1 Cheat Sheet (TMP)

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

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

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

1.1 Untertitel

1.1.1 Das ist die tiefste Titelebene

Ich bin Text.

Figure 1: My caption

Davor ein Bild.
Mehr dazu in Abbildung 1.

1.2 Quellen

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

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

Beachten sie die Fussnote[1]

[1] Ich bin die Fussnote

1.3 Aufzählung

- Erstens
- Zweitens

1. Erstens
2. Zweitens

Erstens

Zweitens

1.4 Tabelle

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

Figure 2: My table

Das war also Tablle 1.4.

1.5 Code

Fogelnd Code in C#

```
        public void SendInformation(string msg)
        {
            SendMessage("INFO", msg);
        }
```

Listing 1: My Caption

```
        DafnyCode() {}
```

Listing 2: My Caption

Da wär jetzt so code, siehe Listing 2.
The user needs to write `this` in front o fthe variable `myVariable`.

1.6 Referenced Section

You can read more about references in section 1.6

Dafny Language Server
Bachelor Thesis

Department of Computer Science
University of Applied Science Rapperswil

Spring Term 2020

Authors:	Marcel Hess, Thomas Kistler
Advisors:	Thomas Corbat (lecturer), Fabian Hauser (project assistant)
Expert:	Guido Zgraggen (Google)
Counter reader:	Prof. Dr. Olaf Zimmermann

2 Abstract

Table of Contents

3 Introduction

2do: iwo den Satz "Zielgruppe die HSR Studenten" einbauen. "Messbarkeit von Erfolg."

3.1 Problem Domain

3.2 Relevance

- ...
- ...

3.3 Outlook

Das ist ein Absatz [1].

Und das wäre ein zweiter Absatz. [1]

4 Analysis

4.1 Dafny Language Features

With regard to the symbol table, the Dafny language had to be studied more in detail. For example, overloading describes the existence of multiple methods with the same name, but different signatures. This is obviously 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 [3]. This chapter provides the reader with the most relevant concepts in regard to the symbol table. Of cousre, Dafny offers much more language features.

4.1.1 Modules

Dafny code can be organized with 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`.

```
module Helpers {
    function method addOne(n: nat): nat {
        n + 1
    }
}
```

Listing 3: 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 item. For example, in listing 4, the assert statemtn is violated, since the overwritten `addOne` has priority. [4]

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

import opened Helpers
method m3() {
    assert addOne(5) == 6; //violated
}
```

Listing 4: Naming Clash

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

4.1.2 Functions vs. 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 5. Further, functions are not compiled and may only be used in specification context. That is, in contracts or assert statements to proof logical correctness. [4].
- 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. [4]

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

Listing 5: 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 6. [3]

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

Listing 6: Lemma

4.1.3 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. [3]

```
trait Base {
    var a: int
}

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

Listing 7: Hiding

This means that we do not have to consider this issue any further with regard to our symbol table.

4.1.4 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. [3] This means, that whitin each module, each method name is unique.

4.1.5 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.

```
    class A {
        constructor () { a := 2; }
        var a: int
        method m()
        modifies this
        {
            print a;            // 2
            var a: string := "hello";
            print a;            // hello
            print this.a;       // 2
            {
                print a;        // hello
                var a: bool := true;
                print a;        // true
                print this.a;   // 2
            }
        }
    }
```

Listing 8: 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.

