

1 Cheat Sheet (TMP)

Dieses CheatSheet wird später wieder entfernt.
Hier ist Text auf einer neuen Zeile.

Jetzt ist Text mit einer Zeile Zwischending.
Wegen dem rechten Linebreak füllt diese Zeile den ganzen horizontalen Raum. Nach dem Linebreak kommt eine neue Zeile. Neue Zeile. Bigskip scheint einfach eine neue Zeile zu sein.

Ich bin **fett** und *kursiv*. Inline code geht mit `texttt` oder analog mit `code` weil man die 3t's eh versaut.
C# muss man escapen mit dem command `\Csharp`.

1.1 Untertitel

1.1.1 Das ist die tiefste Titelebene

Ich bin Text.



Figure 1: My caption

Davor ein Bild.
Mehr dazu in Abbildung 1.

1.2 Quellen

Und das wäre ein zweiter Absatz [1].
Wie einer auf 20min sagte:[2]

Immer mehr europäische Länder verhängen im Kampf gegen das Virus eine Ausgangssperre.

Beachten sie die Fussnote¹

¹Ich bin die Fussnote

1.3 Aufzählung

- Erstens
- Zweitens

1. Erstens

2. Zweitens

Erstens

Zweitens

1.4 Tabelle

Col1	Col2	Col2	Col3
1	6	87837	787
2	7	78	5415
3	545	778	7507
4	545	18744	7560
5	88	788	6344

Figure 2: My table

Das war also Tablle 1.4.

1.5 Code

Fogelnd Code in C#

```
1      public void SendInformation(string msg)
2      {
3          SendMessage("INFO", msg);
4      }
```

Listing 1: My Caption

```
1      DafnyCode() {}
```

Listing 2: My Caption

Da wär jetzt so code, siehe Listing 2.

The user needs to write this in front of the variable myVariable.

1.6 Referenced Section

You can read more about references in section 1.6

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

2 Abstract

(Abschnitt 1-2 teilweise übernommen)

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 implemented to access Dafny-specific static analysis features. For example, if Dafny cannot prove a postcondition, the code will be highlighted and a counter example is shown. Furthermore, it provides access to code compilation, auto completion suggestions and various automated refactorings.

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 semester project, the language server was integrated into the Dafny backend to make the JSON-interface obsolete.

This bachelor thesis is a direct continuation of the preceding term project. It had two major goals:

- Improvement of previously implemented features in usability, stability and reliability.
- Implementation of a symbol table to facilitate the development of navigational features.

The symbol table was 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 a small usage statistic.

Using the visitor pattern, the Dafny abstract syntax tree is visited to generate the symbol table. After its generation, the symbol table can be navigated from top to bottom - for example to search for a certain symbol - or from bottom to top - for example to locate all available declared symbols in a scope. Every symbol contains information about its parent, its children and its declaration. Thus, the features goto definition, rename, code lens and auto completion were very simple to implement.

Aside features based on the symbol table, preexisting functionality was revisited as well. The verification and compilation processes were simplified by creating a dedicated translation unit. Its results are buffered for efficient access. Unlike prior versions, warnings are now designated as well. Counter examples are displayed in a simpler matter. By hovering over a symbol, the user receives basic information, such as the symbol type. All features will now also work accross multiple files and namespaces.

Table of Contents

1	Cheat Sheet (TMP)	1
1.1	Untertitel	1
1.1.1	Das ist die tiefste Titelebene	1
1.2	Quellen	1
1.3	Aufzählung	2
1.4	Tabelle	2
1.5	Code	2
1.6	Referenced Section	2
2	Abstract	1
3	Management Summary and Introduction	4
3.1	Dafny	4
3.2	Initial Solution	5
3.3	Feature Set	5
3.4	Goals	6
3.5	Results	6
3.6	Outlook	7
4	Analysis	8
4.1	Language Server Protocol	8
4.1.1	Message Types	8
4.1.2	Communication Example	9
4.1.3	Message Example	9
4.2	OmniSharp	10
4.2.1	Basic OmniSharp Usage	10
4.2.2	Custom LSP Messages	11
4.3	Dafny Language Features	11
4.3.1	Modules	12
4.3.2	Functions and Methods	13
4.3.3	Hiding	13
4.3.4	Overloading	14
4.3.5	Shadowing	14
4.4	Symbol Table	15
4.4.1	Requirements for the symbol table	15
4.4.2	Dafny Symbols	17
4.5	Visitor	17
4.6	Dafny Expression and Statement Types	18
4.6.1	Expressions	18
4.6.2	Statements	19
4.6.3	Declarations	19
4.7	Dafny AST Implementation	19
4.8	Continuous Integration (CI)	20
4.8.1	Initial Situation	20
4.8.2	Aimed Solution	20
4.8.3	Docker	21
4.8.4	2do - Kapitelaufteilung komisch	21

5	Design	23
5.1	Client	23
5.1.1	Initial Situation	23
5.1.2	New Architecture	24
5.1.3	Components	25
5.1.4	Logic	26
5.1.5	Types in TypeScript	27
5.2	Integration Tests	27
5.2.1	Dafny Test Files	27
5.2.2	String Converters	28
5.2.3	Test Architecture	29
6	Implementation	31
6.1	Symbol Table	31
6.1.1	Feature Support	31
6.1.2	Code Review	31
6.2	Client Code Review	31
6.3	Runtime Analysis of the Essential Server Components	31
6.3.1	Generation of the Symbol Table	31
6.3.2	Use of the Symbol Table	31
6.3.3	Corresponding Duration of the Individual Features	32
6.4	Usability Test	32
6.5	Mono Support for macOS and Linux	32
7	Result	33
7.1	Features for the Plugin User	33
7.1.1	Compile	33
7.1.2	Counter Example	33
7.1.3	Code Verification	33
7.1.4	CodeLens	33
7.1.5	Automatic Completion	33
7.1.6	Hover Information	33
7.1.7	Rename	33
7.2	Achieved Improvements for Further Development	33
8	Conclusion	34
9	Project Management	35
10	Designation of chapters taken from the preexisting term project	36
	Glossar	37
	References	38
	Anhang	39

3 Management Summary and Introduction

In this chapter, the technologies touched by this bachelor thesis are explained to provide the reader with the necessary context. Afterwards the motivation and the goals of the thesis are stated in more detail.

