HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

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 Zgraggen (Google)
Counter reader:	Prof. Dr. Olaf Zimmermann

1 Abstract

(Abschnitt 1-2 teilweise übernommen)

Dafny is a formal programming language to proof 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 defitnion, rename, code lens and auto completion were very simple to implement.

Aside features based on the simbol table, preexisting functionality was revisisted 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

2 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.

2.1 Dafny

copy pasta oben, das mit lemma is neu
Dafny is a compiled language that targets C#which can prove formal correctness.[1] Dafny bases on the language "Boogie", which uses the Z3 automated theorem prover for discharging proof obligations.[1] 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 1: 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.

*Abschjnitt 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 devide 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 compelted 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 2: Lemmas

2.2 Initial Solution

In a previous bachelor thesis by Markus Schaden and Rafael Krucker, a plugin for Visual Studio Code was created to support Dafny.[2] The plugin was particularly appreciated by the "HSR Correctness Lab"[3] 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[4], 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.

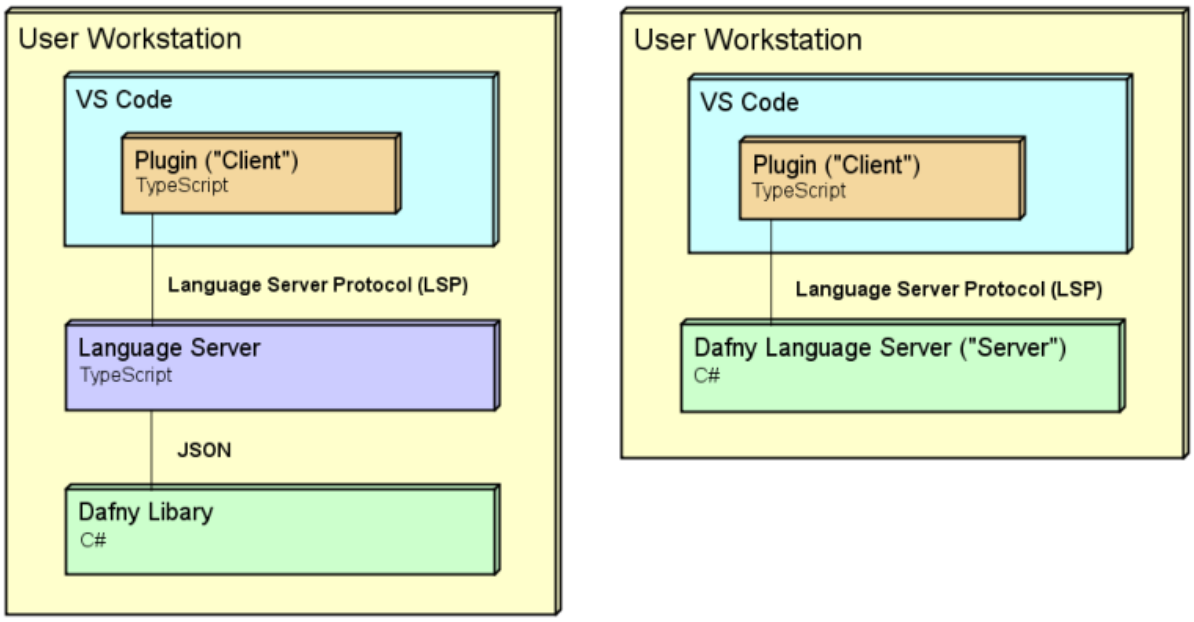

Figure 1: Architecture before (left) and after (right) the preceding term project

alles neu ab hier

2.3 Feature Set

An integrated development environment (IDE) can offer numerous featuers - 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 [4].

2.4 Goals

This bachelor thesis includes two major objectives. In the first segment of the project, the preexisting features taht 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 occurences of a symbol must be stored
- to display code lens, all usages of a declaration must be noted

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

2.5 Results

Without exception, all of the preexisting features could be improved.
The dafny verification process follows now a clear strucutre. 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 it's child nodes during visitation. Aside child-parent relationships, symbol usages are counted too, as well as declarations are resolved. Thus, every occuring 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 declartion, 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 defintion 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 acheived:

- 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 whetever 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.

2.6 Outlook

While the quality of the features, as well as the general code quality could be massively improved, the funcationality 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, 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 market place. 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." habs ams chluss kurz aber nja, evtl noch etwas deutlich iwo. 2do: allfeatures in one bild? hat da snoch platz überhaupt xD*

3 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 detailled 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.

3.1 Language Server Protocol

The language server protocol (LSP) is a JSON-RPC based protocol to communicate between an IDE and a language server [1]. In 2016, Microsoft started collaborating with Red Hat and Codenvy to standardize the protocol's specification [1]. 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. [1] A big advantage of this is that the IDE specific plugin can be kept very simple.

