```
trait Base {
    var a: int
}

class Sub extends Base {
    constructor() {}
    //var a: int           // Error
}
```

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 means defining the same method with a different signature. This is, with different parameters. Dafny prohibits this language concept to be able to uniquely identify each method by its name [10]. 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 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 programmer needs to write a `this` in front of the variable name. One can even go further and redefine a local variable in a nested blockscope.

Consider the following code snippet. 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 first of all printed. This will print 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 necessary.

---

```
1 class A {
2     constructor () { a := 2; }
3     var a: int
4     method m()
5     modifies this
6     {
7         print a;           // 2
8         var a: string := "hello";
9         print a;           // hello
10        print this.a;       // 2
11        {
12            print a;        // hello
13            var a: bool := true;
14            print a;        // true
15            print this.a;    // 2
16        }
17    }
18 }
```

---

Listing 12: Complex Shadowing Example

Next, 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 print the boolean variable. `this.a` will still yield 2, even in 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.

->kapitel implementiaotn Regarding the implementation, the definition of a symbol could be found using the following method. Prerequisite is though, the scope. AllSymbols returns only those symbols that are defined so far.

---

```
1 private Symbol FindDeclaration(Symbol target, Symbol scope)
2 {
3     foreach (Symbol 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 above would basically already resolve the *GoTo Definition* problem.

*Corbat: @if und dann if als variable ist kein gültiger indentifiyer in dafny. Noch weitere versprochene doku in der agenda?*

## 4.5 Symbol Table

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 }
```

---

Listing 14: AST Demo Snippet

The tree segment for this snippet would contain of the while-Statement as the root node. It then has two branches, one for the condition, and one for the body. The body itself consists of a list of expressions. In the above example, there is only one expression, namely an assignemnt. The assignment has again a left and a right side. The right side is a binary expression, with the +-operator. Left of the plus is a name segment, and on the right hand side a literal expression.

Often, the AST doesn't 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. This way, it could be stored that the name segment 'i' is of type int. The two concepts are strongly coupled.

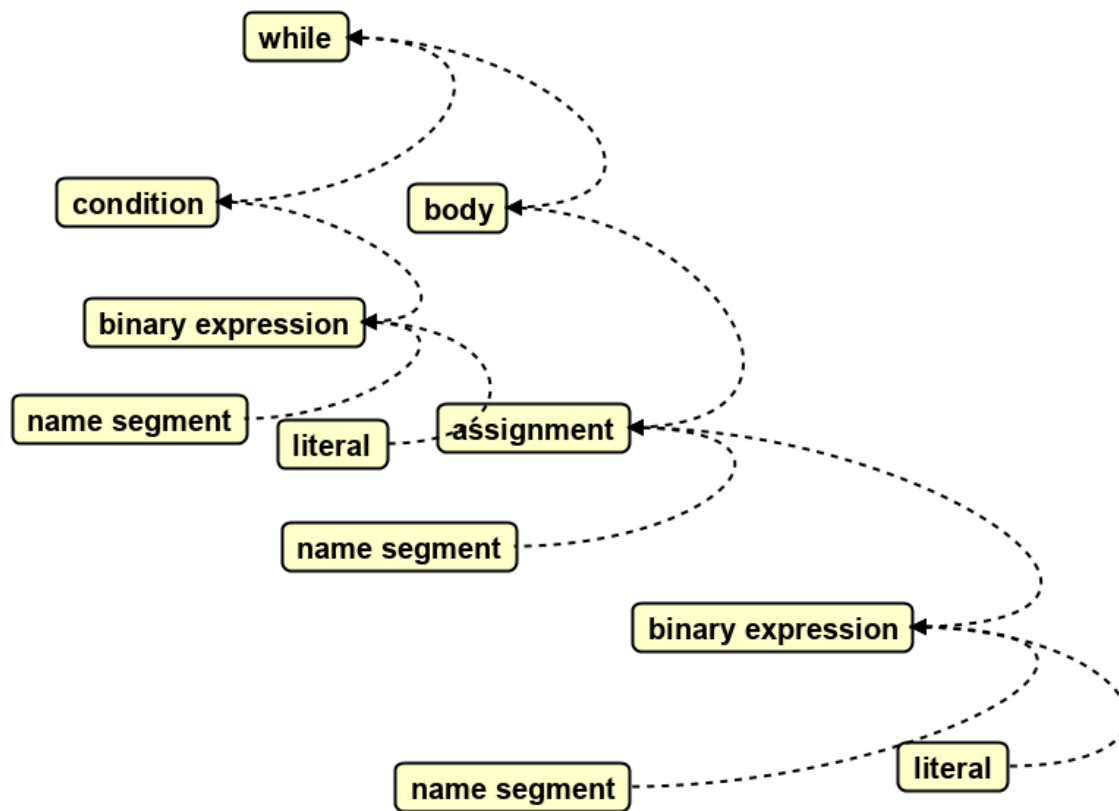


Figure 5: Communication Benefit of LSP

#### 4.5.1 Requirements for the symbol table

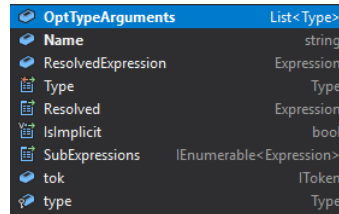
To be able to implement our feature set, the following requirements must be fulfilled by the symbol table.

- Cursor Position
  - Which name segment is at the cursor's position?
- Goto Definition
  - Where is the a symbol declared?
- Code Lense
  - How often, and where, is a declaration used?
- Rename
  - What are all occurrences of a symbol?
- Autocompletion
  - Which declarations are available in a scope?



### 4.5.2 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. The following screenshot 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.

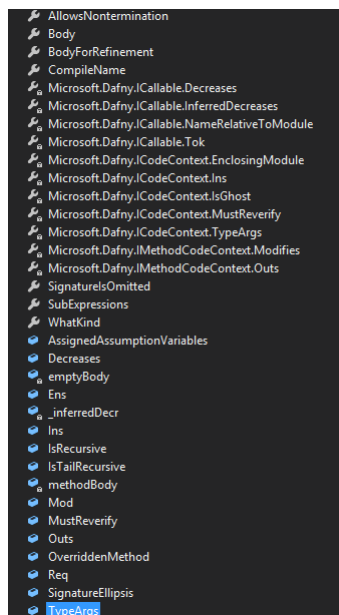


Property/Field	Type
OptTypeArguments	List<Type>
Name	string
ResolvedExpression	Expression
Type	Type
Resolved	Expression
IsImplicit	bool
SubExpressions	IEnumerable<Expression>
tok	IToken
type	Type

Figure 6: Properties and Fields of a NameSegment Expression

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

A better example may be on a higher level. Let us have a look at the class `method`. It contains properties and methods for its body, but not exactly which name segments are declared inside that body. While it has a property `EnclosingModule`, it is not stated in what class that method is defined. The property `CompileName` contains information about the enclosing class, but only as a string and is thus of limited use. Also, there is no way to know where that method is used, which was a prerequisite for the code lens feature.



AllowsNonTermination
Body
BodyForRefinement
CompileName
Microsoft.Dafny.ICallable.Decreases
Microsoft.Dafny.ICallable.InferredDecreases
Microsoft.Dafny.ICallable.NameRelativeToModule
Microsoft.Dafny.ICallable.Tok
Microsoft.Dafny.CodeContext.EnclosingModule
Microsoft.Dafny.CodeContext.Ins
Microsoft.Dafny.CodeContext.IsGhost
Microsoft.Dafny.CodeContext.MustReverify
Microsoft.Dafny.CodeContext.TypeArgs
Microsoft.Dafny.IMethodCodeContext.Modifies
Microsoft.Dafny.IMethodCodeContext.Outs
SignaturesOmitted
SubExpressions
WhatKind
AssignedAssumptionVariables
Decreases
emptyBody
Ens
_inferredDecr
Ins
IsRecursive
IsTailRecursive
methodBody
Mod
MustReverify
Outs
OverriddenMethod
Req
SignatureEllipsis
TypeArgs

Figure 7: Properties and Fields of a Method

Thus, it was decided to implement an own symbol table.

## 4.6 Visitor

To be able to generate an own symbol table, it is common to use the visitor programming language pattern by !!REFERENZ von den heinis gang of four balblada!!. 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 a tree based data structure. Every node in the tree is supposed to offer an `Accept(Visitor v)` method. 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 (Node child in this.Children) {
4         child.Accept(v);
5     }
6 }
```

---

Listing 15: Example for Accept

Note that the navigational aspect - the foreach loop - is inside the accept method, but nothing is said about the visit operation. The visitor can do whatever it wants with the node. The visitor has overload its `Visit` method for each possible node that it is visiting. Within a tree, these are usually nodes and leafs. For a symbol table, these are any kinds of expressions and statements. To complete the example, the visitor could simply print the node. Its implementation could look like shown below:

---

```
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 16: Example for Visitor

## 4.7 Dafny Expression and Statement Types

*Ist dieses Kapitel an einer guten Position? Besser nach Dafny Language Features oder so? Ahh. Ne schon richtig hier. "Platsch" einstieg ersetzen durch "was es so gibt, und warum wichtig (beachten für visitor) etc. Analyse von was es für types gibt und warum. einstieg."*

Dafny is a very versatile language. While it offers common object oriented language features, it also contains formal language features, comparable to more common languages like Haskell. Thus, it contains numerous

AST-nodes. The most important ones shall be discussed within this section.

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

- Expression
- Statement
- Declaration

Aside these, some AST-nodes are separate, such as `AssignmentRHS`, which is the right side of an assignment. `LocalVariable` is another example for an isolated class, that does not extend any base class. Why this decision was made by Dafny could not be evaluated. Both items are actually expressions and could technically be subclasses of `Expression`.

#### 4.7.1 Expressions

In this chapter, the most important expression types are explained:

- `NameSegment`: Any name of a variable or method.
- `BinaryExpression`: An expression with two operands, for example 'plus', or 'less than'.
- `NegationExpression`: Just the negation of a variable or literal, for example `-b`.
- `UnaryOpExpression`: A unary expression, mostly connected to the "not"-Operator, for example `!someBoolean`.
- `ITEExpr`: If-then-else expression, such as `if a < 0 then -a else a`
- `ParensExpression`: Any expression surrounded by brackets.
- `AutoGhostIdentifierExpr`: If a variable declaration also contains an assignment as well, the left hand side of the declaration is a ghost-identifier.
- `LiteralExpression`: Literals like numbers or strings.
- `ApplySuffix`: The brackets after a method. This expression just refers to the brackets, the actual arguments are stored within the method expression.
- `MaybeFreeExpr`: Occurs at ensure-clauses and just contains a subexpression.
- `FrameExpr`: Occurs at the modifies-clause and just contains a list of subexpressions.

The reader notes himself, that many expressions contain of other subexpressions.

#### 4.7.2 Statements

- `BlockStmt`: Anything surrounded by curly brackets.
- `IfStmt`: A classic if-statement. It contains an expression for the condition, a blockstatement for the then-block, and another, optional if-statement for the else-block.
- `WhileStmt`: A while-loop. It also contains a blockstatement for its body and an expression for the condition. Aside these, it also contains expressions for the loop invariants and decrease-clauses.
- `Method`: The method contains a block statement for its body. The arguments and return values are stored as `Formals`.
- `Function`: Analog to method.

### 4.7.3 Declarations

The following declarations were analyzed:

- ModuleDecl: A module declaration
- Class: A class declaration
- Field: A variable member of a class
- Method: A method member of a class
- Function: A function member of a class

## 4.8 Dafny AST Implementation

During analysis of the Dafny AST, it was noticed that the file `DafnyAst.cs` is huge. 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 crashed occasionally on performing autocompletions.

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

Splitting the file into individual class files and dividing it into a separate packages would provide a 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:

- Time-consuming
- Inconsistency: Any other Dafny files would still be rather large. Refactoring should then be extended
- The maintainers of Dafny may not want a refactored style 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 concept of the bachelor thesis, such as the implementation of the symbol table. However, refactoring the code of Dafny itself is one of the possible outlooks of this project.