3.1 Dafny

copy pasta oben, das mit lemma is neu

Dafny is a compiled language that targets C# which can prove formal correctness.[3] Dafny bases on the language “Boogie”, which uses the Z3 automated theorem prover for discharging proof obligations.[3] That means, that a programmer can define a precondition - a fact that is just given at the start of the code. The postcondition on the other hand is a statement that must be true after the code has been executed. The postcondition is also defined by the programmer. In other words, under a given premise, the code will manipulate data only thus far, so that also the postcondition will be satisfied. Dafny will formally proof 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. Thus, we ensure, that b must be negative, which is the postcondition.

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

Listing 3: Simple Dafny Example

This example is of course trivial. In a real project, correctness is not that obvious. But with Dafny, a programmer can be sure if his or her program is correct. Since the proof is done with formal, mathematical methods, the correctness is guaranteed.

Abschchnitt wirklich needed? 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. 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 4: Lemmas

3.2 Initial Solution

In a previous bachelor thesis by Markus Schaden and Rafael Krucker, a plugin for Visual Studio Code was created to support Dafny.[1] The plugin was particularly appreciated by the "HSR Correctness Lab"[4] to make coding in Dafny easier. The preexisting solution used a proprietary JSON-interface to communicate with the Dafny server. Dafny's verification results were directly parsed by Dafny's console output. Thus, functionality was limited to what Dafny printed onto the console.

In the preceding semester project[5], the language server was integrated into the Dafny backend. Thus, any functionality was directly accessible and the proprietary JSON-interface, as well as console parsing could be omitted. All features had to be reimplemented to satisfy the new architectural layout.

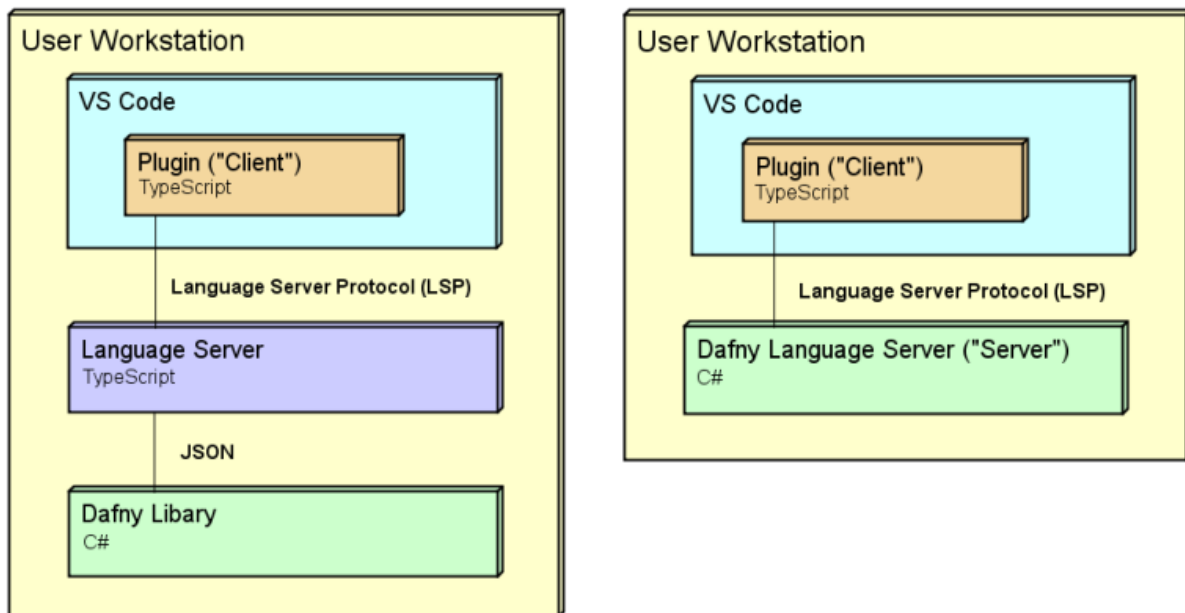


Figure 3: Architecture before (left) and after (right) the preceding term project

alles neu ab hier

3.3 Feature Set

An integrated development environment (IDE) can offer numerous features - the options are nearly unlimited. Influenced by the preexisting thesis, the following features were subject to this bachelor thesis:

- Syntax highlighting

- Verification, highlighting of errors
- Compilation
- Show Counter Example
- Code Lens
- Auto Completion
- Renaming
- Hover Information

Aside the latter two, all features were already implemented within the preceding term project. However, many of them contained some flaws which are further stated in the corresponding essay [5].

3.4 Goals

This bachelor thesis includes two major objectives. In the first segment of the project, the preexisting features that do not depend on the symbol table shall be improved. Also, leftovers of the preceding term project should be completed. This included:

- CI: Install quality measures
- CI: Implement integration tests
- Counter Example: Simplify representation
- Verification: Use a workspace buffer to store the verification results
- Verification: Display warnings as well, not only errors.
- Compilation: Finish Dafny integration, use buffered verification results

Afterwards, a symbol table had to be implemented. During Dafny's compilation process, an abstract syntax tree (AST) is generated though, but it does not contain all information that was required for our feature set. For example:

- to go to a symbol's definition, every name segment should know about where it's declared
- to provide proper auto complete suggestions, all available declarations in a scope have to be known
- to rename a symbol, all occurrences of a symbol must be stored
- to display code lens, all usages of a declaration must be noted

Thus, a symbol table containing all of this information had to be implemented.

3.5 Results

Without exception, all of the preexisting features could be improved.