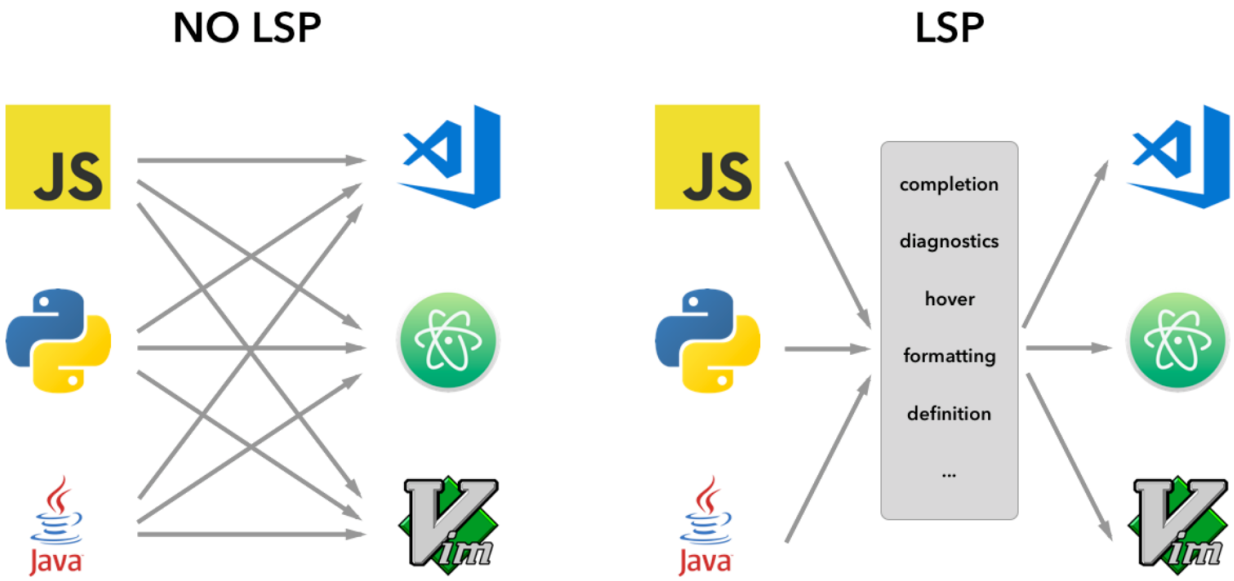

Figure 2: Communication Benefit of LSP

The relevant information is delivered by the language server, which is IDE and language independent. Figure 2 from the VSCode extension guide illustrates these benefits. [5]

3.1.1 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.

3.1.2 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]

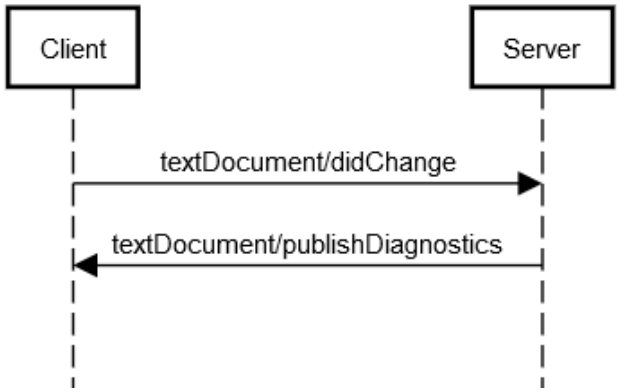

Figure 3: Example Communication

3.1.3 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":{
```

```
                    "line":4,
                    "character":17
                }
            },
            "severity":1,
            "code":0,
            "source":"file:///D[\dots]/fail1.dfy",
            "message":"assertion violation"
        }
    ]
  }
}
```

Listing 3: LSP Message Example

3.2 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.

3.2.1 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.

```
public class AutoCompletion : ICompletionHandler
{
    public Task<CompletionList> Handle(CompletionParams request,
        CancellationToken cancellationToken)
    {
        throw new NotImplementedException();
    }
}
```

Listing 4: LSP Handler Implementation

Listing 4 illustrates that the user simply has to implement an interface provided by Omnisharp. Within the `request` parameter, all required information is passed to the handler. For 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 langauge 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.

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

Listing 5: LSP Handler Implementation

3.3 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 [10]. This chapter provides the reader with the most relevant concepts with regard to the symbol table.

3.3.1 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 6: 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 7, 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 7: 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.

3.3.2 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 8. 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 8: 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 9. [10]

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

Listing 9: Lemma

3.3.3 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. [10]

```
trait Base {
    var a: int
}

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

Listing 10: Hiding

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

3.3.4 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 [10]. This means, that whitin each module, each method name is unique.

3.3.5 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 nasted 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 11: 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 implemnentiaotn 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)
```

```
{
    foreach (Symbol s in scope.AllSymbols)
    {
        if (s.Name == target.Name && s.IsDeclaration)
        {
            return s;
        }
    }
    if (scope.Parent != null)
    {
        return FindDeclaration(target, scope.Parent);
    }
}
```

Listing 12: Finding Symbol Definition

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

3.4 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.