## 4.9 Continuous Integration (CI)

Continuous integration is a very important part for code quality and collaboration. Unfortunately, setting up the CI process in the preceding project[3] took almost until the end of the project stage.

According to our 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.

This chapter describes the initial situation, the desired objectives and the targets achieved.

#### 4.9.1 Initial Situation

On the client side, code was analyzed with SonarQube during the CI process. If the code contained TypeScript errors, the build failed [3]. The end-to-end tests, provided by the preceding bachelor thesis[1] could not be integrated until the end of the project, due to their heavy dependencies.

On the server side, the project was automatically built by the CI server. Our own unit tests, as well as tests provided by Dafny could as well be automatically executed. Integration testing and code analysis by SonarQube remained pending [3].

#### 4.9.2 Targeted Solution - Achieved Solution

According to our research, a major problem was that the scanner for SonarQube can only analyze one language at a time. [12]. This means, that the TypeScript code in the client and the C# code in the server cannot be analyzed simultaneously. Furthermore, in the preexisting Dafny project, there are also single Java files. 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 [13]. The available statistics are very helpful for code reviews.

The only downside is that the code coverage 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 [14]. The CI server bases on Linux, though. During our research we came across *monocov* [15]. This tool would run under Linux and analyze .NET Framework projects. Unfortunately this project was archived and has not been supported for almost 10 years [15].

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 locally installed IDEs anyway.

The end-to-end tests base on a lot of dependencies, such as a headless instance of Visual Studio Code. In consultation with our supervisor, we have removed these tests from the client project and replaced them with own specially written integration tests on the server side.

#### 4.9.3 Docker

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

For an easier testability of the CI, we also installed Docker locally. This allowed us to resolve CI issues locally and platform-independently (through the Docker Client) in case of problems. See the developer documentation for more details [13].

Docker also realizes the principle "Cattle, not pets" with docker. *hier noch irgend eine cite adden wo das prinzip definiert wird... hab nur grad stackoverflow gefunden, was zitierfähiges wäre nett. sonst wirkt es fast etwas kindisch/verpsielt für eine BA.* 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 the like 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 within the CI server. This is, since Dafny relies on specific deployments of these products. See the developer documentation for more details [13].

*wtf ist go, und node und sonar ist doch egal welche version oder?*

#### 4.9.4 2do - Kapitelaufteilung komisch

Ich hab hier jetzt in der Analyse auch schon die Lösung vorabgegriffen. Sollen wir das splitten? Bricht das nicht den Lesefluss? Evt besprechen. ja es is iwie komisch. auch unten habenw ir ja noch testing und blabla... kann man evtl so ein CI Kapitel vlt sogar machen? grml.

todo

#### 4.10 notizen von marcel

→ Gedanken zum Updaten sind wichtig. Evt ned alles implementieren aber dokumentieren.... Effizienz. Ned alles neu Builden wenn in einem File nur ein Zeichen geändert wird auf einer Linie etc. → Ausblick. Und Testing in einem grossen Dafny Project wär evt auch noch ganz nice... ein paar Performance-Tests und so? Und die dann mit dem Plugin aus der Studienarbeit vergleichen? Und dem alten-alten Plugin? Käme bestimmt jut an.

#### 4.11 notizen von tom

gleich nachm schreiben, was mir onch so durch den kopf geht:

- Visitor Pattern braucht auch olaf und guido nicht jeden tag. Kommen die draus? Wäre nicht ein schema-bildchen noch gut, aka 'kuck hier links, der accept sagt wo ich durch muss, udn kuck hier rechts, der visitor sagt was getan wird. oder irgend sowas, ka.
- expression list und co total random... halt brutal unvollständig etc pp. erklären, warum das unvollständig ist.
- SymbolTable, AST, connected, blabla... ist das gut genug? Corbat hat sehr viel Ahnung davon und mein Halbwissen ist da ein bisschen gefährlich, das gibt dann ja total schelchten eindruck.
- Lanague Feautres: am anfang noch spezifisch schreiben, dass wir jetzt nur auf den shit eingehen, der für symboltable relevant ist, aka wo könnten doppelte namen vorkommen.
- Ist der flow eig klar? wir müssen ne symbolt able machen, ⇒ was hat n dafny da überhaupt so, ⇒ wie machen wir es, ⇒ visitor is gut für sowas ⇒ muss dies und das visitien.

## 5 Design

This chapter contains discussions of fundamental design decisions. It is primarily divided into client and server architecture.

### 5.1 Technologies

The choice of technologies is determined by external circumstances. The client will be written in TypeScript, since VSCode plugins are required to be written in TypeScript [16]. The client will be written in C# using .NET Framework 4.6.1, since Dafny is already coded in this setup and direct integration is the key goal of the project.

### 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 on the server side. This allows to implement support for another IDE with as little effort as possible.

In the following it is discussed how this lightweight design of the client has been achieved. Furthermore, the splitting into components is also explained.

#### 5.2.1 Initial Situation

The client, taken over from the preexisting bachelor thesis by M. Schaden and R. Krucher [1], was already refactored in the preceding term project. Aside establishing a connection to the new language server, a lot of dead code was removed and logic was moved to the server. Figure 8 gives an impression of the architecture at the beginning of this bachelor thesis..

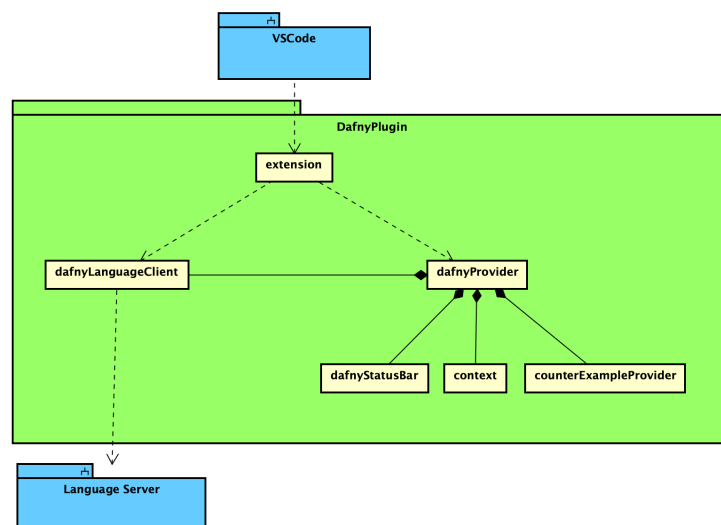


Figure 8: Client Architecture - Term Project

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. That made it difficult to maintain the code. Furthermore, it was difficult to identify all dead code due to the non-transparent dependencies.

Because of these problems it was decided to redesign the client itself 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 grouped into packages as you can see in figure 9. These packages are discussed in the following sections.

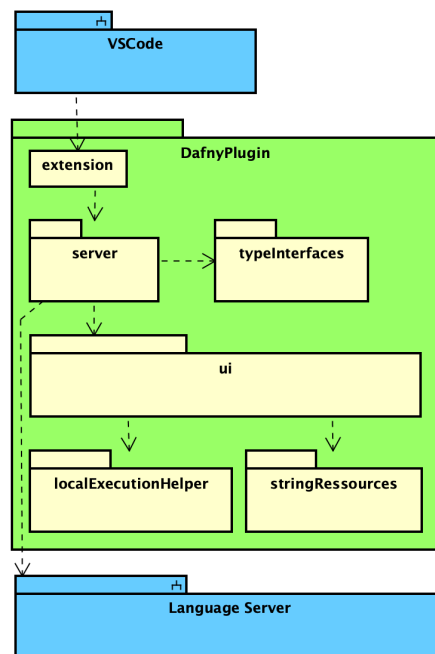


Figure 9: Client Architecture - Packages

Additionally, all logic was detached from the extension class (the main component). This resulted in the root directory containing only a lightweight program entry point and 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.

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

**Server** – The server package contains the initializing of the language server and the establishing of the connection between client (plugin for VSCode) and Dafny language server. In addition, all server requests,



which extend the LSP by own functions, are sent to the server via this package.

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

**UI** – The UI is responsible for all visual displays, 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 chapters the individual components and their contents are described in more detail.

### 5.2.3 Components

Figure 10 shows a more detailed view of the client, including the components within the packages. The contents of type interfaces and string resources have been omitted for clarity.

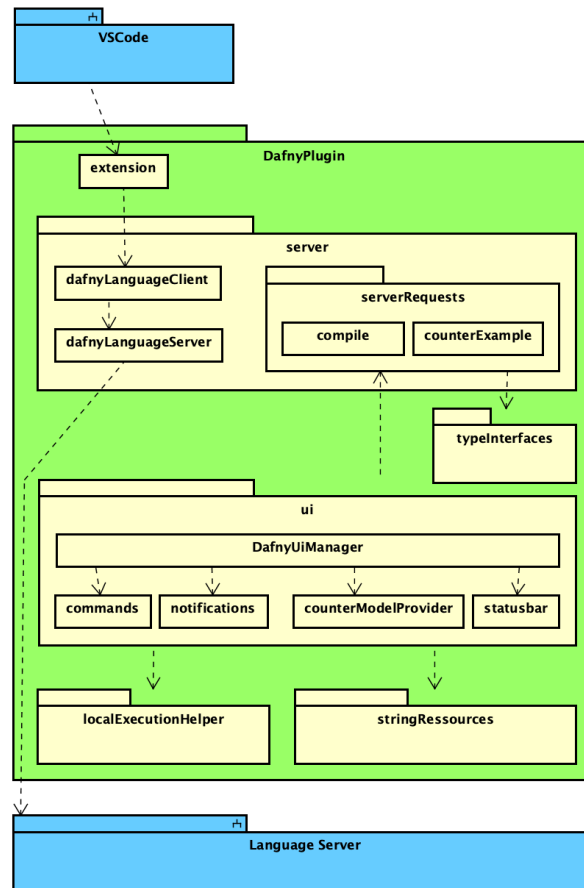


Figure 10: Client Architecture - Components

It can be seen that only *compile* and *counterexample* exist as dedicated server access classes. All other features, such as *code lens* or *auto completion* are natively supported by the LSP. This means that no additional client logic is necessary to support these features. If VSCode receives a *auto completion 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. For future plugin developments for other IDEs, the effort is minimal.