The Dafny verification process follows now a clear structure. First, the Dafny lexer is called. Then, the Dafny resolver performs semantic checks. Then, the Dafny program is translated into Boogie programs which are then logically verified. Any errors during this process are collected and properly displayed to the user. Intermediary compilation results are stored for later reuse. Previously, only logical errors were displayed. Warnings or Informations were not displayed at all.

Compilation will use the precompiled result and is much faster by now. The user can easily enter custom compilation arguments within the Visual Studio Code client. The representation of counter examples is now

less cryptic and easier to read.

By using the visitor pattern, the Dafny AST could be traversed. While navigating through it, a symbol table is built in the form of a tree. Each symbol is a tree node and stores its child nodes during visitation. Aside from child-parent relationships, symbol usages are counted too, as well as declarations are resolved. Thus, every occurring name segment in the Dafny code contains the following information:

- Which symbol is my parent? For example, this could be a method body or a while loop, or just a block scope introduced by `{ · · }`
- If I am a declaration, where am I used?
- If I am not a declaration, where am I declared?
- If I contain a body, which symbols are declared within my body?
- If I contain a body, which symbols occur at all within my body? This is, declarations and usages.

Thus, a feature like goto definition can just call the information about the declaring symbol and the cursor can jump to it. Thus, compared to the preexisting features, the following improvements could be achieved:

- Goto Definition works now with respect to scopes and will not just jump to the first name match.
- Auto completion works with respect to scopes and will also guess whatever the user is interested in a class, for example after a `new`.
- Code Lens no longer trivially counts name matches. Instead it shows correct usage counts and previews can be displayed.

3.6 Outlook

While the quality of the features, as well as the general code quality could be massively improved, the functionality of the project could be improved even further. Ideas include:

- Automatic generation of contracts
- Debugging
- Create clients for other IDE's.

Aside from the widening of the feature range, it is definitely necessary to complete the visitor, which currently only traverses the most important AST node. This was due to the limited time frame of the bachelor thesis. Nevertheless, the plugin is of a nice quality and may be deployed into the VSCode marketplace. Thus, future students can work with it and make their first steps in the Dafny programming language using our plugin.

2do: iwo den Satz "Zielgruppe die HSR Studenten" einbauen. "Messbarkeit von Erfolg." hats am schluss kurz aber nja, evtl noch etwas deutlich iwo. 2do: allfeatures in one bild? hat da noch platz überhaupt xD

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.

in der sa war hier noch 'wie macht mane in vscode plugin tutorial'... das fänd ich gehört eigentlichs chon auch hier hin, aber es gibt halt keinerlei informaitonen, also hat kein bsp und nix. aka es sagt nicht, was man jetzt wirklich tun muss. wills tdu marcel evtl noch kurz was schreiben aka: man muss ein extension.ts machen, dann muss man da zum language server connecten indem man so und so macht und fertig, lsp macht den rest.

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 [3]. In 2016, Microsoft started collaborating with Red Hat and Codenvy to standardize the protocol's specification [3]. 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. [3] A big advantage of this is that the IDE specific plugin can be kept very simple.



Figure 4: Communication Benefit of LSP

The relevant information is delivered by the language server, which is IDE and language independent. Figure 4 from the VSCode extension guide illustrates these benefits. [6]

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/didChange` 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. [7]