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.

```
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 9: Finding Symbol Definition

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

4.2 Symbol Table

Was ist eine Symbol tAble? Inwiefern modifiezeren wir das? Dafny hta keine, drum selber bauen.
Was erwarten wir von der Symbol table? Goto Def -¿ Wo ist die Deklaration? Rename -¿ Get all usages with position. Code Lens -¿ Get all usages

4.3 Continuous Integration (CI)

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

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

4.3.1 Initial Situation

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

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

4.3.2 Aimed Solution

According to our research, a major problem was that the scanner for sonarqube does not support any other languages besides C# [6]. This means that in addition to C# in a project, TypeScript (for the client) cannot be analyzed simultaneously. Furthermore there are also single Java files in Dafny project. This also led to conflicts in the Sonar analysis in our student research project [5].

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

As a result, the client could still be easily analyzed with the previous Sonar scanner. For the Language Server in C# a special Sonar scannerfor MSBuild had to be used, which publishes the analysis in a separate SonarCloud project [2]. Beside the code from our Lanugae server the whole Dafny project code is now analyzed by sonar. This can be very helpful for code reviews.

The only downside is that the code coverage is not analyzed. For .NET OpenCover is a very common tool for code coverage analysis. Unfortunately, it only works on windows and not on our linux CI server [7]. Other tools that works with mono Support .NET Core but not Framework. During our research we came across monocov [8]. This tool would support mono for .NET Framework. Unfortunately this project was archived and has not been supported for almost 10 years [8].
Since we would not gain much added value with sonar code coverage, we decided not to pursue this approach any further. The coverage information is provided by the locally installed IDEs anyway.

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

The headless integration tests were a bit more tricky. In consultation with our supervisor, we have removed these tests from the client project and replaced them with own specially written integration tests on the server side.

4.3.3 Docker

As mentioned in the previous chapter, we rely on Docker. The simplicity of Docker Container allows us a comfortable way to integrate the building and testing process into our CI.
Furthermore, the lightweight virtualization is ideally suited to run and debug our Linux CI environment locally and platform-independently (through the Docker Client) in case of problems.

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

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

4.3.4 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.

5 Design

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

5.1 Client

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

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

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

5.1.1 Initial Situation