```
while(i<5) {
    i = i + 1;
}
```

Listing 13: 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.

3.4.1 Requirements for the symbol table

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

- Cursor Position
  - Which name segment is at the cursor's position?
- Goto Definition

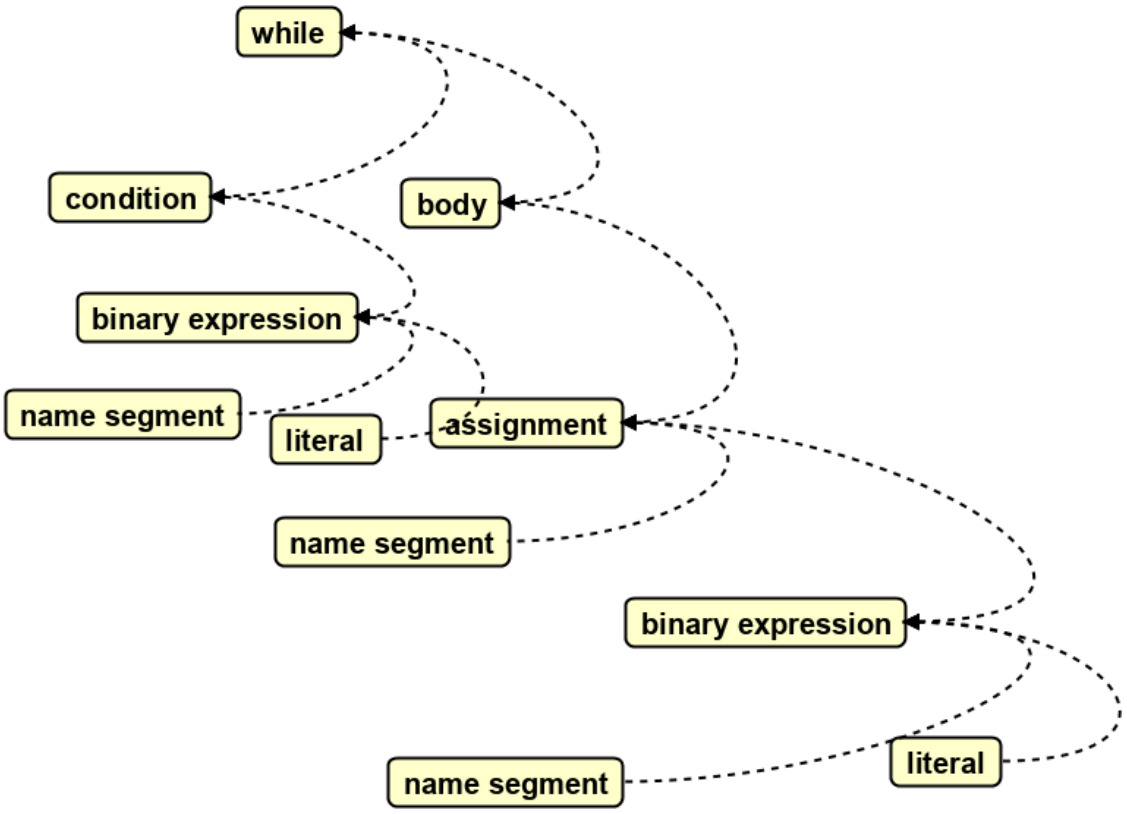

Figure 4: Communication Benefit of LSP

  - 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?

3.4.2 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 occurence 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 a regular case, not to the declaration or such. Thus, if a name segment is encountered, for example as the

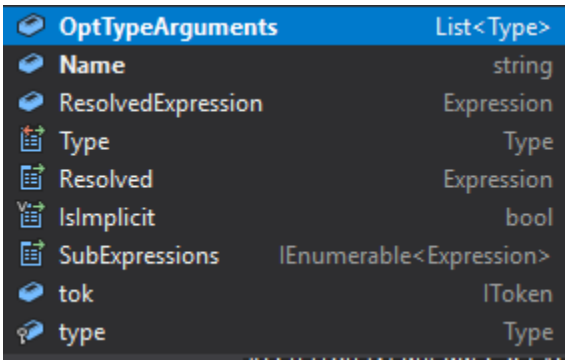

Figure 5: Properties and Fields of a NameSegment Expression

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 it's 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.

3.5 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 accpeptor 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 14: 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 15: Example for Visitor

3.6 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 clases 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`.

3.6.1 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-identifer.
- 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.

3.6.2 Statements

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

3.6.3 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

3.7 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 mutch 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, ressources 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.

3.8 Continuous Integration (CI)

Continuous integration is a very important part for code quality and colaboration. Unfortunately, setting up the CI process in the preceding project[4] 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.

3.8.1 Initial Situation