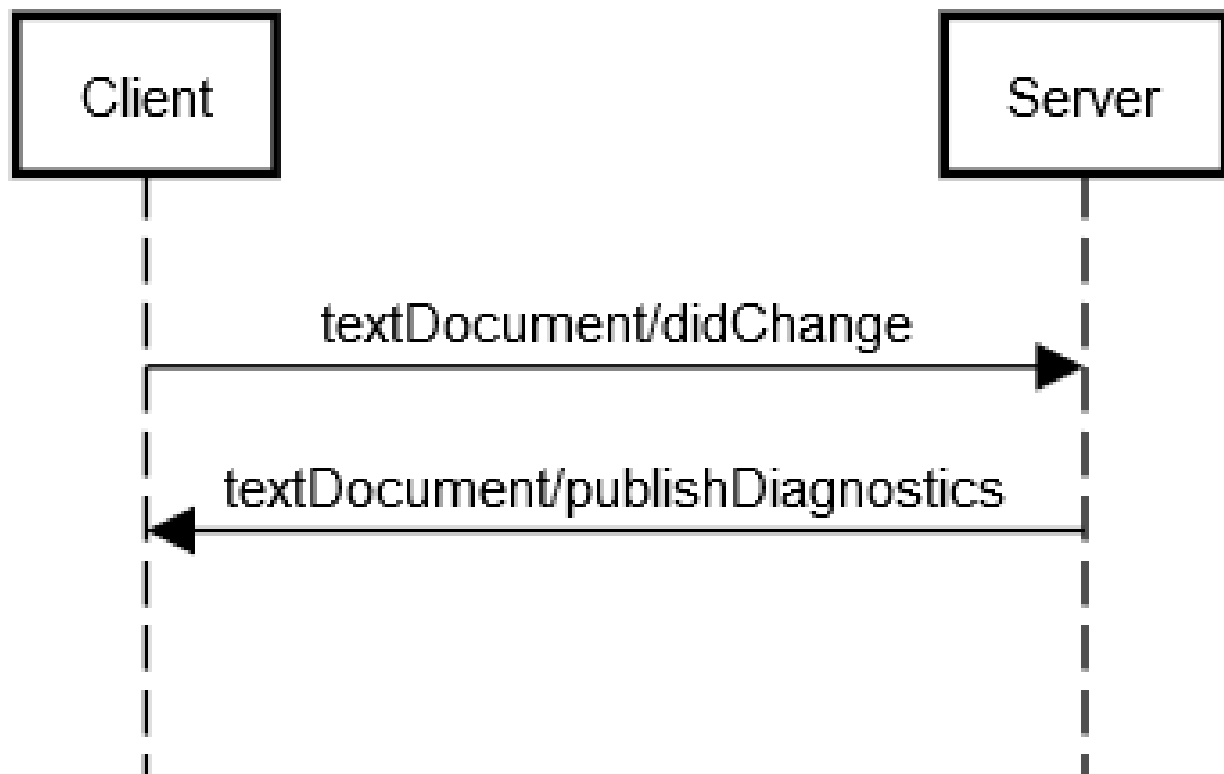


Figure 5: Example Communication

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:/[...]/faill.dfy",
7    "diagnostics":[
8      {
9        "range":{
10         "start":{
11           "line":4,
12           "character":12
13         },
14         "end":{
15           "line":4,
16           "character":17
17         }
18       },
19       "severity":1,
20       "code":0,
21       "source":"file:///D/[dots]/faill.dfy",
22       "message":"assertion violation"
23     }
24   ]
25 }
26 }
```

Listing 5: LSP Message Example

4.2 OmniSharp

To work with the language server protocol, a proper LSP implementation was required. OmniSharp offers support for C#[8]. 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 6: LSP Handler Implementation

Listing 6 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 [9]. 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 [10]. 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 7: LSP Handler Implementation

4.3 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 [11]. This chapter provides the reader with the most relevant concepts with regard to the symbol table.

4.3.1 Modules

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 8: 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 9, the assert statement is violated, since the overwritten `addOne` has priority. [12]

```
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 9: 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.

4.3.2 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 10. Further, functions are not compiled and may only be used in specification context. That is, in contracts or assert statements to proof logical correctness. [12].
- 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. [12]

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

Listing 10: 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 11. [11]