Already in our term project we made revisions on the client. We have connected our new Language Server and greatly reduced the unnecessary logic. The following figure 3 gives an impression of the architecture.

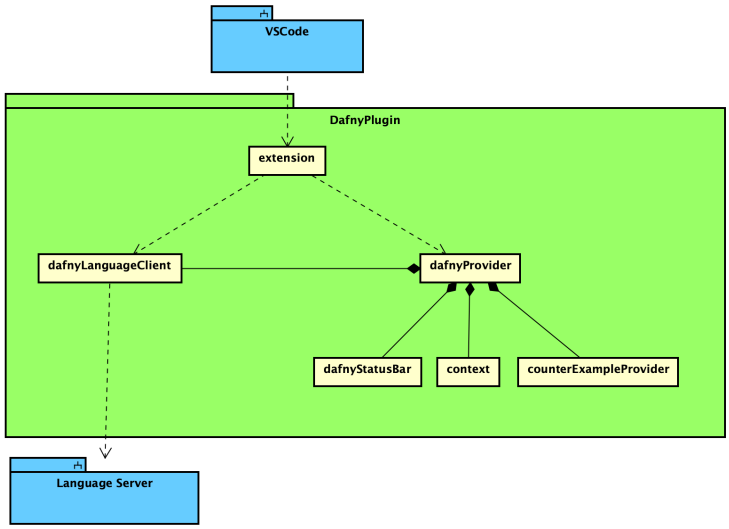

Figure 3: Client Architecture - Term Project

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

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

5.1.2 New Architecture

To achieve the goal of a more manageable architecture and to reduce coupling, we have implemented the following measures. As a first step, we aimed to divide all areas of responsibility into separate components. We grouped the components into packages as you can see in figure 4. These packages are discussed in the following sections.

Additionally we detached all logic from the extension class (the main component). This resulted in the root directory containing only a lightweight program entry point and the rest of the logic was split between the created packages.

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

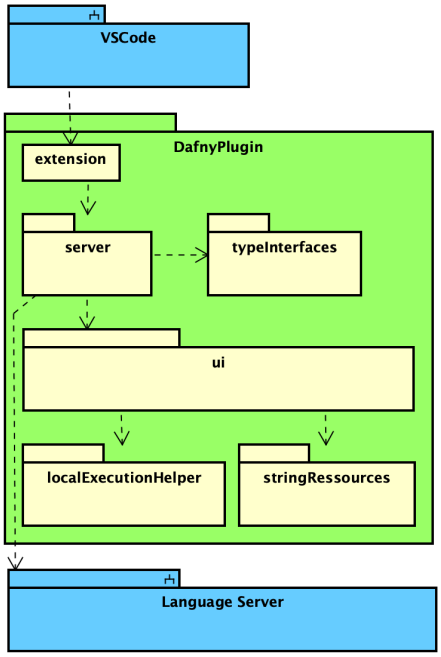

Figure 4: Client Architecture - New Packages

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

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

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

UI – The UI is responsible for all visual displays. Especially VSCode commands and context menu additions. Core components like the status bar are also included in this package.

LocalExecutionHelper – This package contains small logic extensions like the execution of compiled dafny

files. The UI package accesses this package.

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

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

5.1.3 Components

In the following figure 5, the components within the packages are also shown for a more detailed view. The contents of type interfaces and string resources have been omitted for clarity.

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

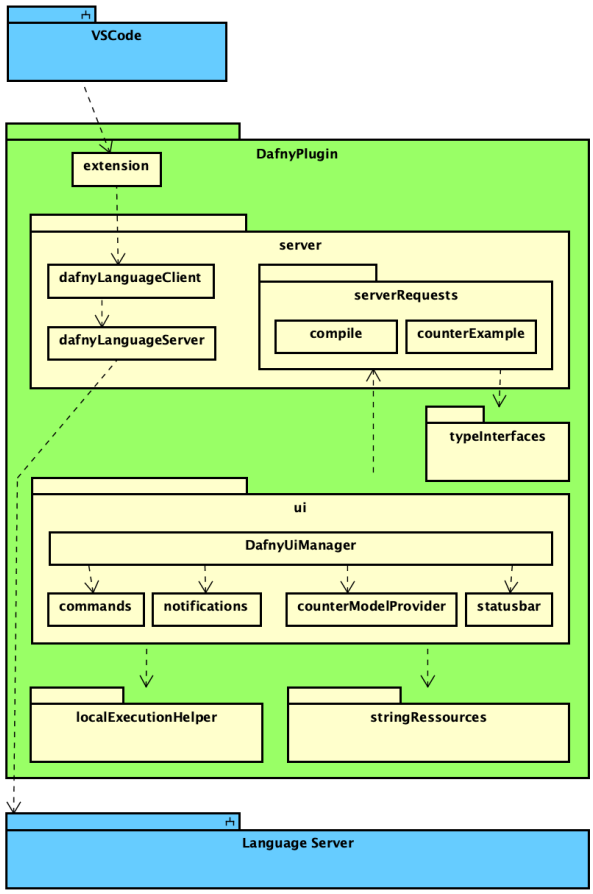

Figure 5: Client Architecture - Components

Figure 6 shows the public methods for the components. All instance variables were set to private. Constructors were not included for simplicity. The contents of type interfaces and string resources were also omitted for clarity

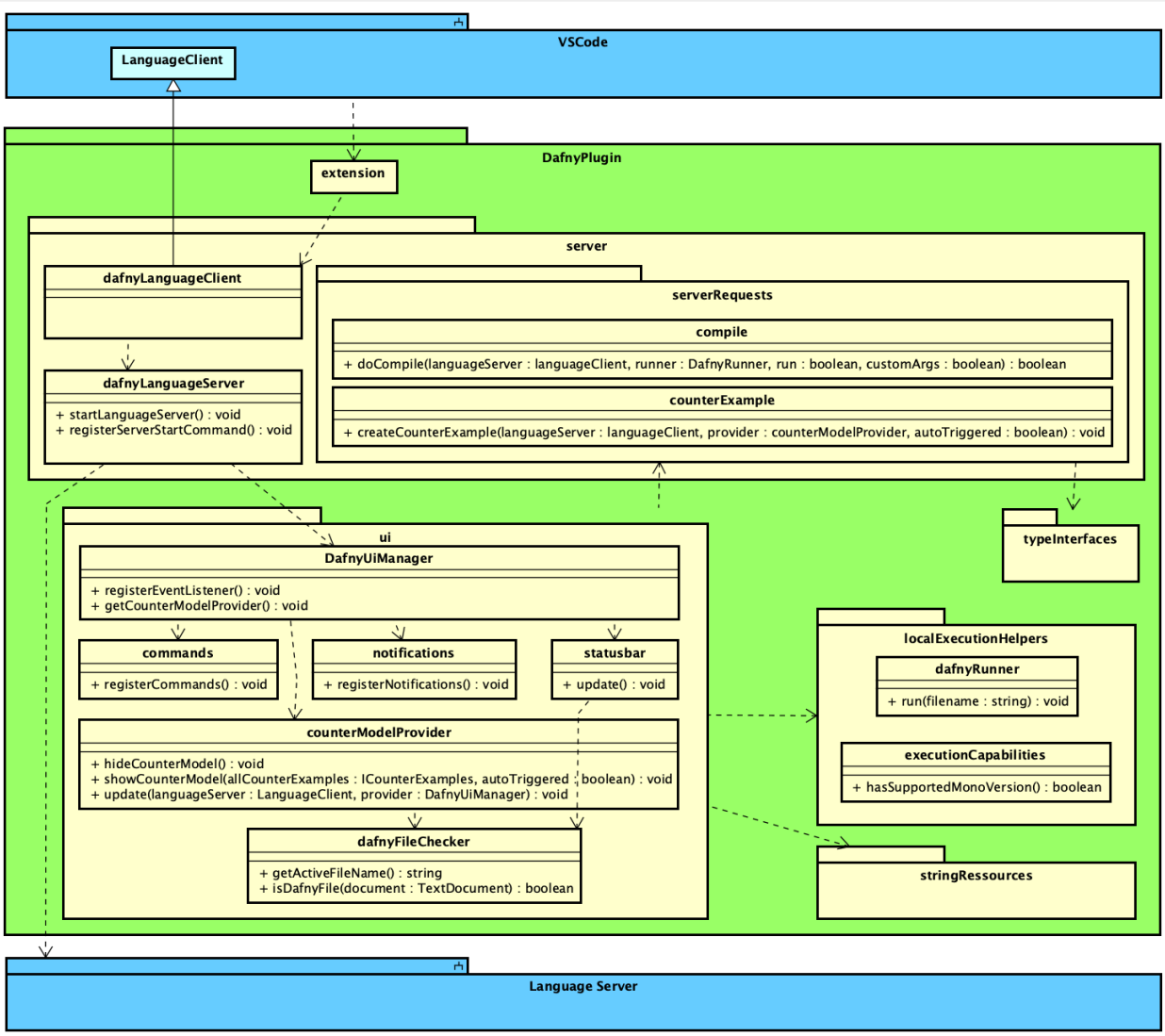

Figure 6: Client Architecture - Components and Public Methods

There is also a compromise that we have made. This distribution has certain upward dependencies, which is not perfect. The UI package accesses the server requests to be exact. Nevertheless, we have decided on this grouping, so that the server access functionality is encapsulated.

5.1.4 Logic

The logic contained in the client has been reduced to a minimum. This mainly with the goal of making porting to other IDEs as easy as possible. This subchapter describes in rough outlines the logic still contained in the client and its benefits.

Server Connection – Starting the Language Server and send API requests. In addition, the client has a simple logic that certain server requests (such as updating the counter example) are sent a maximum of twice per second.

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

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

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