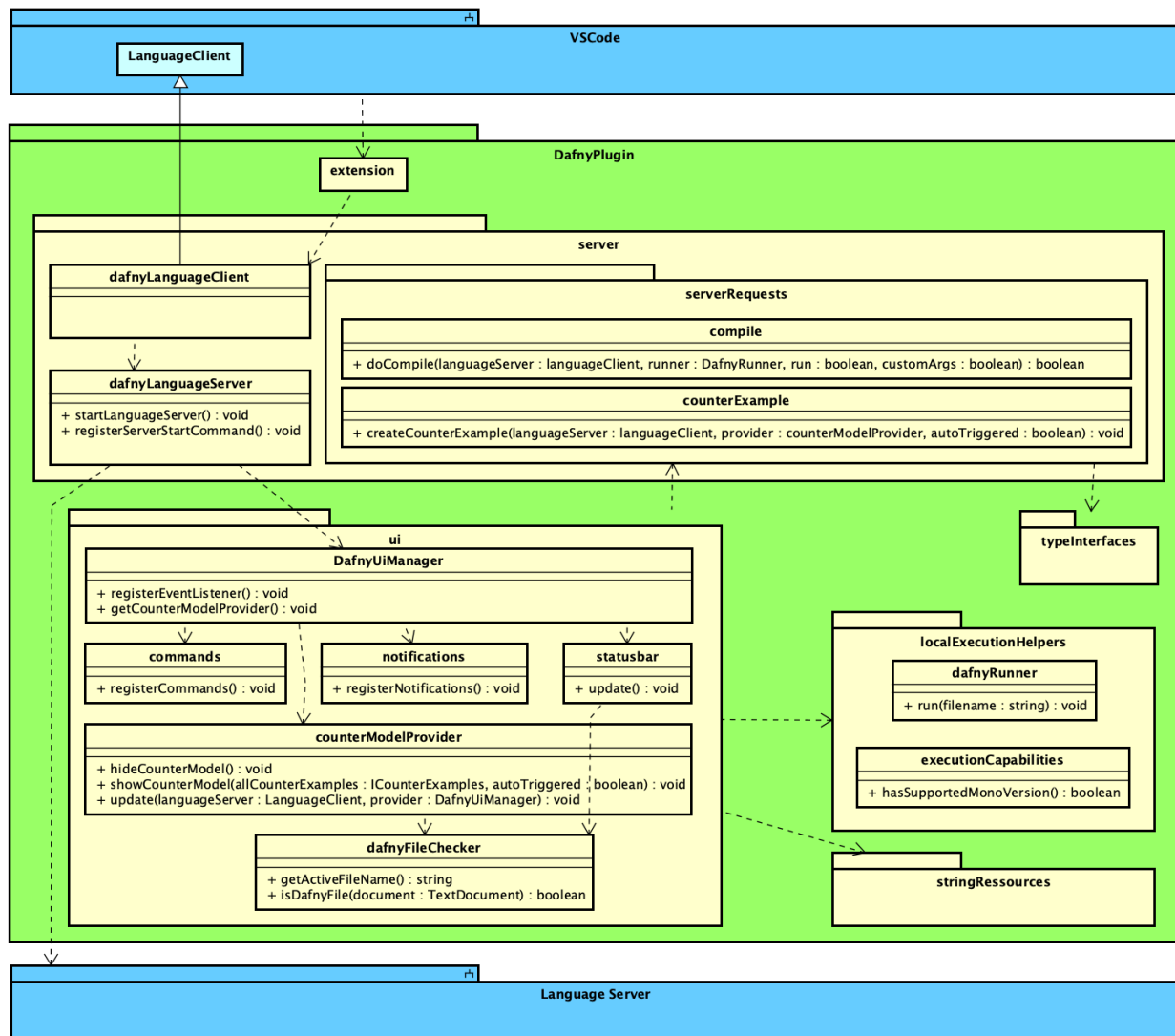


Figure 11: Client Architecture - Components and Public Methods

Figure 11 shows the public methods of each of the components. Only these are accessible, all 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 a certain upward dependencies, which is not perfect. The UI package accesses the server requests to be exact. Nevertheless, we have decided on this grouping, so that the server access functionality is encapsulated.

### 5.2.4 Logic

As said in the introduction, 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 subchapter describes where and why the client still contains logic.

**Server Connection** – Starting the language server and send API requests. In addition, the client has a simple logic that certain server requests (such as updating the counter example) 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.

**Statusbar** – The information content for the statusbar 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.

**Counter Example** – The counter example has a similar buffer as the statusbar. For each Dafny file in the workspace, a buffer stores whether the user wants to see the counter example. 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 chapters, `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 typecastings.

There are individual built-in datatypes like `number`, `boolean` and `string` [17]. For custom types, such as `CounterExampleResults`, we have defined separate interfaces.

*«<hier vielleicht nich beispiel screenshots oder codes rein? >> 2do -> würde ich dann deifnitiv bei implementaiton machen, nicht hier. oder? Nennen welche; Sind alle Language Server Response (Begründung).*

*OmniSharp Omnisharp, VSCode VS Code. überall einheitlich.*

*Mir ist aufgefallen, dass die Information zum teil doppelt kommt: erst beim intro 'wir machen ejtzt interfaces statt any zu verwenden. dann 'jetzt kommt allesnochmal im detail', dann kommt 'wir haben statt any interfaces verwendet' nicht sicher ob das aber schlimm ist oder nur mir auffällt, ein intro zu haben ist ja legitim.*

### 5.3 Server

*vlt noch ne übersicht iwo, wie figure 9 SA - hab grad keine übersicht* The server's responsibility is to offer a language server that is able to reply to a client's request. Omnisharp offers a simple static method to launch such a server. As launch options, all supported handler classes can be passed. Listing 24 shows a simplified example with two handlers for text document changes and renaming.

```
1 LanguageServer.From(options => options
2     .WithHandler<TextDocumentSyncTaskHandler>()
3     .WithHandler<RenameTaskHandler>()
4     ...
5 );
```

Listing 17: Language Server Initialization

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 results. This is the basic workflow of a language server implementation. The actual challenge is formed by the implementation of the `Handle` methods.

Figure 12 shows the basic, fundamental layout of the server implementation. Be aware that this illustration is highly simplified for a better understanding.

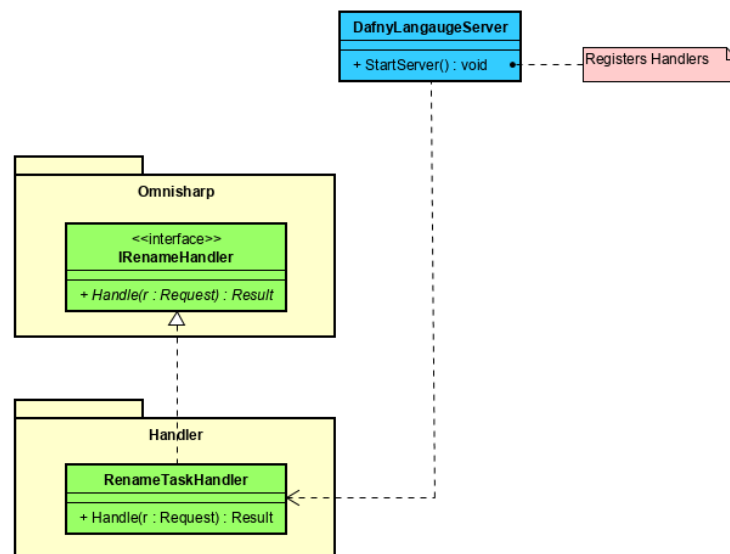


Figure 12: Basic Server Concept

The system under development will not evaluate Dafny code itself. Instead, requests are just forwarded to Dafny. Such requests could for example be:

- Verify the Dafny code
- Provide a counter example
- Compile the code

Other functionality is not directly implemented by Dafny, for example the option to jump to a definition of a symbol. Thus, the already discussed symbol table will be required this functionality. The following subchapter provide a step-by-step overview of the design decisions taken for this project.

### 5.3.1 Dafny Translation Unit

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 class `PhysicalFile` just represents a file on the user's filesystem and will be discussed in the next chapter. The only public method `Verify` will start the Dafny verification process based on this file. As a result, the process will yield a list of errors, warnings and informations, but also a precompiled `dafnyProgram` and a precompiled `boogieProgram` for later reuse. All of this information is stored in the `TranslationResults` wrapper class. Since `TranslationResults` and `PhysicalFile` are used on multiple parts of the code, these classes were place within a `Commons` package.

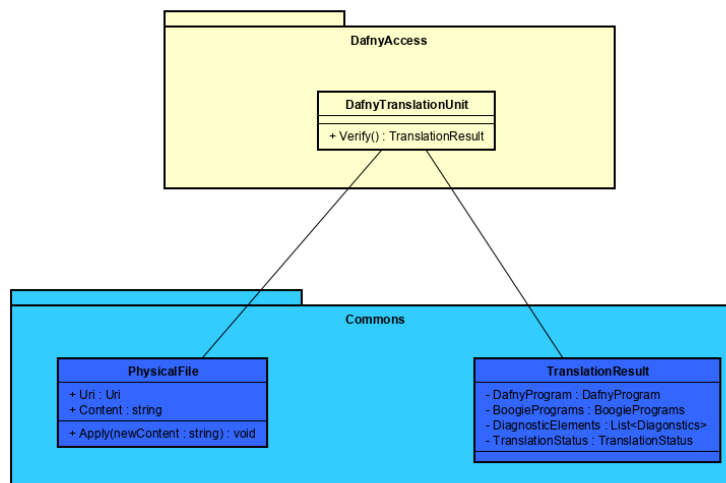


Figure 13: Dafny Translation Unit

Excluded from the figure are some converter methods, that convert `Diagnstoics`, such as warnings and errors, to an uniform format. Dafny and Boogie report their errors in different ways and different formats, which made their collecting actually relatively complex. To counterfeight that, simple converter extensions were written to unify any occuring errors.

*eben hier noch einfügen wie das jetzt mit der ST ist*

### 5.3.2 Workspace Manager

A Dafny project consists out of `.dfy` files. Thus, it was evident to create a class `PhysicalFile`. It just has two properties, an URI and the file content. Since the user is 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:

- Is the file valid?
- Does ist contain errors?
- What does the compiled Dafny und Boogie program look like?

- What symbols occur in the file? *ST diskutieren*

*Bei Aufzählungen gross/klein starten. einheitlich internen noch auflösen ob symbol table pro workspace oder pro file*

Thus, a class `FileRepository` was created. It contains a `PhysicalFile`, but also all of the requested information in the list above. For that, a wrapper `TranslationResults` was created, containing information about errors and compiled items. Symboltable?

To request information about a file, or to update a file, a `WorkspaceManager` was created. It contains a dictionary, linking a file identifier (an URI) to a `FileRepository`. It offers a method to update a file, but also to get information about a file.

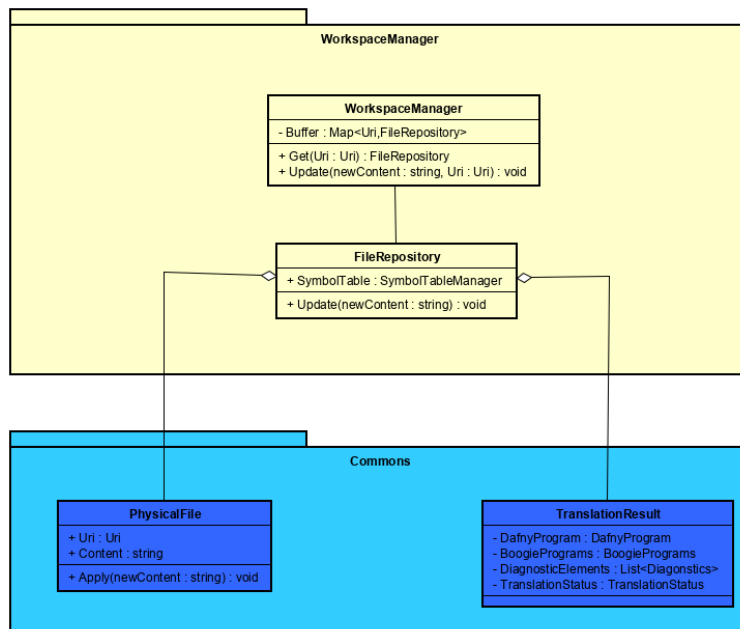


Figure 14: Workspace Component

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` and a *new symbol table!!* by invoking the `Dafny Access` package. Any changes are then stored in its `Buffer` property.

Inside the code, the reader will find many more methods than shown in figure 14. The `PhysicalFile` will contain a variety of 'helper'-methods, such as getting the length of a specific line and such. These do not play a central role and are not further discussed. Any `Update` method is further generic and available for incremental text changes sent by the client, but also for full text mode.

### 5.3.3 SymbolTableManager

Creating, managing and navigating through the symbol table is a highly complex process. Thus, it was decided to create an own package for it. The symbol table is built based on a `dafnyProgram`. This has the advantage, that any included files (but only as much as necessary) are directly included within the `dafnyProgram`. Furthermore, the `dafnyProgram` is available from the `WorkspaceManager` for every file. This component contains of three classes:

- `SymbolInformation`, a class containing just data about a symbol.

- Navigator, a class to navigate through the symbol table.
- Manager, the class that generates the symbol table.

The `SymbolInformation` class 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? This is just how they were discussed in chapter ??.

The `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. One task could be to navigate upwards, starting from a symbol dep inside the tree. This way, all available declarations could be retrieved. Another task is to navigate from top to bottom, for example to locate a specific symbol. As a global access point, there is a root symbol. Thus, every top-down search task can be started from that root symbol.

The `Manager` is responsible to build the table. It will already make use of the navigator, namely to locate declarations of symbols.

All three components are highly coupled. The dependencies are not drawn in the following figure.