On the client side, code was analyzed with SonarQube. If the code contained TypeScript errors, the build failed [4]. The end-to-end tests, provided by the preceding bachelor thesis[2] 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 exectued. Integration testing and code analysis by SonarQube remained pending [4].

3.8.2 Targeted 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 [4].

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 SonarCloud 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.

3.8.3 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 priciple "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 fürü 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 werlche version oder?

3.8.4 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. 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.

3.9 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.

3.10 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.

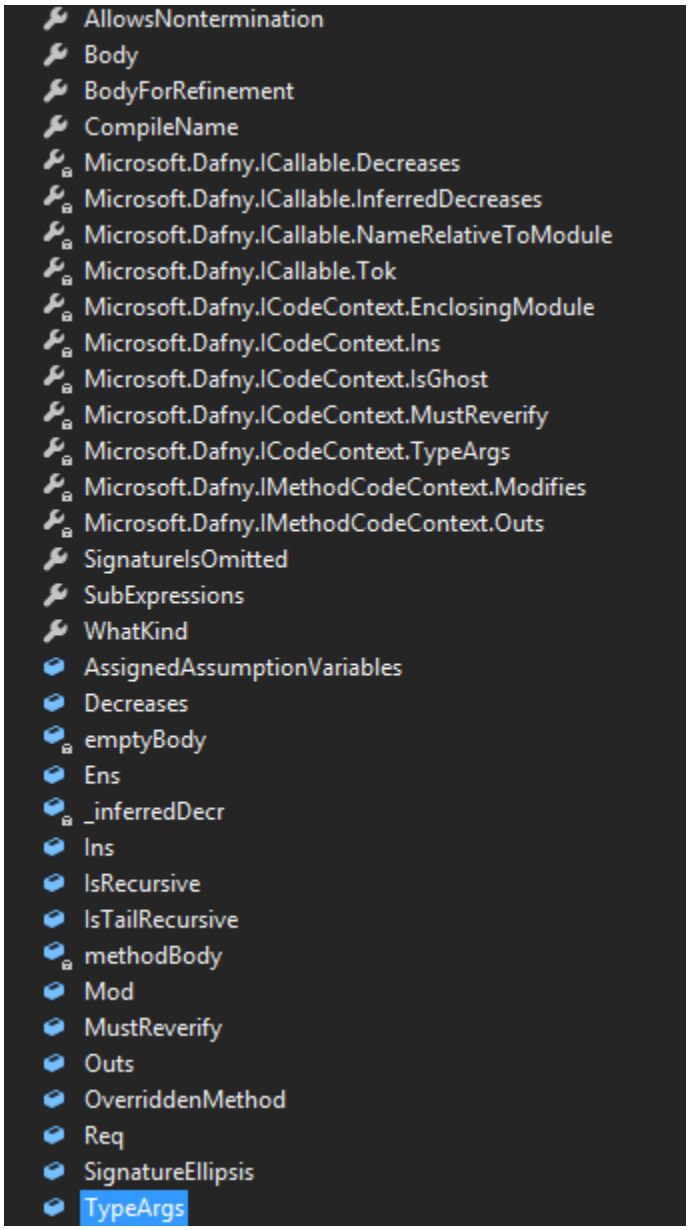

Figure 6: Properties and Fields of a Method

4 Design

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

4.1 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.

4.2 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.

4.2.1 Initial Situation

The client, taken over from the preexisting bachelor thesis by M. Schaden and R. Krucher [2], 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 7 gives an impression of the architecture at the beginning of this bachelor thesis..

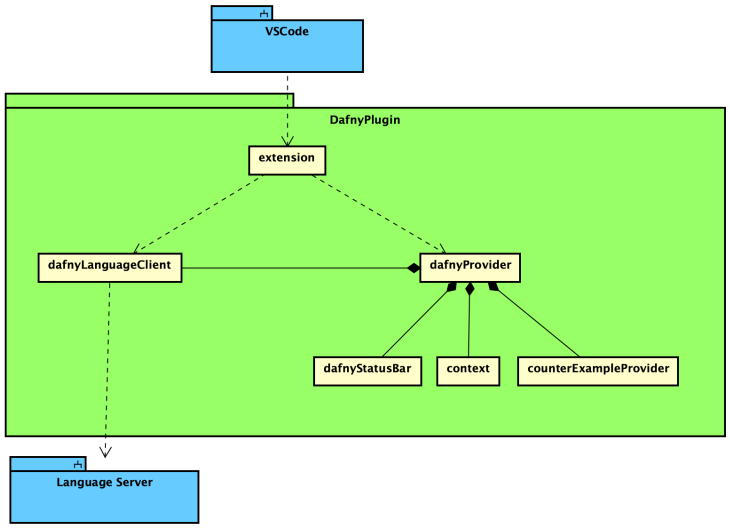

Figure 7: Client Architecture - Term Project

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.

4.2.2 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 **??**. These packages are discussed in the following sections.

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 basic triggering *was ist 'triggering a server?' starten? msg senden? aktivieren?* 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 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.

StringRessources – 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.

4.2.3 Components

Figure 8 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.

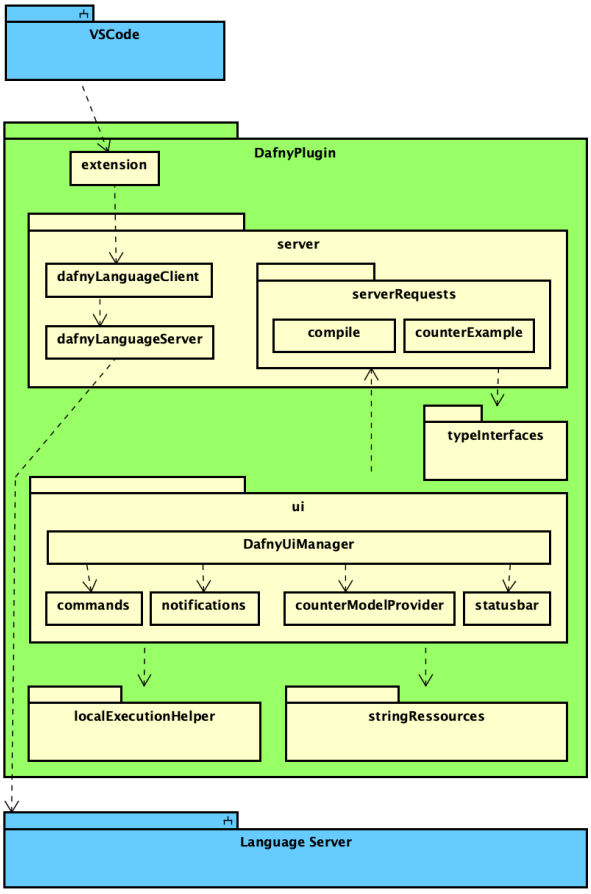