Statusbar – The information content for the Statusbar is delivered completely by the Language Server. The client only takes care of the user friendly display with icons and help texts. Therefore certain event listeners must be registered, which react to the server requests. Furthermore the received information is buffered for each Dafny file. This allows the user to switch seamlessly between Dafny files in VSCode without having the server to send the status bar information (like the number of errors in the Dafny file) each time.

Counter Example – The Counter Example has a similar buffer as the Statusbar. For each dafny file in the workspace a buffer stores if the user wants to see the Counter Example. This way the Counter Example is hidden when the user switches to another file and automatically shown again when switching back to the previous Dafny file.

5.1.5 Types in TypeScript

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

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

There are individual Built-In Data Types like Number, Boolean and String [9]. For this purpose we have defined separate interfaces for each of our own types. With an interface we make a promise for a data type what kind of content will be contained. For each separate type of non-standard LSP features (such as Compile and Counter Example), server request arguments and server responses as results are defined as interfaces [10].
¡¡¡hier vielleicht nich beispiel screenshots oder codes rein? ¿¿¿ 2do

5.2 Integration Tests

Unlike in the preceding semester thesis, integration tests could be implemented using Omnisharp's Language Server Client [11]. Each test starts a language server and a language client, then they connect to each other. Now, the client can send supported requests, for example "get me the counter examples for file ../test.dfy". The result can be directly parsed into our `CounterExampleResults` datastructure and be compared to the expectation. Thus, tests can be written easily and are very meaningful and highly relevant.

5.2.1 Dafny Test Files

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

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

Listing 10: Test File Reference

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

5.2.2 String Converters

Many tests return results in complex data structures, such as `CounterExampleResults`. Comparing these against an expectation is not suitable, since many fields and lists had to be compared to each other.
To be able to easily compare the results against an expectation, a converter was written to translate the complex data structure into a simple list of strings. For example, each counter example will be converted into a unique string, containing all information about the counter example. All counter examples together are assembled within a list of strings. This way, they can be easily compared against each other.
Since not only counter examples, but also other data structures such as `Diagnostic` were converted into lists of strings, the converters were held generic as far as possible. The following listing shows how this was realized. The method takes 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.

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

Listing 11: 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.