*noch ergänzen dass navigator von nem symbol asu was macht aber der manager immer über die ganze table geht falls wir das nicht mehr ändern*

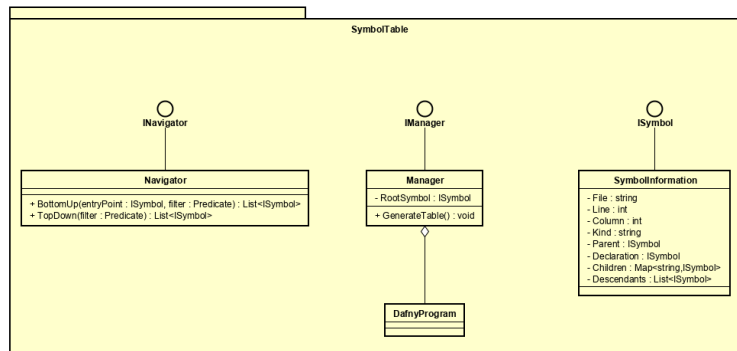


Figure 15: SymbolTable Component

### 5.3.4 Core

The core package contains the actual logic. There is one class for each feature, for example `RenameProvider` or `DiagnosoticsProvider`. The provider is called by the handler. This way, any Omnisharp duties, such as registration of capabilities is separated by the actual core logic. The necessary parameters are forwarded from the handler to the provider. Most often, this is the `FileRepository`, which the handler requested from the `WorkspaceManager`. This way, the provider has access to all precompiled results, including the symbol table. Each provider implements a specific interface, so that fakes can easily be created for testing.

### 5.3.5 Ressources

String ressources, such as error messages, were extracted to a specific ressource package. This way, they are easy to maintain and adjustable at a central place.

### 5.3.6 Tools

Some tasks require specific components. For example, there is one class reading the launch arguments to set up some options. Another class reads configurable reserved Dafny keywords. All these auxiliary classes were collected within the `Tools` package.



### 5.3.7 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.

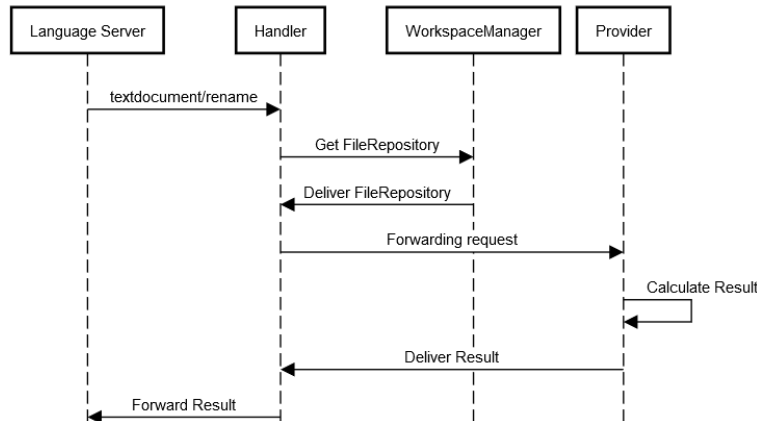


Figure 16: Overview

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. Again, the handler would forward the repository to a provider of his choice. In the case of an update, the VerificationProvider, which sends Diagnostics to the Client would to be invoked.

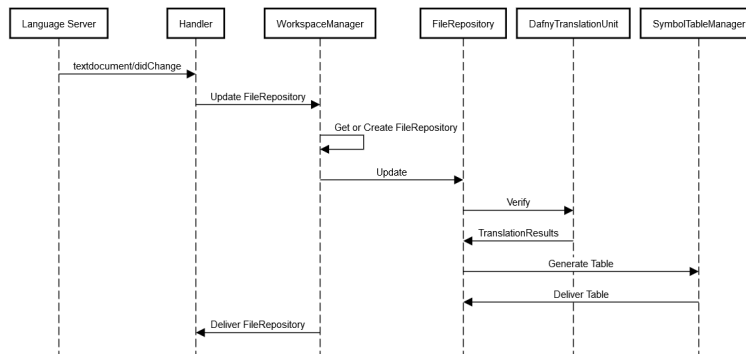


Figure 17: Updating a File

Plan Kapitelablauf:

1. Aside the basic layout, we also needed to store some information for files. Also -> Workspace Manager
2. Dann Workspace Manager ansich erklären, isoliert. schaut dann voll aufgeräumt aus.
3. Evtl schon sagen, dass Workspace als singleton service injected wird und somit immer bei alles handlern available ist. (was man übrigens mit msg sender auch tun könnte, das wär noch fancy aber wieder aufwand.)
4. Dann brauchen wir Dafny Access -> Translation Ding Zeugs da sagen. (evtl sogar zuerst?)

5. Die eigentliche Core Logic wurde dann in die Handler Services ausgelagert (NAME BESSER!!). also handler -> handlerService, welcher dann die eigentliche logic durchbuttert. service somit isoliert testbar mimi interface blabla. itnerace müsste man noch machen hust hsut. ok. dfas lohnt sich wohl aber dneke ich. (notenmässig)
6. resx als krimskrams
7. config und msg sender und wordsprovider als 'library' layer. den besser auch noch renamen.

## 5.4 Integration Tests

Unlike in the preceding semester thesis, integration tests could be implemented using Omnisharp's language server client [18]. Each test starts a language server and a language client, then they connect to each other. Now, the client can send supported requests, for example "get me the counter examples for file ../test.dfy". The result can be directly parsed into our `CounterExampleResults` datastructure and be compared to the expectation. Thus, tests can be written easily and are very meaningful and highly relevant.

### 5.4.1 Dafny Test Files

Integration Tests usually run directly on `dfy` sourcefiles. Those testfiles 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 testfile 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 18: Test File Reference

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

### 5.4.2 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 had to be compared to 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 counter example will be converted into a unique string, containing all information about the counter example. All counter examples together are assembled within a list of strings. This way, they can be easily compared against each other.

Since not only counter examples, but also other data structures such as `Diagnostic` were converted into lists of strings, the converters were held generic as far as possible. The following listing 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 }

---

Listing 19: Generic Method to Convert an IEnumerable

Calling the above method for counter examples are made as follows. A list of counter examples is handed as the argument, and a `Func<CounterExample, string> ToCustomString` is handed as the converter. The converter is also shown in the following code segment. Not that it is defined as an extension method.

---

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

---

Listing 20: Converting CounterExamples to strings

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

---

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

---

Listing 21: Expectation

The results can be converted to a string list using the defined `results.ToStringList()` method. By taking advantage of the method `CollectionAssert.AreEqual(expectation, actual)` from NUnit's test framework, the two lists can be easily compared against each other [19].

### 5.4.3 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, so that every test will at first setup the client-server infrastructure, and tear it down after the test has been completed.

It was considered if the `IntegrationTestBase` class should directly contain a class member `T TestResults` to store the test results, as well as a method `SendRequest` and `VerifyResults`. While storing the test results could have been realized, this was not possible for the methods `SendRequest` and `VerifyResults`. The problem is, that these methods have different signatures from test case to test case. A compilation request has different parameters (such as compilation arguments), than a goto-definition request (which has a position as a parameter).

Instead, it was decided to create a second base class for each test case. 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)`. One can now easily see the dedicated parameter list.

The test class itself inherits from its case-specific base class. The tests themselves 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 our 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 22: Sample Test for Missing Semicolon

As you can see, the test contains only two lines of code. The first handles the test file, the second one defines our expectations. By the way, the boolean values represent if there were errors and if an executable was generated.

The same applies for tests about counter examples, goto definitions and other use cases. Thus, the integration test architecture could be created in a way so that the creation of tests is extremely simple and user friendly. The code can be kept very clean and contains no duplicated code. 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 such.

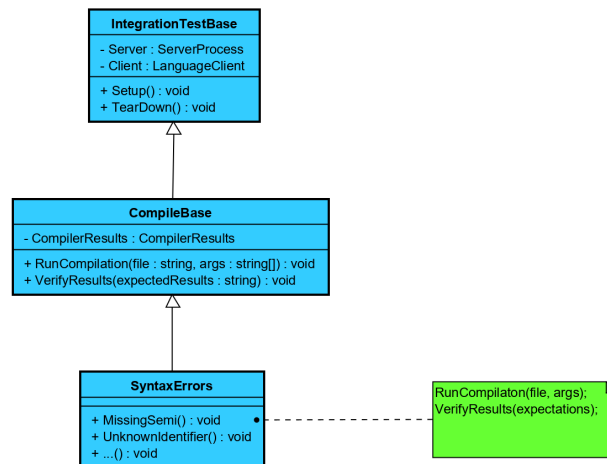


Figure 18: Test Architecture on the Basis of Compilation

#### 5.4.4 Performance Measurement

To measure performance, a little algorithm was written that creates a pseudorandom Dafny file. Within a classmethod, 10'000 LOC are generated, either containing

- a variable definition: `var v142 := v16;`
- a variable access: `print v142;`
- a blockscope: `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 9'000 variable name segments, that have to be resolved. This challenges the symbol table quite a bit. Since the `textDocument/didOpen` notification is not awaitable, a random LSP request was sent after the opening to wait until all actions are finished. Since we implemented compile in such a way, that it uses the precompiled `dafnyProgram`, we have chosen to run a `compile` command. The test has shown, that the process completes within 20 seconds on a 3.4GHz machine, which seems quite resonable for such a large file.

For reproducibility, `nUnits` randomizer was used, which will create the same 'random' testfile on every run. Technically, the file generation could be excluded from the test engine, but it was left inside, in case someone wants to create additional or other tests.

## 6 Implementation

### 6.1 Client

Das Refactoring bereits hier mitmachen? Dann gleich den "Ist zustand" mit dem Refactroing vom Design hier einflieessen lassen. Ist für den Leser meine ich weniger verwirrend.

#### 6.1.1 Client (Code Review)

After a joint code review together with our advisors, individual optimisation potential was identified. This subchapter describes the associated improvements to the architecture.

Although interfaces were used for the individualized types, the individual 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 19 shows an overview of the interfaces. In addition, the dependencies among each other are shown. For simplicity, the contents of `stringRessources` and `typeInterfaces` have been omitted.