Figure 8: Client Architecture - Components

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 respone*, 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.

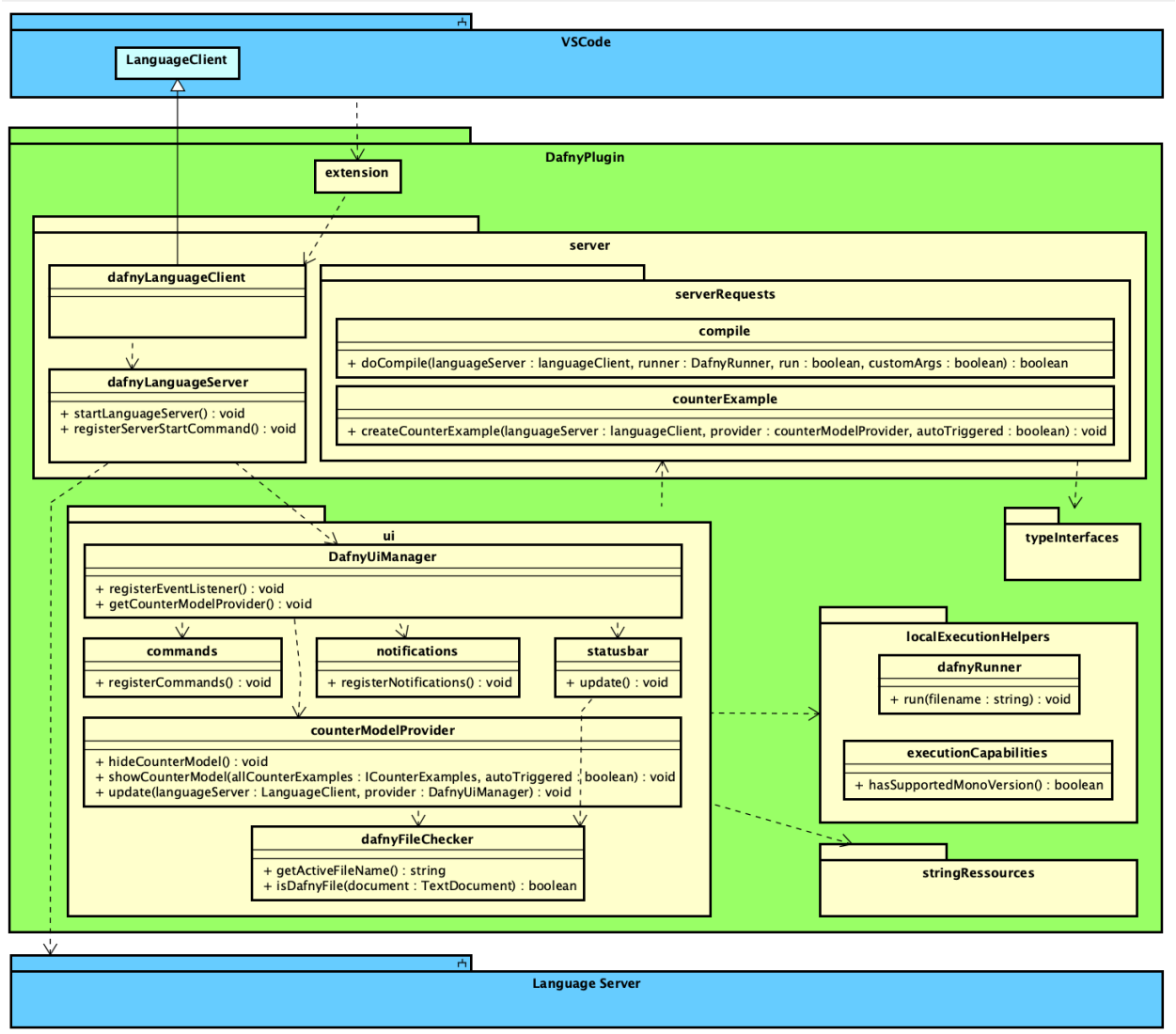

Figure 9: Client Architecture - Components and Public Methods

Figure 9 shows the public methods of each of the components. Only these are accessbile, 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.

4.2.4 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.

4.2.5 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?
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' nict sicher ob das aber schlimm ist oder nur mir auffällt, ein intro zu haben ist ja legitim.

4.3 Server

vlt noch ne übersicht iwo, wie figure 9 SA - hab grad keine übersciht 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 23 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 16: 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 10 shows the basic, fundamental layout of the server implementation. Be aware that this illustration is highly simplified for a better understanding.

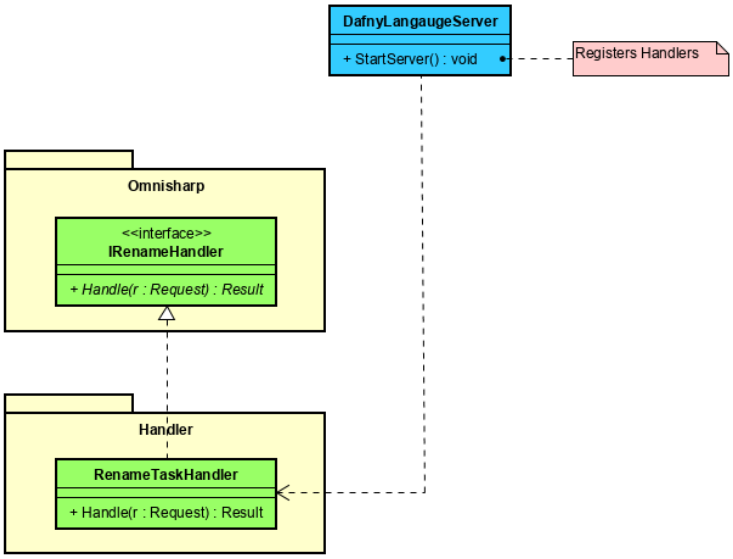

Figure 10: Basic Server Concept

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.

4.3.1 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.

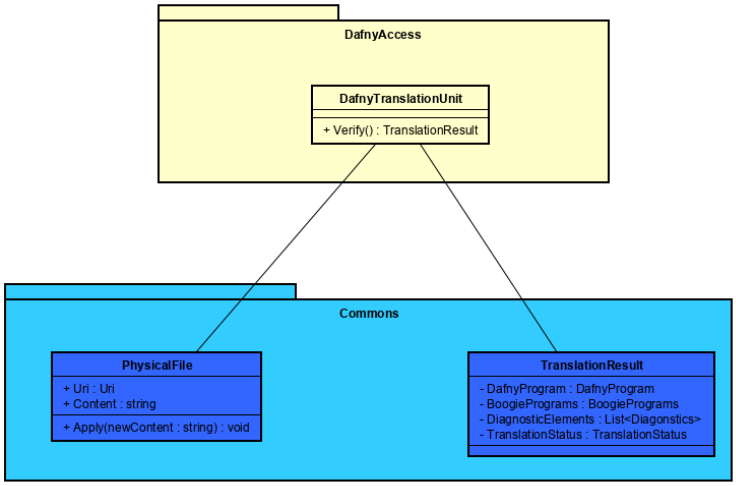

Figure 11: Dafny Translation Unit

Excluded from the figure are some converter methods, that convert `Diagnstoics`, such as warnings and errors, to a einheitlich 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

4.3.2 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*

internen noch auflsöen 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 identificator (an URI) to a `FileRepository`. It offers a method to update a file, but also to get information about a file.

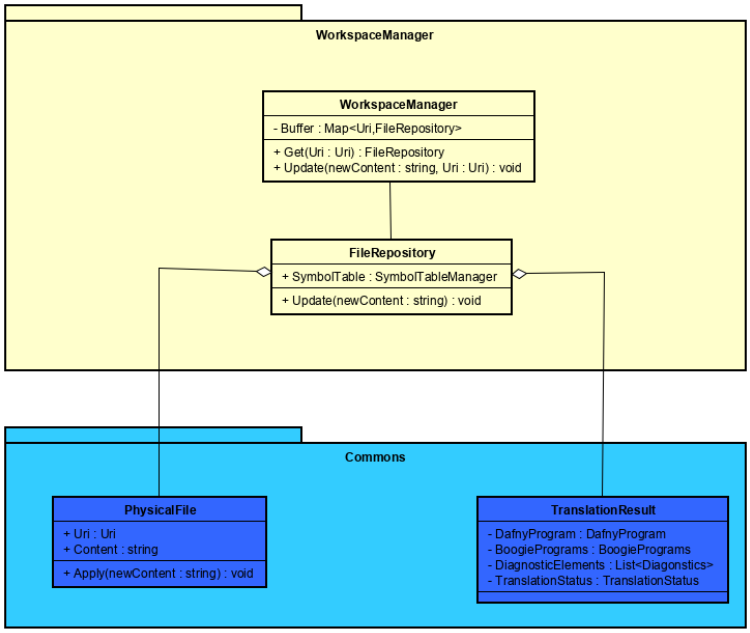

Figure 12: Workspace Component

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 12. 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.