```
public static List<string> ToStringList(this List<CounterExample> source)
{
    return GenericToStringList(source, ToCustomString);
}

public static string ToCustomString(this CounterExample ce)
{
    if (ce == null)
    {
        return null;
    }
    string result = $"L{ce.Line} C{ce.Col}: ";
    result = ce.Variables.Aggregate(result, (current, kvp) => current + $"{kvp.Key} = {kvp.Value};
        ");
    return result;
}
```

Listing 12: 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:

```
List<string> expecation = new List<string>()
{
    "L3 C19: in1 = 2446; ",
    "L9 C19: in2 = 891; "
};
```

Listing 13: 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 [12].

6 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:

```
[Test]
public void FailureSyntaxErrorSemiExpected()
{
    RunCompilation(Files.cp_semiexpected);
    VerifyResults("Semicolon expected in line 7.");
}
```

Listing 14: 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.

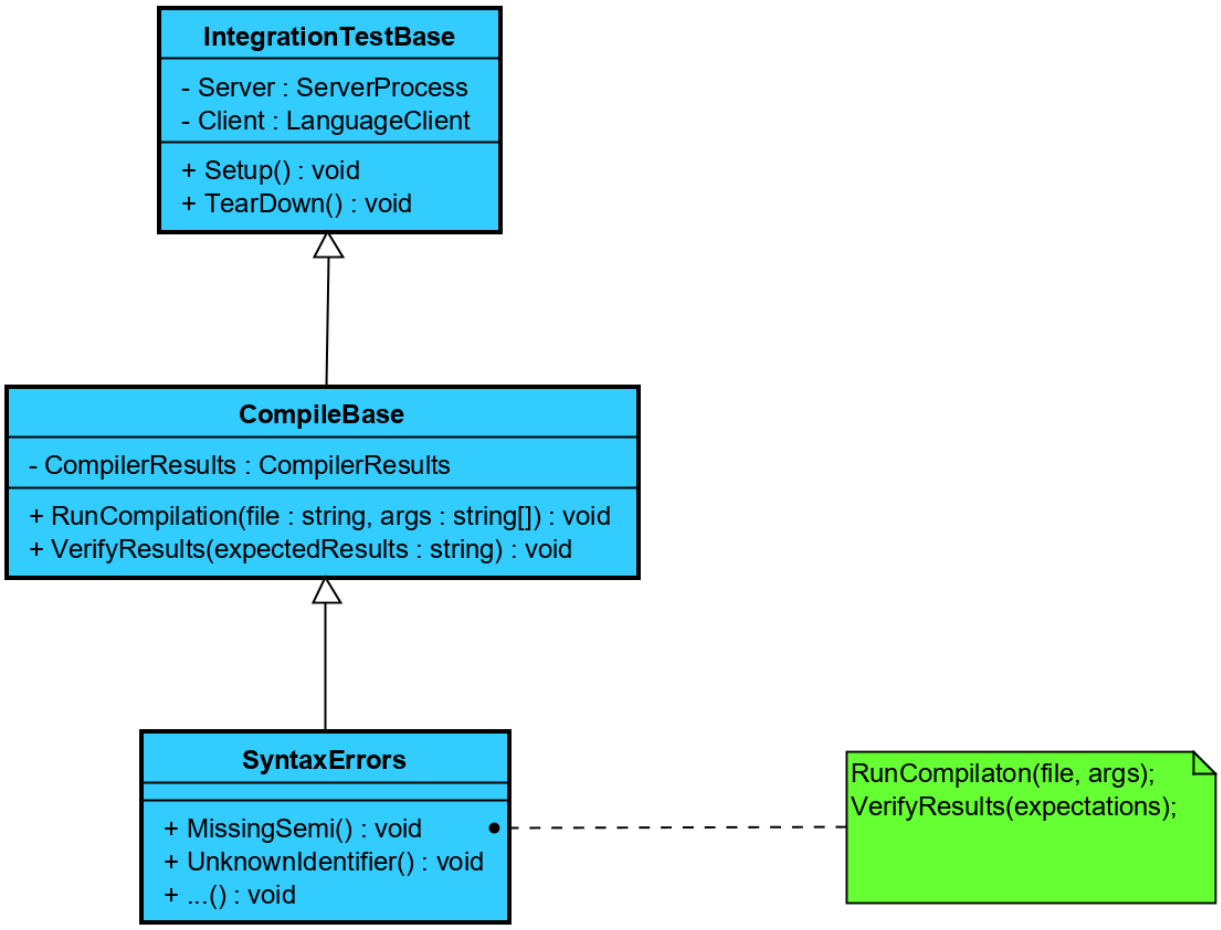

Figure 7: Test Architecture on the Basis of Compilation

7 Implementation

8 Result

9 Conclusion

10 Project Management

Glossary

IDE Eine integrierte Entwicklungsumgebung für die Herstellung von Software. 22

References