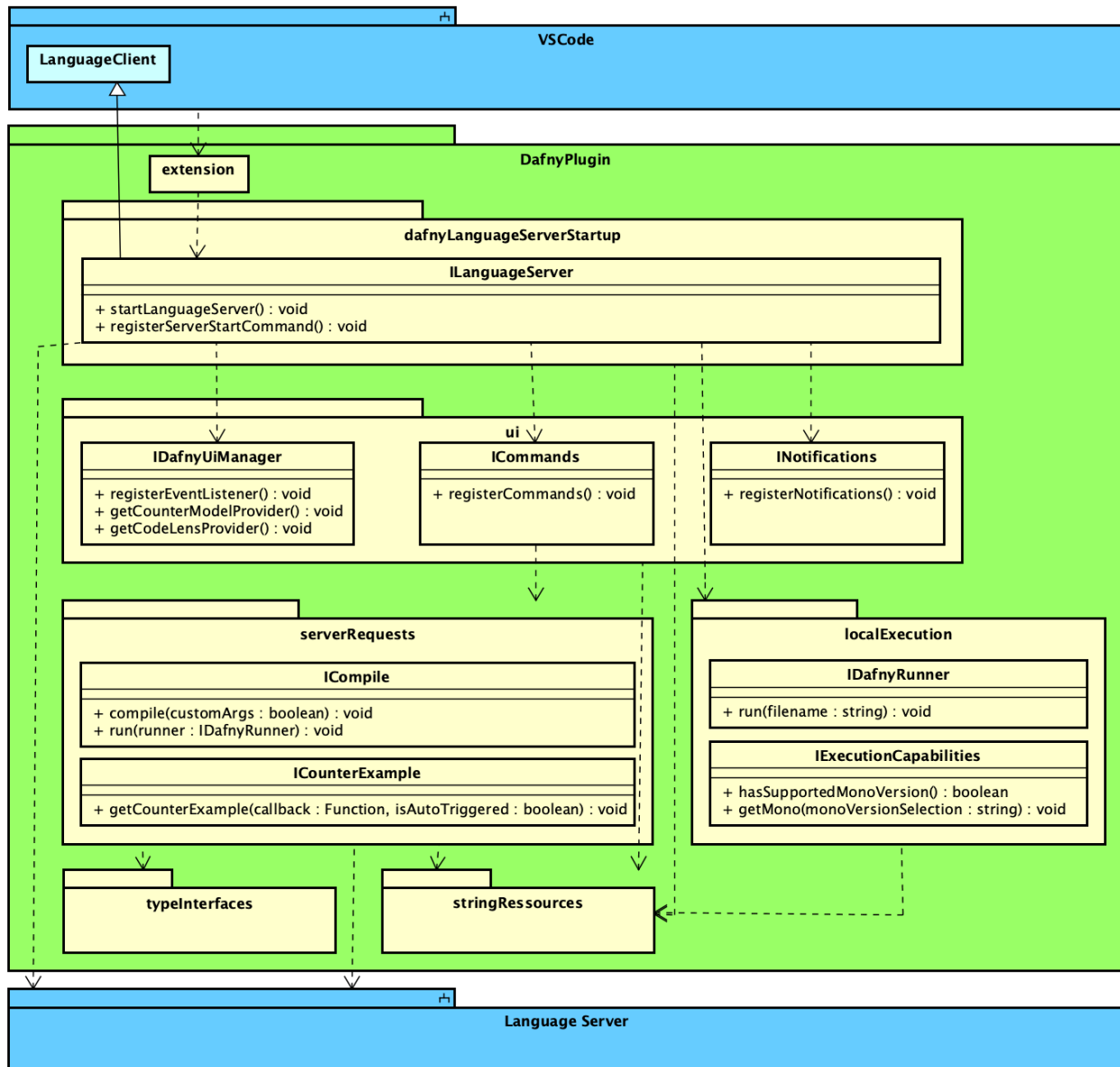


Figure 19: Second Major Client Refactoring

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 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.

## 6.2 Server

### 6.2.1 Server Launch

As done in common practice, the `Main` function is kept very short. All it does is launching the language server, which is already handled by another class.

---

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

---

Listing 23: Main Function

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

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

Secondly, 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 startup. Lastly, the console output stream is redirected to keep the language server stream clean.

---

```
1 public async Task StartServer()
2 {
3     log.Debug(Resources.LoggingMessages.server_starting);
4     server = await LanguageServer.From(options => \dots );
5     ExecutePostLaunchTasks();
6     await RedirectStreamUntilServerExits();
7     log.Debug(Resources.LoggingMessages.server_closed);
8 }
```

---

Listing 24: Starting the Language Server

### 6.2.2 Tools

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

*kann man hier so subsubsubchapter machen? geht das iwie? sons tienfach so lassen oder?* This class is called 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 to avoid any kind of null pointer exceptions. Afterwards, the file `Config/LanguageServerConfig.json` is parsed with Newtonsoft's `Json.NET` library [20]. Any available values will be written to the static configuration class. Unknown or illegal values will not be set and errors are added to a 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 [21]. The config initializer is implemented exception safe. This means that it will run to completion and at worst just provide default values. Errors can later be extracted from the `InitializationErrors` property.

### LoggerCreator

This class simply sets up a Serilog [22] logger. For this purpose, it will already read from the static config class `Commons/LanguageServerconfig.cs` the information what the minimum loglevel is and where to locate the logfile.

### MessageSenderService

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

### ReservedWordsProvider

This is a class reporting simply offering a method returning 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 [10].

While this component is specifically used solely for the `rename`-Feature, it was extracted to be also available at other spots if required for future features.

## 6.2.3 Handler

Handlers are passed to the language server and are called whenever the language server receives a corresponding request. Services, such logging or workspace management, can be injected and are thus available for the handler. Omnisharp directly defines interfaces, which have to be used for the handler implementation. For example, the interface `IDefinitionHandler` requires the class to implement a `Handle` method. Everytime the server receives a `textDocument/Definitions` request, this `Handle` method will be called. The parameter and return types are specific per request. `Goto Definition` would pass a `textdocument` 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 chapter 4.2.2 by defining an own interface.

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

- `GetRegistrationOptions`: Is called when the handler is registered. It allows to set a document selector. This is, that the handler is only active for '.dfy' files.
- `SetCapability`: Allows to set handler-specific capabilities, such as if the rename-handler will also support a 'prepare rename' feature.

The code for these methods is always identical and was thus extracted to a generic base class. The generic type parameter refers to the kind of capability, necessary for the `SetCapability` method. This could be, for example, `RenameCapability` which has the mentioned bool property about the preparation support.

To keep all this boilerplate code separated from the core logic, the actual `Handle` method will in most cases just create a provider instance, where all core logic is placed, and forward its result. For example, the full code of handling a compilation looks as shown below.

---

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

---

Listing 25: Handling Compilation

This is a typical structure of handler. It requests the injected workspace manager for the file and passes it to the core logic provider. Finally, the result is returned as a result. 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 way, the message sending service does not have to be passed downwards to the core logic component.

#### 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 26: Workspace Property

It offers methods to retrieve files and to update them. Since updates can be down in two different kinds, incremental or full, the update method is overloaded.

More interesting is 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. *bild hier unbedingt mit workspace -> filerepo -> physfile/translationresult/symboltablemanager und methoden die hier beschrieben werden.*

Aside 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. Thus, all information about a file is accessible from within the File Repository. To obtain all of these results, the class simply invokes the `SymbolTableGenerator` and the `DafnyTranslationUnit`.

---

```
1 public interface FileRepository
2 {
3     PhysicalFile PhysicalFile { get; }
4     TranslationResult Result { get; }
5     ISymbolTableManager SymbolTableManager { get; }
6     void UpdateFile(string sourceCodeOfFile);
7     void UpdateFile(Container<TextDocumentContentChangeEvent> changes);
8 }
```

---

Listing 27: Handling Compilation

### 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 preexisting projects, but as part of this bachelor thesis it was refactored and simplified.

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

1. It is checked that the instance has never been used before. This is, since the error reporter must be 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 calculation. The configuration is shown in 28.
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.

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 provided 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.

*evtl bild, so ablaufdiagramm bild mit TranslationResult + TranslationStatus wäre gut ich denke*

---

```
1 private void SetUpDafnyOptions()  
2 {  
3     DafnyOptions.Install(new DafnyOptions(_reporter));  
4     DafnyOptions.Clo.ApplyDefaultOptions();  
5     DafnyOptions.O.ModelViewFile = FileAndFolderLocations.modelBVD;  
6  
7 }
```

---

Listing 28: Setting up Dafny Options

### 6.3 Symbol Table

This package provides four components: *Ich finde diese Aufzählungen haben wenig Aussage. Zusätzlich noch ein Klassendiagramm mässiges Bild mit Typ. Im Text noch beschreiben "was das ist/tut". todo*

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

**Symbol Information** This is a component that summarizes all information about a symbol. Aside the name, this also includes the location, the parent, the declaration, children, and so forth. The class contains a lot of properties, but in exchange, it provides any information that is required. The most important properties are:

- Name
- File
- Position in File
- Body Size, if any
- Kind and Type
- Link to Parent Symbol
- Link to Declaration Symbol
- Hash with all Children Symbols (Only Declarations)

- List with all Descendants (Any symbol occurring in the body)
- List with all Usages of the symbol
- BaseClasses
- The associated module
- Link to the associated default class for quick access.

The provided properties are supposed to make work with the symbols as comfortable as possible. For example, if a symbol's definition cannot be found, the property with the associated default class can be used to search the symbol there. This is convenient, since the default class is in the global namespace and not a direct ancestor of a symbol.

Technically, the symbol table is more a tree, then table. The data structure is double linked. Each symbol knows about its descendants, but also about its ancestor. Navigating to either one can thus be done in  $O(1)$ . If the name of a descendant is known, navigating to the symbol can also be done in  $O(1)$  due to the hash map. For example, if a module 'M' contains a class 'C', and within the class there is a method 'foo', one can simply start from the root symbol and navigate through the hash maps. The `[]` was overloaded to make this as convenient as possible:

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

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

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

Listing 29: Example Code Regarding Symbol Information

The symbol 'foo' profits by almost all properties. It can have children (the variable x), it has some parent, it can be used. Since foo is a method declaration, the declaration property of the symbol does not make sense. In this case, it actually just points to itself. The declaration of x in the second line will have many null values within the symbol information. While the parent is foo, it can not have any children. Since it is a variable definition, it 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 just null for that symbol.

As a consequence, operating on the symbol table must be done with possible null reference expectations in mind.

Aside the properties, the `SymbolInformation` also offers a public method. This method will check if a certain location is wrapped by the symbol. This namely answers the question, if for example line 5, column 2 is within the symbol's body.

**Symbol Table Creation** The symbol table generator accepts a precompiled dafny program in the constructor. It should be available after the dafny translation unit has been executed. The generator offers a public method `GenerateSymbolTable`. It will then 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.

Then, all modules (similar to C# or C++ namespaces) will be extracted out of the dafny program. 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 the modules. The proper parent symbol will be extracted. For a top level module, this is just the root symbol. Otherwise, for nested modules, the parent module will be located. Finally, the module will accept the declaration visitor. The visitor is described later. Once it completed, all declarations are registered in the symbol table. A second iteration is then started, using the deep visitor, which will ignore declarations but run through all method bodies and take care of symbol usages.

**Visitor** 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, the method also contains an `ensures` statement, which may contain variables or such. 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.EnsureExpressions) 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 last section, it was said that two runs are actually performed. One to capture all declarations, and one to visit all method bodies. Thus, the visitor is having a boolean `GoesDeep`, which decides if bodies like occurring at a method are visited at all. The final `Accept` method for a method looks as shown below. The method is shortened, there are more clauses like for example 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.Body)
11        {
12            stmt.Accept(v);
13        }
14    }
15    v.Leave(this);
16 }
```

---

Listing 30: Accepting a Visitor

Note that the method is marked with the `override` keyword. This is, since every `AST-Element` is either a statement or an expression, among others. In case we left out a specific `AST` element, just a general `accept` method is defined for statements, as well as expressions. However, for `AST` elements that we support, a more specific method is supposed to be used. (hmm abschnitt weg? is gebrabbelt)

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

The `Visitor` itself will now actually build up the symbol table. For that, it stores the current scope in a property.

For example, when visiting a method, the parent scope is always some kind of class that was visited before. When the method is finally visited, the property 'parent' can just be set with the scope the visitor is in. Since the method itself will have its own body, the new scope can then be set to a method. This will attach all symbols to the method. Once the method is done, `Leave()` is called, which will reset the scope to the class.

The Visit method of the first visitor, which only takes care of declarations, will itself just create a symbol, and attach it to the current scope. All required information can be taken from the AST element 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 31: Visiting a Method

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 scope of the visitor is set to the method for future visitations.

The second visitor will also visit declarations, but no longer create a symbol for them. Instead, just the proper scope will be set. The second visitor will now also visit method bodies. Within the body, it will encounter local variables. These were not caught by the first visitor, but this is ok, since local variables are not accessible before they weren't declared. Furthermore, symbol usages such as method calls or variable usages are now encountered. The visitor now has the responsibility, to 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 done:

- Where am I declared?
- Add usage to my declaration

To find the declaration, the symbol table navigator can already be used. It is discussed below. The navigator will just iterate from parent to parent and will return the first symbol, that is a declaration, and matches the name. Challenges occurred when a symbol is defined in global scope or in a base class. Both difficulties were resolved by adding separate checks for them.

Once the declaration is found, it is simple to add the just newly created symbol as one of the declaration's usages.

The tricky cast is indeed to find a symbols declaration, which could be extracted to the navigator:

```
1 protected ISymbol FindDeclaration(string target, ISymbol scope, Kind kind)
2 {
3     INavigator navigator = new SymbolTableNavigator();
4     bool filter(ISymbol s) => s.Name == target && s.IsDeclaration && s.Kind ==
        kind;
5     return navigator.BottomUpFirst(scope, filter);
6 }
```

Listing 32: Finding a Declaration

This snippet assigns the navigator to move upwards in scope and search for a symbol, that is a declaration and matches in name and kind. If found, the proper symbol must be the according declaration.

**Symbol Table Navigator** To operate on the (partially) constructed symbol table, a separate component to navigate was created. It has basically 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 moves upwards the tree and searches a specific symbol.

*2do Bild von nem baum und dann wie es so hoch und runter geht, Beispiel, Visualisierung einbauen fuer die Laufzeitanalyse.*

*Keine apostrophe als abkürzungen. keine "you" sondern indirekte ansprachen. one.*

Both options are implemented so that they can return a single, first match, or any symbols that match a criterion. To illustrate this, let us have a look at two examples.

- You want to know what symbol is at the cursor position. You call `TopDown` and pass the `rootSymbol`, as well as the cursor position as arguments. The `rootSymbol` has 3 modules attached to it. One ranging from line 1 to 20, another ranging from line 21 to 40. The algorithm will now decide, in which of the two modules a further search is worthwhile. If the cursor is located at line 25, it will continue to search in the second module. This can simply be done by calling the same function recursively, handing the second module as the entry point for the recursive search. Within the recursive call, the proper class will be found, and so on. Default namespaces and default classes had to be treated separately for this case.
- To build up autocompletion suggestions, you want to know what declarations are available at the symbol you found just before. Thus, you navigate to the parent symbol, and then you collect all children of that symbol. Then, again, go to the next parent and continue like that until the `rootSymbol` is reached. Again, the problem can be solved using recursion.

Note that this navigation can be executed very fast.

**Symbol Table Manager** The manager is a rather simple component and can be used as an access point for the user of the symbol table. It is constructed by handing over a root symbol of a fully generated symbol table. It then offers methods such as `ISymbol GetSymbolByPosition(Uri file, int line, int character)`. That method will then just create a navigator and use the (maybe rather complex navigator) to provide the user with the desired result.



### 6.3.1 Core

In this package, the actual task of providing results for the server is done. Often, not much code is necessary, since the symbol table provides all necessary information. In this subchapter, a few features are explained more in detail.

#### Goto Definition

Goto Definition is a very simple feature once the symbol table is created. As parameters, it receives a location in a file. All this provider does, is ask the `SymbolTableManager` what symbol is at that position, jumps to the declaration of that symbol, and converts the declaration position to the proper response format. To provide a better usability, it does a few further checks, such as checking if the symbol already was a definition or if no result could be found at all. The results are stored in an `Outcome` property, which the handler can use to send proper client feedback.

---

```
1 public LocationOrLocationLinks GetDefinitionLocation(Uri uri, int line, int
   col)
2 {
3     List<LocationOrLocationLink> links = new List<LocationOrLocationLink>();
4     var symbol = _manager.GetSymbolByPosition(uri, line, col);
5     if (symbol == null)
6     {
7         Outcome = DefinitionsOutcome.NotFound;
8         return new LocationOrLocationLinks();
9     }
10    if (symbol.IsDeclaration)
11    {
12        Outcome = DefinitionsOutcome.WasAlreadyDefintion;
13    }
14    var originSymbol = symbol.DeclarationOrigin;
15    Position position = new Position((long)originSymbol.Line - 1, (long)
        originSymbol.ColumnStart - 1);
16    Range range = new Range { Start = position, End = position };
17    var location = new Location { Uri = originSymbol.FileUri, Range = range };
18    links.Add(new LocationOrLocationLink(location));
19    Outcome = DefinitionsOutcome.Success;
20    return new LocationOrLocationLinks(links);
21 }
```

---

Listing 33: Providing Goto Definition

**DiagnosticProvider** This component is called everytime a document is updated. It accepts a `FileRepository` as an argument. Within the repository, the translation results are stored, including the diagnostics. This component will read the diagnostic, convert them to an LSP-suitable format, add usability information, and finally send the result back to the client.

**HoverProvider** This component ist very simple. It just requests the `SymbolTableManager` to provide the symbol at the hover location. The hover location is passed as an argument within the request. Afterwards, basic information about the symbol is assembled and returned. That information includes

- A quick summary, including the name and the location
- Symbol Kind

- Symbol Type
- Parent Symbol
- Declaration Origin

**RenameProvider** Rename is another feature that profits strongly by the symbol table. Again the feature requests the symbol at the cursor. Afterwards, it jumps to its declaration. The declaration has all usages of the symbol stored, and thus, all occurrences are known. The provider will now just assemble a `WorkspaceEdit` and return it. The `WorkspaceEdit` contains the new name and as well all `Ranges`, where the name has to be applied.

Additionally, the rename provider performs a few checks, if the new name is valid. The checks are

- Name must not start with an underscore
- Name must not be a reserved word, such as 'method'
- Name must not contain any other than alphanumerical character or underscores

The last check is stronger than it needs to be, but since special characters are extremely uncommon in programming, it is well suitable. It was mainly implemented to prohibit brackets in names.

Eigentliche logik. implementiert interface. Die Features ev einzeln druch, msus man fast, hat ja vieles interessantes, z.b. so diagnostic conversion (wobei xcdas bei dafny access is)

### Completion

todo Recht komplex Wir supporten drei cases; nennen. Beschreiben. Ablauf. Diagramme wie in der SA sind nützlich.

### Code Lens

todo komplett server seitig nicht übers lsp möglich reicht daten für codelens popup an den client weiter; ruft vs code feature auf. beide aspekte werden durch integration tests abgedeckt?

### Compilation

Every time a file gets updated, the whole Dafny backend is triggered and the results are stored in the `FileRepository`. This also includes the precompiled `DafnyProgram`. The compilation provider will take advantage of that and use the precompiled item, and just hand it forwards to the 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 by using

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

Listing 34: Calling the Dafny Compiler

The provider will check, if any errors occurred and return the outcome within a wrapper class.

### Counter Example

This feature bases 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 tells how the variables need to be set at the beginning to achieve a counter example.

This provider reads the model file, uses the Boogie backend to convert into a useful format. Afterwards it extracts the initial state from the model. Out of the remaining model, all key-value-pairs containing useful information are extracted, assembled and returned. The component will also transform information into a more human readable format, e.g.  $((-12)) \rightarrow -12$ . Furthermore, 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]`.

**Symbol Table Runtime -> ich überlege grad ob das besser bei results ist sonst einfach hochtun.. später schauen wenn results auch steht.**

The features themselves are primarily based on the symbol table. In particular auto completion, go to definition, CodeLens, hover information and rename.

Due to the structure of our Symbol table (which is updated after every change in a Dafny file) the basic information is provided by references. Each symbol carries references to its child symbols, to the parent symbol, to the original declaration and much more information. All these references were prepared when the symbol table was created. You can therefore call them immediately (runtime  $O(1)$ ). *Das steht alles schon oben...*

The difficulty lies in finding the "entry symbol".

The navigation component described above is used for this. The system uses the cursor position to find the deepest symbol that encloses the cursor position. This symbol is the entry point. And to find this symbol, the longest runtime is required for the features - apart from the creation of the actual symbol table of course.

**Das hier nach results (klassischer vorher nachher vergleich)**

Since we have object information (and not just strings anymore) with our self-written symbol table, the whole position to string parsing was dropped.

In our old version we had to find out from the current cursor position which word in the code could be meant. Then we iterated over the whole symbol table and checked if there was a symbol with the same string as name. The first match was looked at as a meant symbol.

Our new design eliminates all of this effort and avoidable assumptions. We access the currently marked symbol directly via the position data. String comparisons and corresponding string extractions are completely eliminated. This leads to better performance and above all to reliable symbol references.

To enable efficient access to the entry points, 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 us to access child symbols with a runtime of  $O(1)$ . Since every symbol also has a link to its parent, navigation in both ways can be done within  $O(1)$ .

## 6.4 Testing

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

### 6.4.1 Unit Tests

Our core logic components all use an interface. When using components, programming is done exclusively against the corresponding interface.

This encapsulation of components allows essential core logic to be efficiently covered by unit tests.

Due to the fact that our unit tests are run significantly faster than the integration tests, it is very important that unit tests are written. Due to the fast run-through they are a very good support in the ongoing development. If an error creeps in due to a change, it will be detected immediately by the existing tests. If an error needs to be corrected or a function needs to be extended, a Test-Driven-Development approach can be used to work in a goal-oriented manner.

In our unit tests, primarily isolated and more complex logic blocks are tested. These are not only covered by a Code Coverage metric, as described in the chapter "Project Management" `todo REFERENZ`, but we

were also busy covering borderline cases by testing. TODO DAS IST NICHT DAS KORREKTE WORT FÜR GRENWERTTESTS(?)

Of course the interaction of the components is also tested. This will be discussed in the following chapter regarding integration tests.

### 6.4.2 Integration Tests

As described in chapter 5.4, a very nice test architecture was builded for integration tests. Each feature could be tested by creating a base class. The base class usually contains one method `Run` and another one `Verify`. The first one uses the inherited client-server infrastructure, opens a Dafny file, sends the according request, and collects the results. The following example is representative for such a method:

---

```
1 public void Run(string testfile, int lineInEditor, int colInEditor, string
   newText = "newText")
2 {
3     Client.TextDocument.DidOpen(testfile, "dfy");
4     RenameParams p = new RenameParams()
5     {
6         NewName = newText,
7         Position = new Position(lineInEditor-1, colInEditor-1),
8         TextDocument = new TextDocumentIdentifier(new Uri(testfile))
9     };
10    var response = Client.SendRequest<WorkspaceEdit>("textDocument/rename", p,
        CancellationSource.Token);
11    result = response.Result;
12 }
```

---

Listing 35: Finding a Declaration

The `Verify` method will just compare the results against the provided expectation. Often, rather complex data structures with a lot of nested classes come into play. To be able to compare them easily, most of them are just converted to a string representation, which can easily be dealt with. This was done using extension methods located in the `TestCommons` project.

A test itself is created very easily with all this infrastructure. One simply inherits from the base class, and most tests can be written in just a few lines. For example, a rename test could look like this:

---

```
1 [Test]
2 public void LocalVariableUsage()
3 {
4     Run(Files.rn_scope, 9, 17);
5     List<string> expected = new List<string>()
6     {
7         "newText at L7:C12 - L7:C15",
8         "newText at L8:C14 - L8:C17",
9         "newText at L12:C18 - L12:C21",
10        "newText at L20:C14 - L20:C17"
11    };
12 }
```

---