4.3.3 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 it's 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 naviagte 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

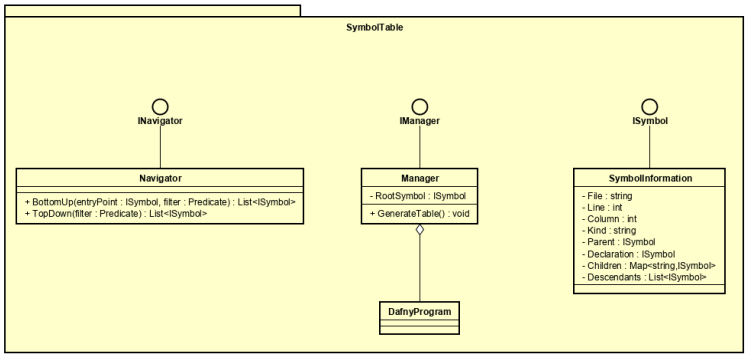

Figure 13: SymbolTable Component

4.3.4 Core

The core package contains the actual logic. There is one class for each feature, for example `RenameProvider` or `DiagnsoticsProvider`. 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.

4.3.5 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.

4.3.6 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.

4.3.7 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.

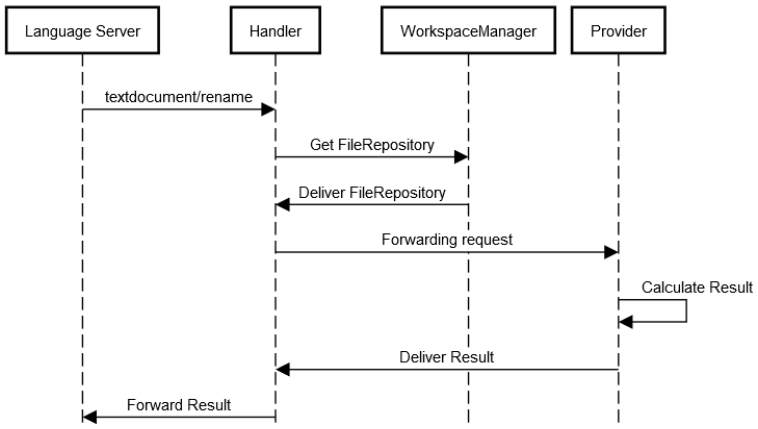

Figure 14: Overview

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

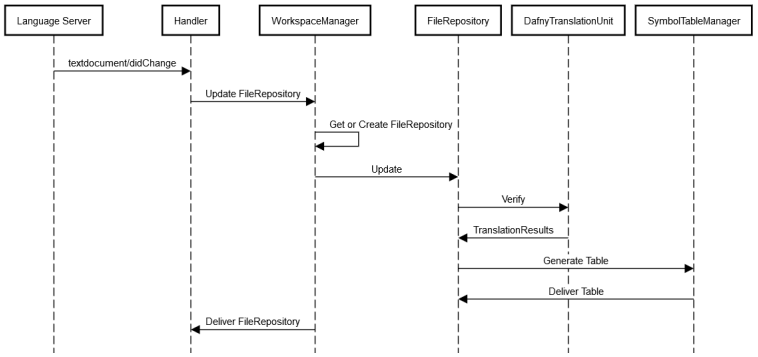

Figure 15: Updating a File

Plan Kapitelablauf:

1. Aside the basic layout, we also needed to store some inforamtion for files. Also -¿ Workspace Manager
2. Dann Workspafce MAanager 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 überigens mit msg sender auch tun könnte, das wär noch fancy aber wieder aufwand.)
4. Dann brauchen iwr Dafny Access -¿ Translation Ding Zeugs da sagen. (evtl sgoar tzuerst?)
5. Die eigentliche Core Logic wurde dann in die HAndler Services ausgelart (NAME BESSER!!). also handler -¿ handlerSErvice, welcher dann die iegentliche logic durchbuttert. service somit isoliert testbar mimi interface blabla. itnerace müsste man noch machen hust hsut. ok. dfas lohnt sich wohl aber dnek ich. (notenmässig)

6. resx als krimskrams
7. config und msg sender und wordsprovider als 'library' layer. den besser auch noch renamen.

4.4 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.

4.4.1 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 17: Test File Reference

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

4.4.2 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 a 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 18: 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
           .Key} = {kvp.Value}; ");
14     return result;
15 }
```