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

Listing 11: Lemma

4.3.3 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. [11]

```
trait Base {
    var a: int
}

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

Listing 12: Hiding

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

4.3.4 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 [11]. This means, that within each module, each method name is unique.

4.3.5 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 13: 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 14: Finding Symbol Definition

The code above would basically already resolve the *GoTo Definition* problem.

4.4 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 15: 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 not contain information about the type of symbols. This is where the symbol table comes into play. The symbol table contains that information and is connected to the AST, for example by the use of a dictionary. This way, it could be stored that the name segment 'i' is of type int. The two concepts are strongly coupled.

4.4.1 Requirements for the symbol table

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

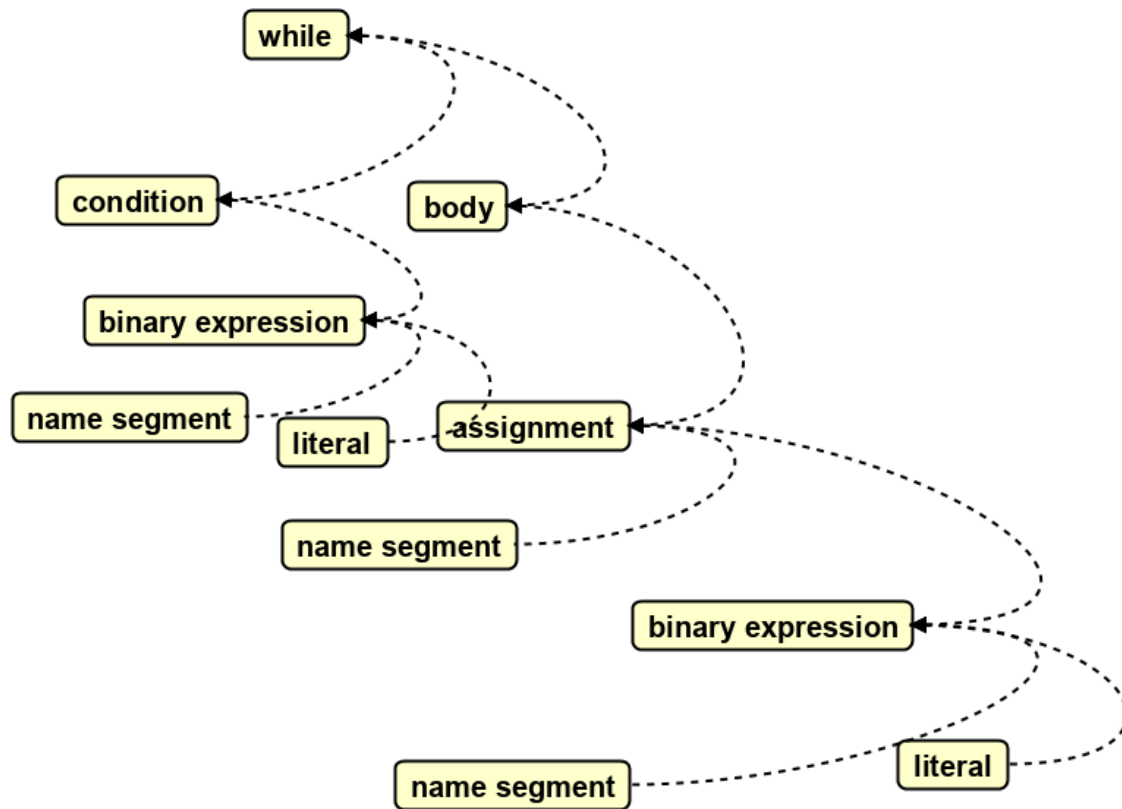
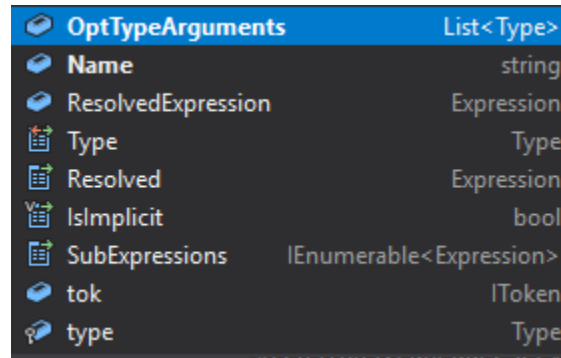


Figure 6: Communication Benefit of LSP

- 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 occurences of a symbol?
- Autocompletion
 - Which declarations are available in a scope?

4.4.2 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. While `ResolvedExpression` looks like an interesting property, it just points to itself in



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

Figure 7: Properties and Fields of a NameSegment Expression

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

4.5 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 16: 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 17: Example for Visitor

4.6 Dafny Expression and Statement Types

Dafny is a very versatile language. While it offers common object oriented language features, it also contains formal language features, comparable to 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`.

4.6.1 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.

4.6.2 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.

4.6.3 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

4.7 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.

4.8 Continuous Integration (CI)

Continuous integration is a very important part for code quality improvement and collaboration. Unfortunately, the CI process in our student research project extended to almost the entire semester [5].

According to our project plan, we wanted to work on open points regarding the CI initially and have completed the theme accordingly for the remaining duration of the bachelor thesis.

4.8.1 Initial Situation

We achieved in our client CI that code was analyzed with SonarQube and the build failed if it contained TypeScript errors [5]. We did not achieve it within reasonable time headless integration test [5].

On the server side we reached the build process as well as the dafny tests and our own unit tests [5]. Automated integration tests and code analysis by SonarQube remained outstanding [5].

4.8.2 Aimed Solution

According to our research, a major problem was that the scanner for sonarqube does not support any other languages besides C# [13]. This means that in addition to C# in a project, TypeScript (for the client) cannot be analyzed simultaneously. Furthermore there are also single Java files in Dafny project. This also led to conflicts in the Sonar analysis in our student research project [5].

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. For the Language Server in C# a special Sonar scanner for MSBuild had to be used, which publishes the analysis in a separate Sonar-Cloud project [2]. Beside the code from our Language server the whole Dafny project code is now analyzed by sonar. This can be very helpful for code reviews.

The only downside is that the code coverage is not analyzed. For .NET OpenCover is a very common tool for code coverage analysis. Unfortunately, it only works on windows and not on our linux CI server [14]. Other tools that work with mono Support .NET Core but not Framework. During our research we came across monocov [15]. This tool would support mono for .NET Framework. Unfortunately this project was archived and has not been supported for almost 10 years [15].

Since we would not gain much added 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.

For an easier testability of the CI, we now also used local docker. This allows us to test CI customizations efficiently. See the developer documentation for more details [2].

The headless integration tests were a bit more tricky. 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.

4.8.3 Docker

As mentioned in the previous chapter, we rely on Docker. The simplicity of Docker Container allows us a comfortable way to integrate the building and testing process into our CI.

Furthermore, the lightweight virtualization is ideally suited to run and debug our Linux CI environment locally and platform-independently (through the Docker Client) in case of problems.

Furthermore we can easily realize the principle "Cattle, not a pet" with docker. Instead of having certain package dependencies that need to be updated continuously (pet), we use a "build, throw away, rebuild" procedure (cattle). So we don't have to worry about security patches and the like. We simply install the latest versions when we create a docker.

Excluded from this are of course specific versions such as Node, Z3, Go, Boogie and Sonar. There the version to be installed is explicitly specified, since version changes there can have essential impairments of the Language Server's function. See the developer documentation for more details [2].

4.8.4 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.