```
12     Verify(expected);  
13 }
```

---

Listing 36: Sample Integration Test

Note that the test is kept as concise as possible. The tester does not even have to care about the result provided by `Run`, since that method will store the result inside a class member and clean it after the test is done.

Since all tests base on actual Dafny files, a dedicated subfolder was created to store them. All files can be referenced globally from within the `TestCommons` project, where also the base class for all integration tests is located.

**m m m m m m**

## 6.5 Usability Test

Nein. Das ist Implementation. Bei Implementation Test gemacht; Implementation verbessert. todo  
Entgegen der ursprünglichen Annahme, nicht "alt und neu" als vergleich testen lassen (fehlende probanden)  
Testing der allgemeinen Benutzbarkeit unserer Features (verständlichkeit, nützlichkeit) Testing der VSCode  
Universe Integration (VSCode Benutzer)  
Auswertung der Tests; Kernpunkte und Verbesserungsvorschlägathier Drauf eingehen, welches Feedback  
umgesetzt wurde, welches nicht, warum, outlooks.  
Tests in den Anhang aufnehmen!! Und dann hier entsprechend referenzieren.

## 7 Results

*bei der SA hatte er hier gesagt: was wir unter result hatten ist 'user guide' und was wir unter conclusion hatten war results. wir dürfen hier wohl drum schon kritisch die features ansehen. dann können wir bei conclusion eher so 'es hat spass gmacht und war eine etolle erfahrung mimi fühlsch mi gsürsch mi' - ich glabu das erwartet er eher. und vlt noch so outlook und so bei conclusion - ake 'jetzt ist es einfacher, neue ffeatures hinzuzufügen, weil die symboletable so geil ist'.* In this chapter, the achieved results of our work are described and critically reflected. The current version of the plugin supports numerous features:

1. Syntax Highlighting
2. Code Verification
3. Compilation
4. Counter Example
5. Hover Information
6. Goto Definition
7. Rename
8. Code Lens
9. Autocompletion
10. mehr?

Aside the features, also architectural improvements are discussed, which facilitate further development of the project for other developers.

### 7.1 Accessibility

Some of the features are just active by default, namely the syntax highlighting and the code verification. Hover information triggers everytime the user moves his mouse over a symbol. This should happen by chance, or by accident. Auto completion triggers whenever the user enters a dot or types a `new` and will thus be discovered automatically. Furthermore, the user can type `CTRL+Space` to trigger auto completion. Further functionality, such as goto definition, compile or rename, can be accessed via the context menu. Within the context menu, hotkeys are displayed, thus the user automatically learns how to increase his efficiency. Although there are a lot of different features, they are visible and perceivable to the enduser without the necessity of a tutorial. The user can start right away. *Ref auf usability tests. evtl noch ein bisschen mehr 'es ist voll geil zugänglich'.*

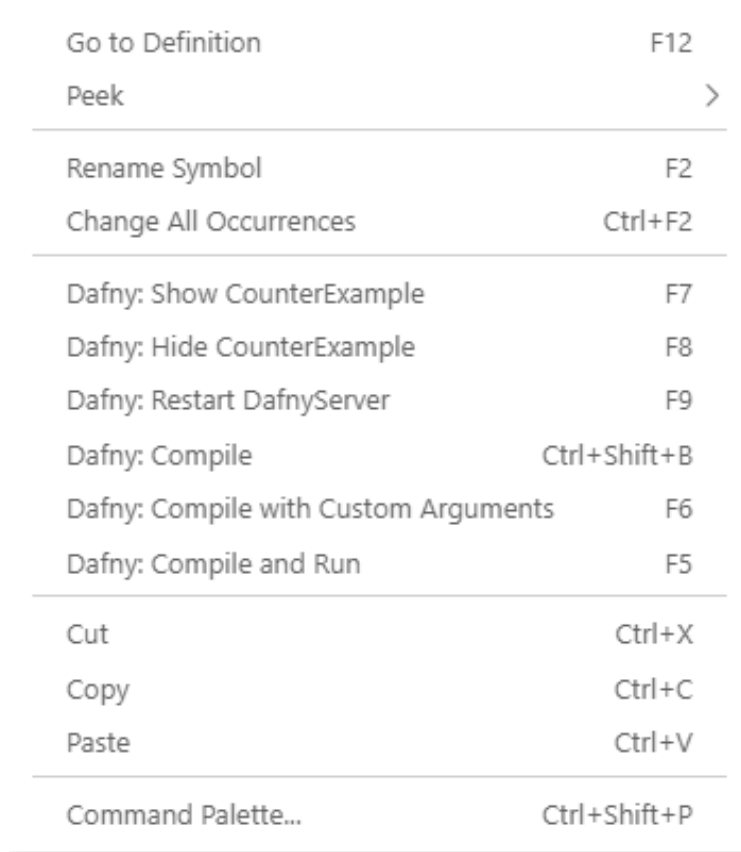


Figure 20: Plugin Context Menu

## 7.2 Intro

hier noch so: feature sind über context menü erreichbar hotkey einsehbar -> automatisches user learning für faster feature access -> user sieht auch, was es alles so gibt voll geil. advanced shit, wie etwa farbe konfigurierbarkeit, launch arguments, lo gplacement ist dann im readme.md für die advanced user, sodass der otto normalo SE2 student nicht überflutet wird.

andere feautres laufen einfach so, wie etwa syntax und veri

## 7.3 Sntax Highlighting

Syntax highlighting is realized by a given Dafny grammar file. The file contains regex expressions defining the highlights. It is provided by Dafny *link reffen, ich find den link nicht* and could simply be downloaded. The feature was already implemented in the preceding term project [3], thus no further actions 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 21: 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.

## 7.4 Verification

Verification was already implemented in the preceding project [3]. Previously, the feature held some major flaws. It only reported logical errors. Syntax errors were not reported. The code just invoked the preexisting Dafny Translation Unit, taken over by the original project [1]. Errors reported by Dafny were no longer respected. Only errors reported by Boogie were tapped and reported to the user.

Verification was reworked completely in course of this bachelor thesis. First of all, Dafny code 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. This is something the original project [1] did not do and was actually already issued on the official Dafny git repository [23].

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

*Screenshot mit warning... ich krieg kein unreachable code warning hin wtf xD  
noch todo im code: Ganzer Block underlined oder immer noch nur das {? hier noch ergänzen oder 'gekonnt  
verschweigen'*

## 7.5 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 relatively fast.

The user also has the option to apply custom compilation arguments. These can be directly set within VSCode.

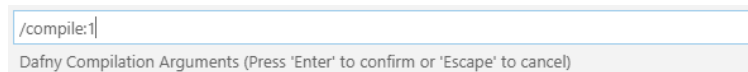


Figure 22: Custom Compilation Arguments

These are directly handed to the Dafny options parser and are directly applied within the Dafny engine. Since compilation uses the precompiled Dafny program, compilation arguments that would actually affect the verification process have no effect. This is something that could be resolved by just restarting the



verification process if custom arguments were given. Unfortunately, due to the hard project deadline, this could not be implemented anymore.

Previously, compilation was implemented so that it just started a subprocess, launching `Dafny.exe` with any custom arguments given. This solution was obviously not integrated at all. To obtain compilation results, the console output of the subprocess was parsed and reported to the user.

The current implementation is completely integrated, both, in terms of invocation and result reporting.

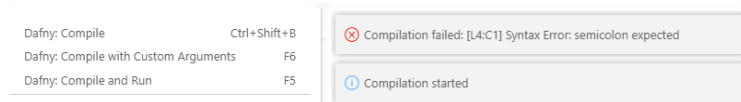


Figure 23: Compilation Context Menu and User Feedback

A little flaw of this feature is, that if the user chooses the option to `Compile And Run`, while the compilation process finishes very quickly, the actual launch of the executable takes quite some time. However, this is nothing in scope of the language server. If the executable is launched manually within an other console outside of Visual Studio Code, it also takes quite some time until the program starts.

## 7.6 Counter Example

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

For this, the related `model.bvd` file was studied. It is a quite cryptic file and getting a full understanding of it would be very complex. However, an `initial state` was located which seemed exactly that part of information, that the user is interested in. Thus, unlike previous 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 reformed 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 24: Counter Example Representation before (top) and after (bottom)

Figure 24 shows a comparison of the counter example feature between the initial state and at the end of the project. The representation is much cleaner and easier to catch. Also note, that more room is given so that it does not have a clumsy effect on the user's eye. *die HCID Trulla reffen? Mehr raum?*

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 (dark or light mode). 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 counter example state is buffered, so that once the user switches back to the original window, the counter example will be shown again.

## 7.7 Hover Information

Hover information displays a set of information, whenever the user hovers with his mouse cursor over a code symbol. The feature itself was very simple to implement, since it basically just asks the newly implemented symbol table engine, what symbol at the cursor position is, and finally displays some information about the symbol. It was taken as an exemplary feature, to show off how easy some LSP functionality can now be implemented using the new symbol table. Similar to hover information, other features like text highlighting could be added as well, which is discussed in chapter 8. Hover may not provide much useful information to the user, but still can be used for debugging. For example, in figure 25 the user can actually find out, where his symbol `field` is declared if he is unsure or what type it has.

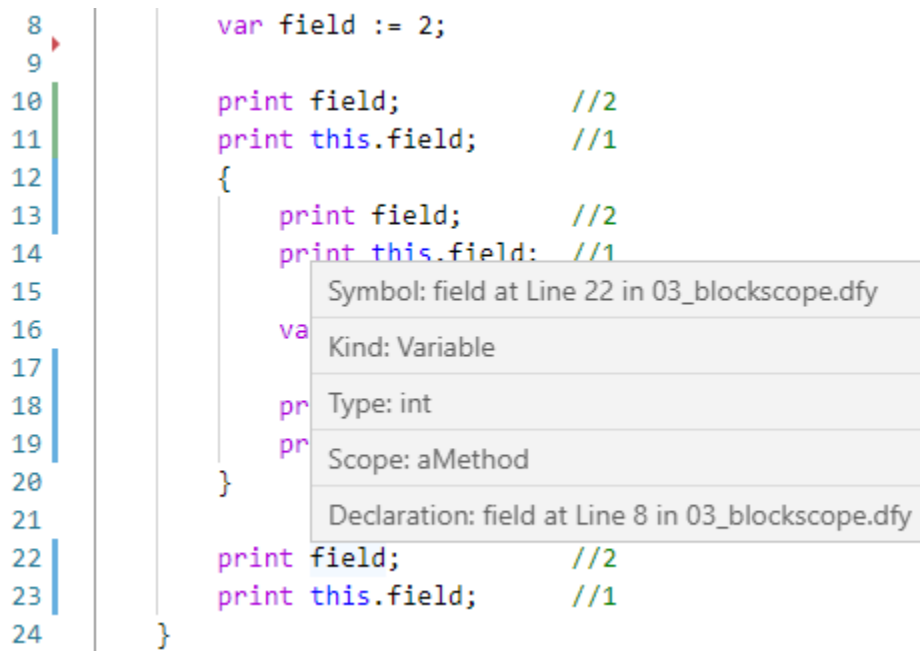


Figure 25: Hover Information Example

## 7.8 Goto Definition

Goto Definition is also a very simple feature. Similar to hover information, it will just ask what symbol is at the cursor's location. Then, it will request the symbol table backend to report the declaration of that symbol. Finally, the declaration origin location is returned to the lsp and the cursor jumps to it. To increase the user experience a bit further, notifications are sent to the user if the symbol was already a declaration, or if there is no valid symbol at the cursor position.

*die msg tut iwie nich xD is aber my bad hab da noch was geändert kürzlich. punkt 2: vscode reported das grad selber bei return null, damüssen wir unsere eigene notification eig net bringen.*

Jumps can also be done accross different files. If the proper 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 accross code that is not within a blockstatement, for example for expressions occuring inside an `ensures`-clause.