Listing 19: 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> expecation = new List<string>()
2  {
3      "L3 C19: in1 = 2446; ",
4      "L9 C19: in2 = 891; "
5  };
```

Listing 20: Expectation

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

4.4.3 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 requests has differnt parameters (such as compilation arguments), than a goto-defintion 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 `RunCompilaton(string file, string[] args)` and `VerifyResults(string expectation)`. One can now easily see the dedicated paramter 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 hour case-specific base class. Thus, the methods `RunCompilation` and`Verify` are at our dispoal. That means, that hour 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 21: Sample Test for Missing Semicolon

As you can see, the test contains only of two lines of code. The first handing 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.

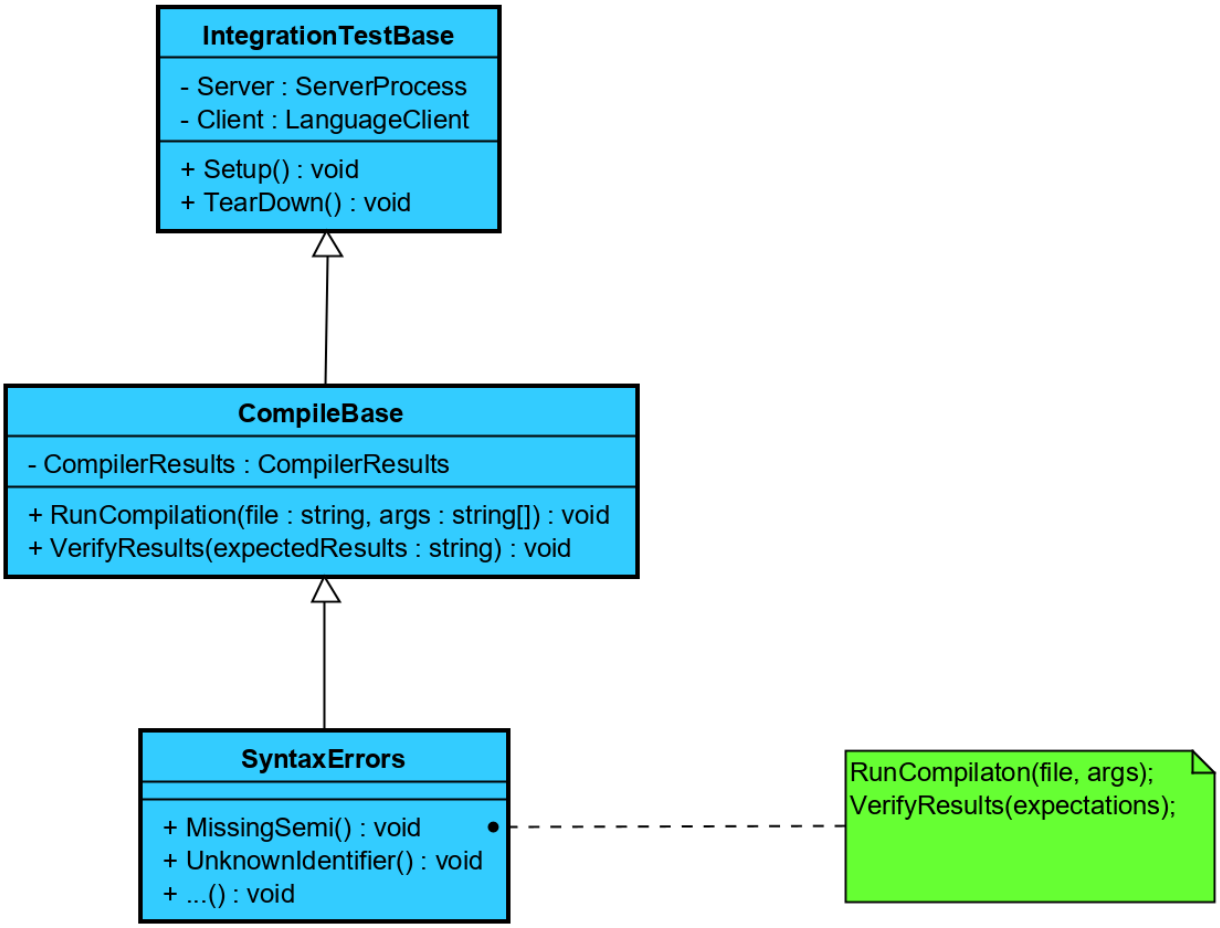

Figure 16: Test Architecture on the Basis of Compilation

5 Implementation

5.1 Client

a

5.2 Server

5.2.1 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 22: Main Function

The launch of the server itself is devided 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 registred. In the third stage, once the server is running, a message sending service is instanciated to notify the client about the successful server start and, if any, errors occured during startup. Lastly, the console output stream is redirected to keep the langauge 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 23: Starting the Language Server

5.2.2 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. Unkown or illegal values will not be set and errors are added to a error reporter. Finally, the launch arguments are parsed, again overwritting 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 initalizer is implemented exception safe. This means that it will run to completion and at wrost 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. Simliar 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.

5.2.3 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 3.2.2 by defining an own interface.

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

- GetRegistrationOptions: Is called when the handler is registred. 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 cabability, necessary for the `SetCapability` method. This could be,

for example, `RenameCapability` which has the metnioned 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,
       CancellationToken cancellationToken)
2  {
3      _log.LogInformation(string.Format(Resources.LoggingMessages.request_handle,
           _method));
4      try
5      {
6          FileRepository f = _workspaceManager.GetFileRepository(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 24: 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.

5.2.4 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 25: 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 resonibility for applying file updates. *bild hier unbedingt mit workspace -¿ filerepo -¿ physfile/translationresult/symobltablemanager 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 occured?
- 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 26: Handling Compilation

5.2.5 DafnyAccess

Dafny Access it 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 checkd 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 27.
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 TranlsationResult + TranlsationStatus 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 27: Setting up Dafny Options

5.3 Symbol Table

This package provides four components:

- 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 occuring 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 proeprty 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 exmaple, 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 28: 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 porperty 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 definiton, 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 expections 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 attched 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 registred in the symbol table. A second iteration is then started, using the deep visitor, which will ignore delcarations 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)` metod. 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 decideds if bodies like occuring 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 29: 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 speicific 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 gebrabbel)
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

visisted before. When the method is finally visisted, the property 'parent' can just be set with the scope the visitior is in. Since the method itself will have it's 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 30: 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 visistor, 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 ot 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 declaraiton, and matches the name. Challenges occured 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 31: Finding a Declaration

This snippet assigns the navigator to move upwards in scope and serach 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.

Both options are implemented so that they can return a single, first match, or any symbols that match a criterion. To illustrate this, let's 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 hast 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 worthwile. 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 clases 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.

5.3.1 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 `SymbolTaleManger` 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 usabilty, 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 32: 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 it's declaration. The declaration has all usages of the symbol stored, and thus, all occurences 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 and underscore
- Name must not eb a reserved word, such as 'method'
- Name must not contain any other then alphanumerical dinger 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 einzln druch, msus man fast, hat ja vieles interessantes, z.b. so diagnostic conversion (wobie xcdas bei dafny access is)
Completion

Code Lens

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(dafnyProgarm, filePath, otherFiles, true,
      textwriter);
```