==¿ 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.

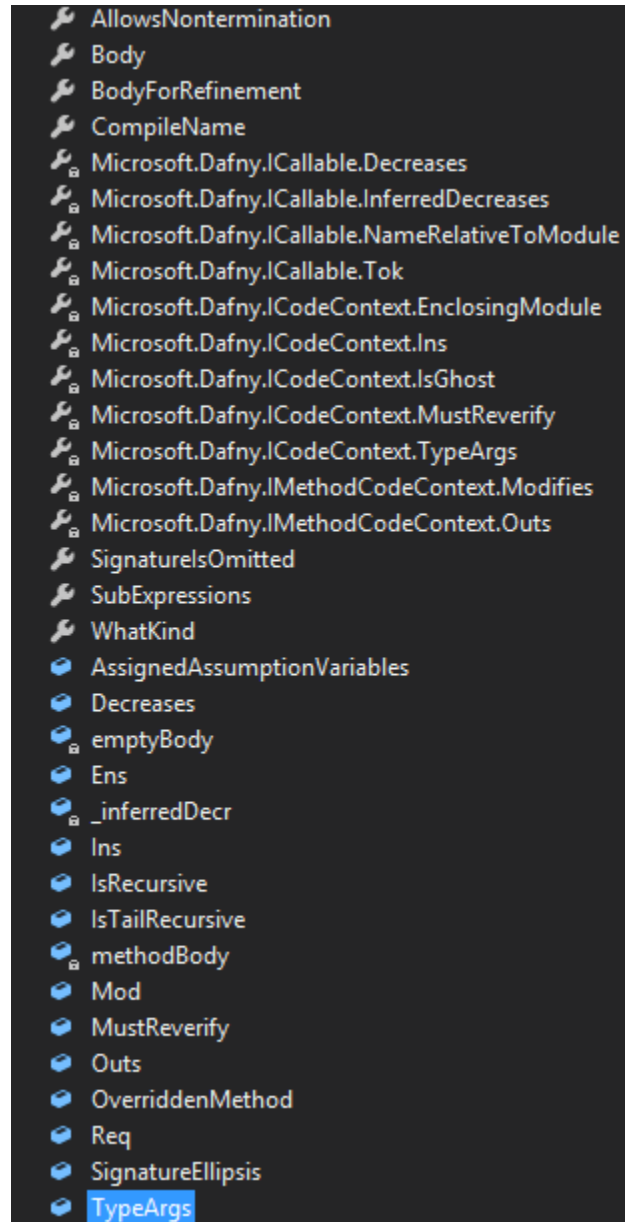


Figure 8: Properties and Fields of a Method

5 Design

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

5.1 Client

The client part of our bachelor thesis includes the VSCode plugin written in TypeScript. It starts the Language Server and establishes a connection via LSP.

Since most of the logic should be kept in the IDE independent Language Server, we have made it our goal to keep the client logic to a minimum. This allows to implement a later plugin for another IDE with as little effort as possible.

In the following we will discuss how we achieved this lightweight, and how and why we decided to split the components in the client part like we did.

5.1.1 Initial Situation

Already in our term project we made revisions on the client. We have connected our new Language Server and greatly reduced the unnecessary logic. The following figure 9 gives an impression of the architecture.

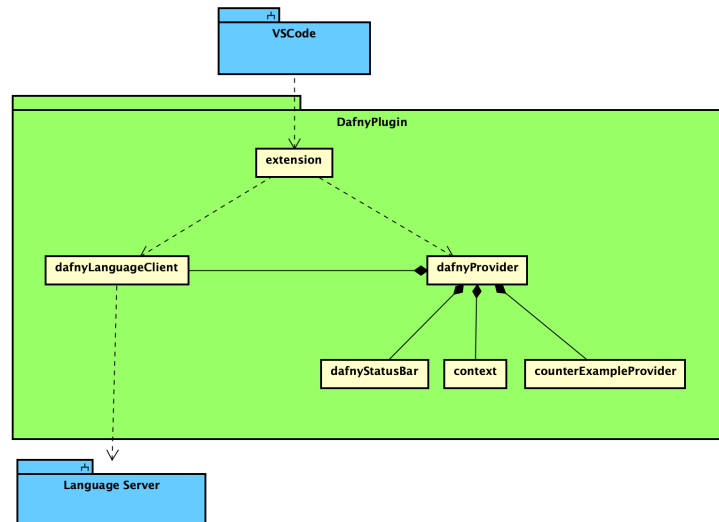


Figure 9: Client Architecture - Term Project

In this simplified representation, the client architecture appears very tidy. Unfortunately, the individual components are very large, almost all members are public and this leads to high coupling and deep cohesion. Furthermore, there are many helper classes that are not grouped into sub-packages. This makes it difficult for other programmers to get into the project. Furthermore, it was difficult to eliminate all dead code due to the non-transparent dependencies.

Because of these problems we decided to redesign the client itself from scratch.

5.1.2 New Architecture

To achieve the goal of a more manageable architecture and to reduce coupling, we have implemented the following measures. As a first step, we aimed to divide all areas of responsibility into separate components. We grouped the components into packages as you can see in figure 10. These packages are discussed in the following sections.

Additionally we detached all logic 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 get started quickly. This is also helpful because they are displayed as hover tooltips.

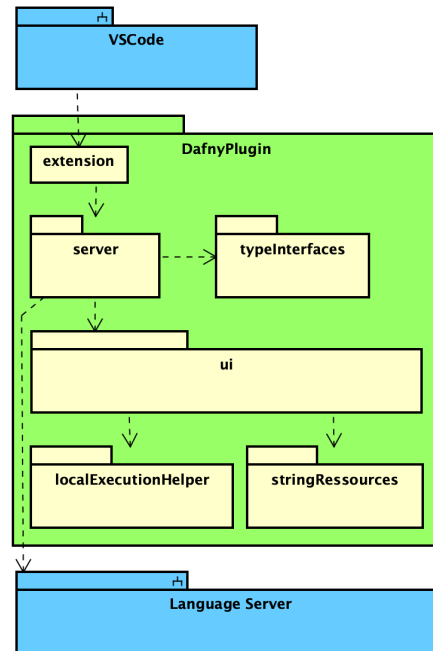


Figure 10: Client Architecture - New Packages

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 basic triggering of the Language Server and the connection setup. In addition, all server requests, which extend the LSP by own functions, are sent to the server via this package.

TypeInterfaces – In our new architecture we do not use the "any" type. We decide all types, in particular the types created specifically for additional functions such as results for counter examples.

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 text strings and command definitions are defined in this package. This package is used by the UI package.

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

5.1.3 Components

In the following figure 11, the components within the packages are also shown for a more detailed view. The contents of type interfaces and string resources have been omitted for clarity.

It can be seen that only compile and counterexample exist as server accesses. All other features were implemented purely through the LSP protocol without additional client logic as support. This server-side implementation via LSP is a great enrichment. For future plugin developments for other IDEs the effort is automatically eliminated.

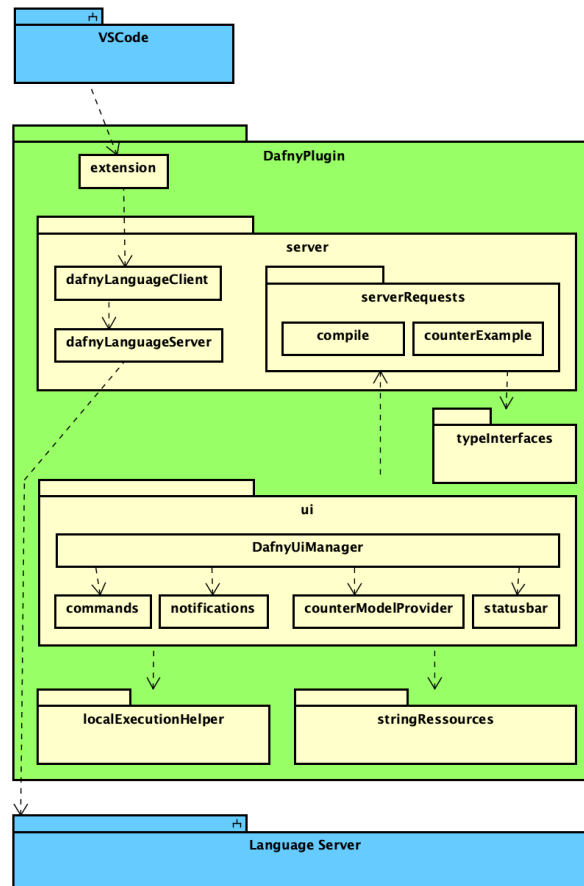
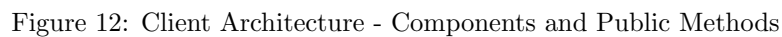


Figure 11: Client Architecture - Components

Figure 12 shows the public methods for the components. All instance variables were set to private. Constructors were not included for simplicity. The contents of type interfaces and string resources were also omitted for clarity



5.1.4 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 a maximum of 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 allows the Language Server to send certain messages which are displayed directly to the user as a popup. These include the types information, warning and error. The corresponding logic on the client side includes the registration of the interception on the server and the corresponding method calls of the VSCode API to display the messages.

Commands – To enable the user to actively use features (such as compile), the corresponding method calls must be linked to the UI. We have three primary links for this: Supplementing the context menu (right-click), shortcuts and entering commands via the VSCode command line.

Statusbar – The information content for the Statusbar is delivered completely by the Language Server. The client only takes care of the user friendly display with icons and help texts. 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 if 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 previous Dafny file.

5.1.5 Types in TypeScript

As already mentioned in the previous chapters, Any types were largely supplemented by the dedicated type. The specified data types for variables in Typescript define which value types can be assigned to the variable.

This prevents type changes of variables as known in pure Java Script. Typed code is accordingly less error-prone - especially for unconscious typecastings.

There are individual Built-In Data Types like Number, Boolean and String [16]. For this purpose we have defined separate interfaces for each of our own types. With an interface we make a promise for a data type what kind of content will be contained. For each separate type of non-standard LSP features (such as Compile and Counter Example), server request arguments and server responses as results are defined as interfaces [17].

hier vielleicht nich beispiel screenshots oder codes rein? lol 2do

5.2 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.

5.2.1 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 `FileNotFoundException`s can be excluded.

5.2.2 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.