```

1 | class Counter {
2 |     var count: nat;
3 |     constructor () { }
4 |     method Reset() modifies this {

```

No definition found for 'method'

Newly used yet. Can you remove a method?

Figure 26: Goto Definition Error Reporting

Compared to the original implementation, goto definition 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 generator.

Runtime could also be improved, since the symbol table generator scans for possible declarations scope by scope, and doesn't just iterate over all symbols. Last but not least, the cursor had to be at the beginning of a symbol to actually recognize the cursor position at all. This has also been improved and the cursor can now be at any spot within a symbol.

Technically, the feature is misnamed. A better name would actually be `Goto Declaration`, since it jumps to the symbol declaration, not definition. LSP would even offer a dedicated handler for `textDocument/declaration` [6]. *ich würd das so lassen aber technisch gesehen müssten wir genau das implementierte interface ändern, der rest ist eh gleich xD vlt könnte man das sogar noch schnell machen*

## 7.9 Rename

Rename is a feature that could only be added, since the symbol table provides all necessary information. Just like hover and goto, the feature itself is very simple. Again it requests the symbol table at the cursor position from the symbol table. Then all occurrences of that symbol are demanded and that is already all information needed. Finally, a proper LSP response has to be assembled and is sent back to the client.

We also tried to provide as much user experience as possible for this feature, 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, if the new symbol name is valid. This means, it must not start with an underscore or it must not be any reserved Dafny word. Also, we limited the allowed new name to alphanumeric 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 new name already exists. Functionality for this is already available by the symbol table, since something similar is needed by autocompletion. Furthermore, it is currently not checked, if the new symbol name starts with a number, which is prohibited. That is also something that could easily be added. *das sind so one-liner im code... könnten wir echt noch schnell umsetzen*

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

Since the symbol table reports occurrences across imported files as well, the rename feature works fine across multiple files.

## 7.10 CodeLens

## 7.11 Automatic Code Completion

## 7.12 Symbol Table

All of the previous features, with the exception of auto completion, do not contain much logic themselves. They just request information from the symbol table and report it back to the client. 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 very important for a nice functionality.

Dafny is a programming language offering a lot of features. Aside common object oriented features, also functional programmatic features are present. Figure ?? 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 show how many AST-elements are actually 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 would have been too time consuming. Since the bachelor thesis has 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 Dafny tutorial, which is done within the software engineering at HSR. This way, the basic concepts of Dafny are supported. 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 chapter 4.7.

*würd hier nicht schreiben, was genau failed, oder? weil das fängt schon bei arrays oder listen an. iwo bei implementation ahben wir ja geschrieben, was wir analysiert ahebn.*

It is subject of further development, to complete the symbol table.

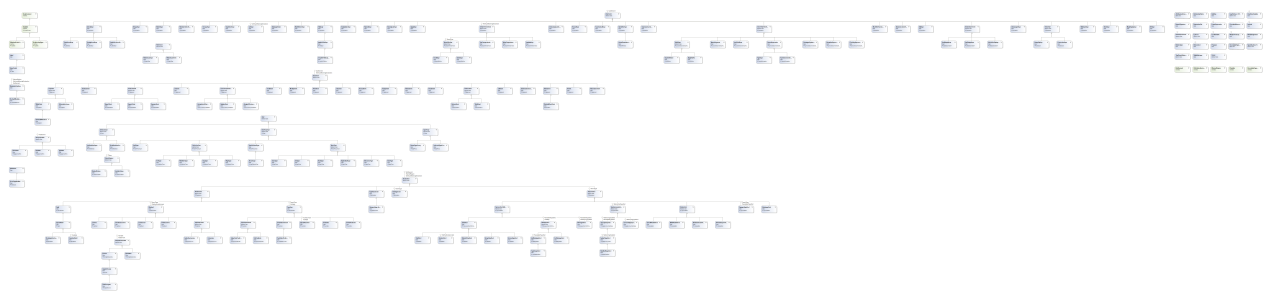


Figure 27: All Dafny AST Classes

### 7.13 Platform Independence / Mono Support for macOS and Linux

*welcher titel ist besser?* Eines der Kernziele war es, Support fuer mehrere Plattformen zu bieten. Dh nebst Windows auch macOS und Linux. Da wir in unserer SA von Core auf Framework umsteigen musste, stand fest, dass wir mono fuer den Support auf Linux und macOS brauchen. (warum in der SA; plficht wegen dafny core. was ist mono)

Leider funktioniert nicht. Anssaetze die wir probiert haben. verschiedene mono versionen, angefragt im slack. antwort erhalten? github issues: allgemein probleme mit linux/mac weil primaer auf windows und gar nicht auf mac getestet wird. (heikle aussage selbs tmit quelle)

[3] [24] [25]

### 7.14 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 be well organized. As seen in figure ??, all dependency are now pointing downwards, just as it should be. Within the picture, dependencies to the Resources project are excluded for more overview. The core layer of the project, containing `SymbolTable`, `Handler` and `Core`, have some sideways dependencies, which could not be avoided completely, since these packages are strongly coupled.

Aside the grob 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 reading, which was one single large class at first. *evtl lassen oder weg, ka wie ich das noch refactftore und ob überhaupt.*

## 8 Conclusion

This chapter contains a conclusion. *2do ironie entfernen*

### 8.1 Project Summary

Overall, the project can be counted as a success. The preexisting features could be improved, and new, major features were added to create a comprehensive user experience. 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 major flaw of the project is, that the symbol table (in particular: the visitor) could not be implemented completely. This allows only for a limited usability of the plugin, namely for the supported Dafny language features as stated in chapter 4.7. However, completing the visitor is not necessarily complex, but rather a diligence task. The reason we couldn't complete the visitor was just the limited time frame of the bachelor thesis. When time is fixed, scope has to be variable, thus we needed to cut out some Dafny language features from the visitor. [26] The foundation to complete it are set though. 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, whenever an exceptional state occurs, which worked out quite well.

Although the symbol table is incomplete, the plugin provides a great experience for someone taking his or her 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, and thus, the project is counted as a success.

Since the symbol table is its own component, the existing features which base on the symbol table will automatically scale, once the symbol table is completely implemented. This means, that once the visitor is completed, the plugin automatically gained a lot of usability and can then also target more advanced users, that use more advanced language features.

The `DafnyServer` component, which originates from old bachelor thesis [1], could be completely superseded by our new `DafnyAccess` package. The code is well organised, splitted in to components, and ready to be merged into the Dafny repository.

### 8.2 Deployment

One goal of the project was the deployment of the project. Fabian Hauser, our co-supervisor prepared already all necessary steps to deploy a preview version. For usage within the software engineering course, a deployment is absolutely viable. However, to target a wide audience, the symbol table implementation should be completed first. Thus, we would only deploy the plugin as a preview version. Nevertheless, the plugin achieved a well enough quality to be ready for deployment.

### 8.3 Outlook

In this chapter, the further development of the project is described.

#### 8.3.1 Completion of the Visitor

As mentioned previously in this section, the visitor should be completed to also target more advanced users. We left that task out, since it is very time consuming. Once this is done, the project instantly gains a lot of quality, since all features scale automatically with the symbol table. While it was said that this is rather a diligence task, some problems may arise though. 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 is not to underestimate. Especially if a new team takes over this task, they first of all have to get into the current project. This is non-trivial, since it has grown quite a lot.

### 8.3.2 LSP Extensions

The language server protocol [6] offers a lot more than what is currently implemented. Many of these features could now be easily 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.

In chapter ?? it was already mentioned, that `textDocument/documentHighlight` is 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 nice first step for a future development team, for example if this project gets further developed within another semester thesis.

Aside highlight, the following LSP requests are also of interest for future development:

- `workspace/workspace/didChangeWatchedFiles`: Actually handle the request (for example remove the file from the buffer if it was deleted) instead of ignoring the change.
- `workspace/symbol`: A request that sends all, project-wide symbols to the client. This could be done relatively easy with the symbol table, for whatever purpose.
- `textDocument/didClose`: Currently, nothing is done. The proper file could actually be removed from the buffer in this case.
- `textDocument/signatureHelp`: A feature that allows to auto-fill the signature of something callable. For this, methods would have to remember their parameters, which could then be automatically filled using this request.
- Differentiation between `textDocument/definition`, `textDocument/declaration`.
- `textDocument/implementation`: Goto 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 More 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. This is also something that could be done with little effort.

Aside quick fixes, the LSP also provides support for refactorings like extractions (extract method, extract variable...) [6]. 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 can also assist 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 feature becomes possible. Băm! *das bām entfernen*

### 8.3.4 Dafny Specific

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 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], is the support for debugging. However, this is a very complex topic and may not fit into the scope of a bachelor thesis.

### 8.3.5 More IDE

Currently, only Visual Studio Code is supported. The idea of the language server protocol is, that the server is isolated and connect to various clients. Thus, a nice extension point would be to implement support for other IDE's, just 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 should actually come out of the box. The client developer would just have to keep an eye on the additional features, namely compile and counter example. These require a minimal amount of extra code. For example, a mechanic has to be found to display a counter example inline.

## 8.4 Achieved Improvements compared to the preceding semester thesis

Regarding the goals provided by Fabian Hauser at the beginning of the thesis, the following goals could be completely achieved:

- Client and Server are split into separate repositories.
- Language Server was rebased to use the newest Dafny upstream
- General Code cleanup was done
- Ready to merge
- Sonar Qube is now running
- Dafny Library is directly integrated by now
- Symbol Table was implemented - clean - geht so.
- Previous features were implemented **cleanly**
- ... tun wir hier wirklich alles listen? kommt mir komisch vor. es steht ja eig schon zuvor im text. grad wenn r dann eben unten schreibt "clean bla, clean bla", short methods.
- korrektur: ja, alles listen, weil wir eig iwrlch alles gemacht haben, je nachdem wie man sein CLEAAN interpretiert.
- clean git branching, hmm, ja, hus thsut.

## 8.5 Metrics

Performance Verbesserungen, Anzahl Code Lines, reduktion der lines/code anzahl / methodenanzahl. "Wie viel chelaner" unsere lösung geworden ist. wie viel schneller. sowas ist wichtig fpr t.corbat. eventuell auch conclusion? ich weiss es wirklich nicht :O doch nach results? implementation?



## 9 Project Management

todo

## 10 Designation of chapters taken from the preexisting term project

The following chapters originated in the preexisting semester thesis "Dafny Server Redesign"[3] and were reincluded in 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.

- Chapter 1 Abstract
  - Paragraph 1 about Dafny
  - Paragraph 2 about the language server protocol communication
- Chapter 3 Introduction
  - Chapter 3.1 except for the paragraph about lemmas
  - Chapter 3.3, partially
- Chapter 4 Analysis
  - Chapter 4.1
  - Chapter 4.2, reworked, better example
- Chapter 7 Results
  - Chapter 7.3

*das partially wirdd ihm sicher nicht gefallen, er wird wissen wollen 'was genau' aber ka wie man das schreiben soll.*



## 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] *Language Server Extension Guide*. URL: <https://code.visualstudio.com/api/language-extensions/language-server-extension-guide>. (Accessed: 15.12.2019).
- [6] *Language Server Protocol Specification*. URL: <https://microsoft.github.io/language-server-protocol/specification>. (Accessed: 26.05.2020).
- [7] *Langserver.org*. URL: <https://langserver.org/>. (Accessed: 05.12.2019).
- [8] *OmniSharp/csharp-language-server-protocol*. URL: <https://github.com/OmniSharp/csharp-language-server-protocol>. (Accessed: 05.12.2019).
- [9] *Martin Björkström - Creating a language server using .NET*. URL: <https://app.slack.com/client/T0RE90CRF/C804W8JHE>. (Accessed: 05.12.2019).
- [10] 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.
- [11] *Functions vs. Methods*. URL: <https://www.engr.mun.ca/~theo/Courses/AlgCoCo/6892-downloads/dafny-notes-010.pdf>. (Accessed: 15.04.2020).
- [12] *SonarCloud for C# Framework Project*. URL: <https://community.sonarsource.com/t/sonarcloud-for-c-framework-project/17132>. (Accessed: 23.03.2020).
- [13] Marcel Hess and Thomas Kistler. *Developer Documentation*. HSR Hochschule für Technik Rapperswil, 2020.
- [14] *OpenCover*. URL: <https://github.com/OpenCover/opencover>. (Accessed: 23.03.2020).
- [15] *monocov*. URL: <https://github.com/mono/monocov>. (Accessed: 23.03.2020).
- [16] *VSCoDe Extension API*. URL: <https://code.visualstudio.com/api>. (Accessed: 22.05.2020).
- [17] *Data Types in TypeScript*. URL: <https://www.geeksforgeeks.org/data-types-in-typescript/>. (Accessed: 15.04.2020).
- [18] *OmniSharp Language Client*. URL: <https://www.nuget.org/packages/OmniSharp.Extensions.LanguageClient/>. (Accessed: 15.04.2020).
- [19] *Nunit CollectionAssert*. URL: <https://github.com/nunit/docs/wiki/Collection-Assert>. (Accessed: 15.04.2020).
- [20] *Newtonsoft Json.NET*. (Accessed: 22.05.2020).
- [21] *CommandLineParser*. (Accessed: 22.05.2020).
- [22] *Serilog*. (Accessed: 22.05.2020).
- [23] *Dafny, Warnings Issue*. URL: <https://github.com/dafny-lang/dafny/issues/168>. (Accessed: 26.05.2020).
- [24] *OmniSharp Slack - Mono*. URL: <https://omnisharp.slack.com/archives/C804W8JHE/p1587578976071500>. (Accessed: 24.04.2020).
- [25] *OmniSharp GitHub Issue - Mono*. URL: <https://github.com/OmniSharp/csharp-language-server-protocol/issues/179>. (Accessed: 24.04.2020).
- [26] Daniel Keller. *Software Engineering Lecture, HSR*. HSR Hochschule für Technik Rapperswil, 2018.

## Anhang (Entwickler Doku)