Listing 33: Calling the Dafny Compiler

The provider will check, if any errors occured 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 reslts 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 simbols, 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 steh talles 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 verlgeich)
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 it's parent, navigation in both ways can be done within O(1).

5.4 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.

5.4.1 Unit Tests

... *hmm, net sicher was schreiben*

5.4.2 Integration Tests

As described in chapter 4.4, 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 34: 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      Verify(expected);
13  }
```

Listing 35: 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.

5.5 Code Reviews

ich glaube es ist nicht gedacht einzelne code reviews zu dokumentieren. das waere ja mehr dann wie ein arbbeitsrapport, aka woche 12: wir haben das code review bearbeitet und dies und das gemacht. eher so bei kapitel design: wir haben entschieden, strings da und da auszulagnern. dies wurde u.a. wegen dem code review so geamcht. dann implementation: hier wurde das so und so implementiert, weil beim code review es so gesagt wurde

5.5.1 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 17 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.

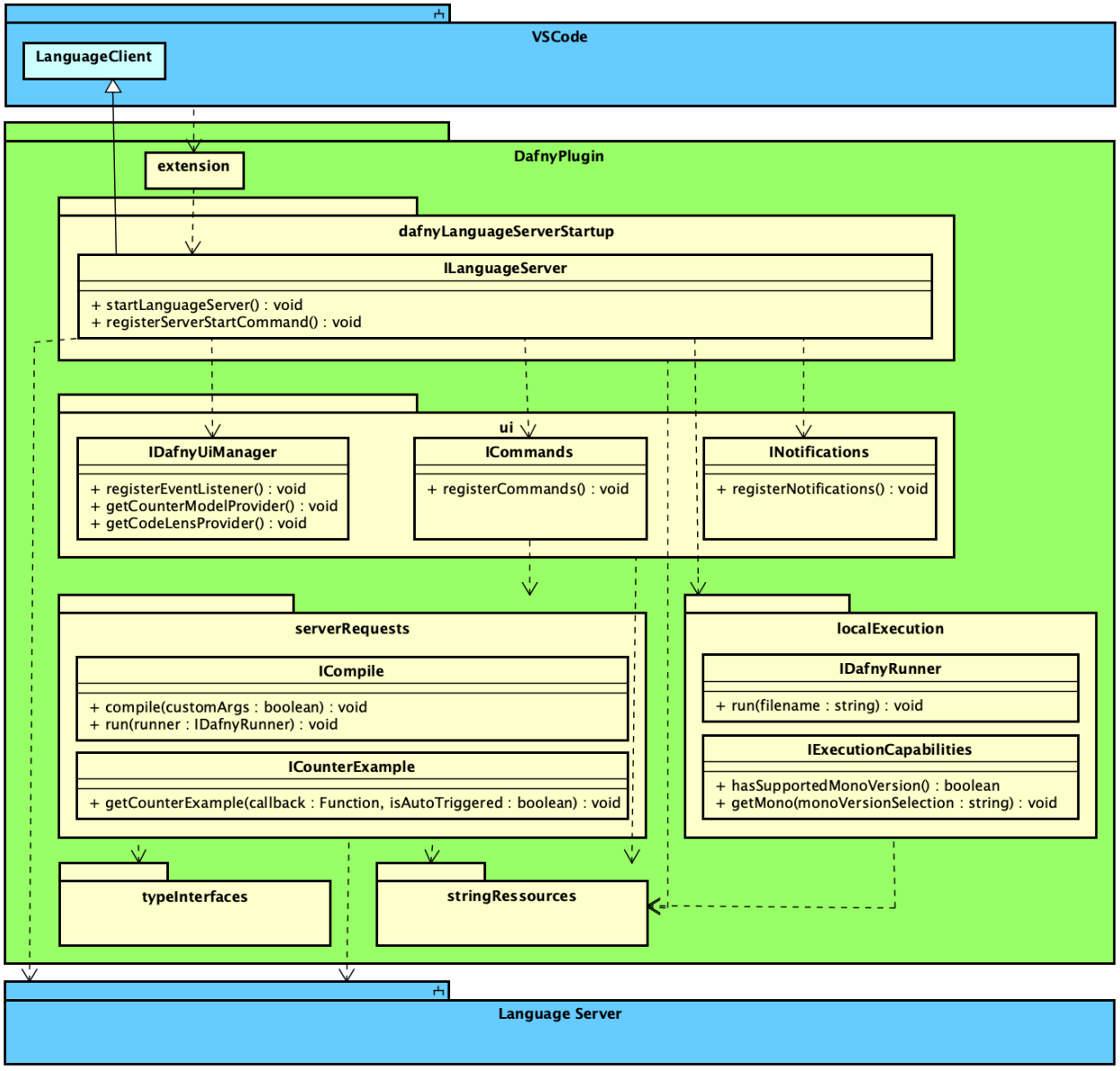

Figure 17: Second Major Client Refactoring

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 `stringRessources`. 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.

m m m m m m

5.6 Usability Test –¿ auch results oder?

5.7 Mono Support for macOS and Linux -Kaptitel nicht hier. entweder anaylse oder Result

Eines der Kernzeiele 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 funktionierts nicht. Anssaetze die wir probiert haben. verschiedene mono versionen, angefragt im slack. antwort erhalten? github issues: allgemein probleme mit lunux/mac weil primaer auf windows und gar nicht auf mac getestet wird. (heikle aussage selbs tmit quelle)
[4] [23] [24]

6 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.

6.1 Features for the Plugin User

6.1.1 Compile

6.1.2 Counter Example

6.1.3 Code Verification

6.1.4 CodeLens

6.1.5 Automatic Completion

6.1.6 Hover Information

6.1.7 Rename

6.2 Achieved Improvements for Further Development

Fabians Feedback aus der SA... neues Review. "Tu Gutes und sprich davon". -¿ vlt auch eher bei results "es wrude gsagt, das muss gemacht werden -¿ haben wir gemacht"

7 Conclusion

8 Project Management

9 Designation of chapters taken from the preexisting term project

The following chapters originated in the preexisting semester thesis "Dafny Server Redesign"[4] 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 sepearately.

- Chapter 1
  - Paragraph 1 about Dafny
  - Paragraph 2 about the language server protocol communication
- Chapter 2
  - Chapter 2.1 except for the paragraph about lemmas
  - Chapter 2.2, partially
- Chapter 3
  - Chapter 3.1
  - Chapter 3.2, 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] *Dafny, Wikipedia*. URL: https://en.wikipedia.org/wiki/Dafny. (Accessed: 14.05.2020).

[2] 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.

[3] *HSR Correctness Lab*. URL: https://www.correctness-lab.ch/. (Accessed: 14.12.2019).

[4] Marcel Hess and Thomas Kistler. *Dafny Language Server Redesign*. HSR Hochschule für Technik Rapperswil, 2019/20.

[5] *Language Server Extension Guide*. URL: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide. (Accessed: 15.12.2019).

[6] *Language Server Extension Guide*. URL: https://microsoft.github.io/language-server-%20protocol/specifications/specification-3-14/. (Accessed: 28.10.2019).

[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-protoco. (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] *OmniSharp Slack - Mono*. URL: https://omnisharp.slack.com/archives/C804W8JHE/p1587578976071500. (Accessed: 24.04.2020).

[24] *OmniSharp GitHub Issue - Mono*. URL: https://github.com/OmniSharp/csharp-language-server-protocol/issues/179. (Accessed: 24.04.2020).

Anhang (Entwickler Doku)