```
private static List<string> GenericToStringList<T>(this IEnumerable<T> source, Func<T, string> converter)
{
    return source?.Select(converter).ToList();
}
```

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. Note 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
        .Key} = {kvp.Value}; ");
14     return result;
15 }
```

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> expection = 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].

5.2.3 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 requests has different parameters (such as compilation arguments), than a goto-definition request (which as 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 itself are very simple. For example, if we want to test if the compiler reports a missing semicolon, we could create a testclass `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 of two lines of code. The first handling in the test file, the second one definind our expectations. By the way, the boolean values represent if there were errors and if an executable was generated.

The same applies for test about counter examples, goto definition 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.

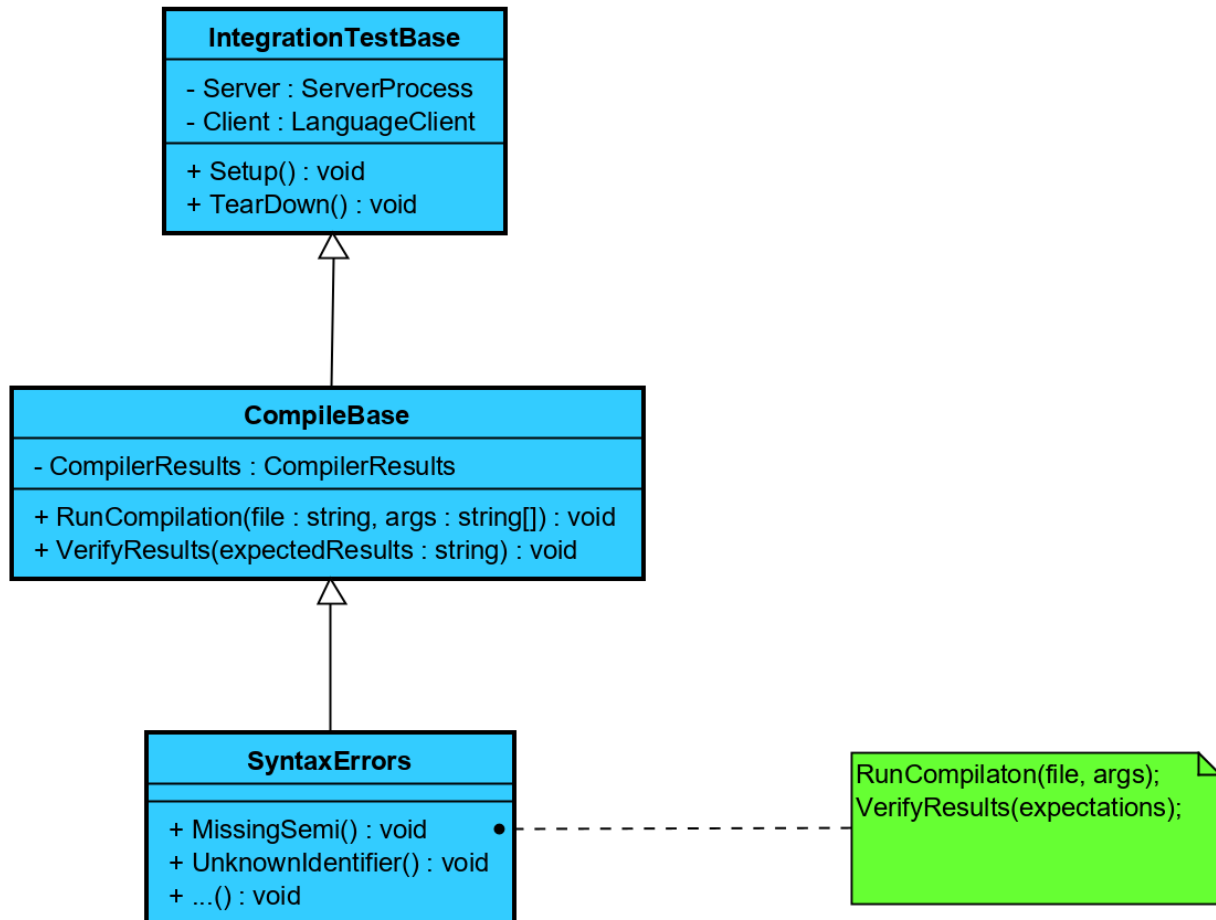


Figure 13: Test Architecture on the Basis of Compilation

6 Implementation

6.1 Symbol Table

6.1.1 Feature Support

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.