[1] Rafael Krucker and Markus Schaden. *Visual Studio Code Integration for the Dafny Language and Program Verifier*. https://eprints.hsr.ch/603/. HSR Hochschule für Technik Rapperswil, 2017.

[2] Marcel Hess and Thomas Kistler. *Developer Documentation*. HSR Hochschule für Technik Rapperswil, 2020.

[3] 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.

[4] *Functions vs. Methods*. URL: https://www.engr.mun.ca/~theo/Courses/AlgCoCo/6892-downloads/dafny-notes-010.pdf. (Accessed: 15.04.2020).

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

[6] *SonarCloud for C# Framework Project*. URL: https://community.sonarsource.com/t/sonarcloud-for-c-framework-project/17132. (Accessed: 23.03.2020).

[7] *OpenCover*. URL: https://github.com/OpenCover/opencover. (Accessed: 23.03.2020).

[8] *monocov*. URL: https://github.com/mono/monocov. (Accessed: 23.03.2020).

[9] *Data Types in TypeScript*. URL: https://www.geeksforgeeks.org/data-types-in-typescript/. (Accessed: 15.04.2020).

[10] *TypeScript: Class vs Interface*. URL: https://medium.com/front-end-weekly/typescript-class-vs-interface-99c0ae1c2136. (Accessed: 15.04.2020).

[11] *OmniSharp Language Client*. URL: https://www.nuget.org/packages/OmniSharp.Extensions.LanguageClient/. (Accessed: 15.04.2020).

[12] *Nunit CollectionAssert*. URL: https://github.com/nunit/docs/wiki/Collection-Assert. (Accessed: 15.04.2020).

Anhang (Entwickler Doku)