6.1.2 Code Review

Fabians Feedback aus der SA... neues Review. "Tu Gutes und sprich davon".

6.2 Client Code Review

Fedback Fabian und Thomas. Interface für Koopelung, weniger Kommentar, mehr Interfaces. Besseres Naming für Variablen. Mehr Interfaces. Beschreiben wie es nun neu aussehen wird.

6.3 Runtime Analysis of the Essential Server Components

6.3.1 Generation of the Symbol Table

Ich glaub beim Aufbau verwendest du nur einmal den Navigator oder? Also $n \cdot n$ Laufzeit?

6.3.2 Use of the Symbol Table

To use the constructed symbol table, we offer a separate navigation component. This navigator has basically two visit procedures. Once from a symbol in the symbol tree up. And once from the root node of the tree down.

The TopDown approach searches down from the entry point and automatically limits the search areas.

For example: TopDown is used especially when searching for symbols at certain positions. If the iteration encounters a symbol that does not cover the range of the search position, the child symbols are not even visited. This avoids a runtime of $O(n)$ for each search. In a dafny file, which was structured very quickly - for example only functions in the highest level - the worst case of $O(n)$ is still reached.

2do Bild, Beispiel, Visualisierung einbauen für die Laufzeitanalyse.

To enable efficient access to the entry points, we have opted for a key-value data structure. The key is the symbol name corresponding to each symbol. This hash structure enables us to access module symbols as entry points with a runtime of $O(1)$. This was especially necessary because certain default modules are automatically created by Dafny if the programmer does not define his own modules. 2do hier noch etwas genauer drauf eingehen... visualisieren... hash besser begründen.

The second search approach is BottomUp. The entry point is a symbol. Starting from this symbol, the system searches for symbols within the reachable scope of the reference symbol.

This approach is used for auto completion, for example. Starting from a passed symbol, all symbols are searched in the corresponding available scope.

6.3.3 Corresponding Duration of the Individual Features

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 $O(1)$).

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.

m m m m m m

6.4 Usability Test

6.5 Mono Support for macOS and Linux

Eines der Kernzeile war es, Support für 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 für den Support auf Linux und macOS brauchen. (warum in der SA; pflicht wegen dafny core. was ist mono) Leider funktioniert nicht. Ansätze die wir probiert haben. verschiedene mono versionen, angefragt im slack. antwort erhalten? github issues: allgemein probleme mit linux/mac weil primär auf windows und gar nicht auf mac getestet wird. (heikle aussage selbs tmit quelle)

[5] [20] [21]

7 Result

In this chapter we describe the achieved results of our work. On the one hand, this concerns the features offered by the plugin (and accordingly by the implemented Language Server), but on the other hand also the architectural improvements to achieve further development of the project for other developers.

7.1 Features for the Plugin User

7.1.1 Compile

7.1.2 Counter Example

7.1.3 Code Verification

7.1.4 CodeLens

7.1.5 Automatic Completion

7.1.6 Hover Information

7.1.7 Rename

7.2 Achieved Improvements for Further Development

8 Conclusion

9 Project Management

10 Designation of chapters taken from the preexisting term project

The following chapters originated in the preexisting semester thesis "Dafny Server Redesign"[5] 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 2
 - Paragraph 1 about Dafny
 - Paragraph 2 about the language server protocol communication
- Chapter 3
 - Chapter 3.1 except for the paragraph about lemmas
 - Chapter 3.2, partially
- Chapter 4
 - Chapter 4.1
 - Chapter 4.2, reworked, better example

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] Marcel Hess and Thomas Kistler. *Developer Documentation*. HSR Hochschule für Technik Rapperswil, 2020.
- [3] *Dafny, Wikipedia*. URL: <https://en.wikipedia.org/wiki/Dafny>. (Accessed: 14.05.2020).
- [4] *HSR Correctness Lab*. URL: <https://www.correctness-lab.ch/>. (Accessed: 14.12.2019).
- [5] Marcel Hess and Thomas Kistler. *Dafny Language Server Redesign*. HSR Hochschule für Technik Rapperswil, 2019/20.
- [6] *Language Server Extension Guide*. URL: <https://code.visualstudio.com/api/language-extensions/language-server-extension-guide>. (Accessed: 15.12.2019).
- [7] *Language Server Extension Guide*. URL: <https://microsoft.github.io/language-server-protocol/specifications/specification-3-14/>. (Accessed: 28.10.2019).
- [8] *Langserver.org*. URL: <https://langserver.org/>. (Accessed: 05.12.2019).
- [9] *OmniSharp/csharp-language-server-protocol*. URL: <https://github.com/OmniSharp/csharp-language-server-protocol>. (Accessed: 05.12.2019).
- [10] Martin Björkström - *Creating a language server using .NET*. URL: <https://app.slack.com/client/T0RE90CRF/C804W8JHE>. (Accessed: 05.12.2019).
- [11] 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.
- [12] *Functions vs. Methods*. URL: <https://www.engr.mun.ca/~theo/Courses/AlgCoCo/6892-downloads/dafny-notes-010.pdf>. (Accessed: 15.04.2020).
- [13] *SonarCloud for C# Framework Project*. URL: <https://community.sonarsource.com/t/sonarcloud-for-c-framework-project/17132>. (Accessed: 23.03.2020).
- [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] *Data Types in TypeScript*. URL: <https://www.geeksforgeeks.org/data-types-in-typescript/>. (Accessed: 15.04.2020).
- [17] *TypeScript: Class vs Interface*. URL: <https://medium.com/front-end-weekly/typescript-class-vs-interface-99c0a1c2136>. (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] *OmniSharp Slack - Mono*. URL: <https://omnisharp.slack.com/archives/C804W8JHE/p1587578976071500>. (Accessed: 24.04.2020).
- [21] *OmniSharp GitHub Issue - Mono*. URL: <https://github.com/OmniSharp/csharp-language-server-protocol/issues/179>. (Accessed: 24.04.2020).

Anhang (Entwickler Doku)