

Dafny Language Server Redesign

Term Project

Department of Computer Science
University of Applied Science Rapperswil

Fall Term 2019/20

Authors: Marcel Hess
Thomas Kistler

Advisors: Thomas Corbat
Fabian Hauser

Assignment for Term Project “Dafny VSCode Server Redesign”

Marcel Hess / Thomas Kistler

1. Supervisor and Advisor

This term project will be conducted with the Institute for Software at HSR. It will be supervised by Thomas Corbat (tcorbat@hsr.ch) and Fabian Hauser (fhauser@hsr.ch), HSR, IFS.

2. Students

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

- Marcel Hess (mhess@hsr.ch)
- Thomas Kistler (tkistler@hsr.ch)

3. Introduction

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

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

In a preceding bachelor thesis at HSR a Visual Studio Code plug-in to support Dafny development has been developed. It facilitates a language server for source code analysis and aids the programming with context sensitive completion suggestions, automated refactorings and performs formal verification on the fly (Dafny VSCode Server). This language server is accessed through the language server protocol (LSP). The VSCode Server relies on the DafnyServer for these analyses, which is accessed through a proprietary API¹. For a visual overview of the architecture see Figure 1.

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

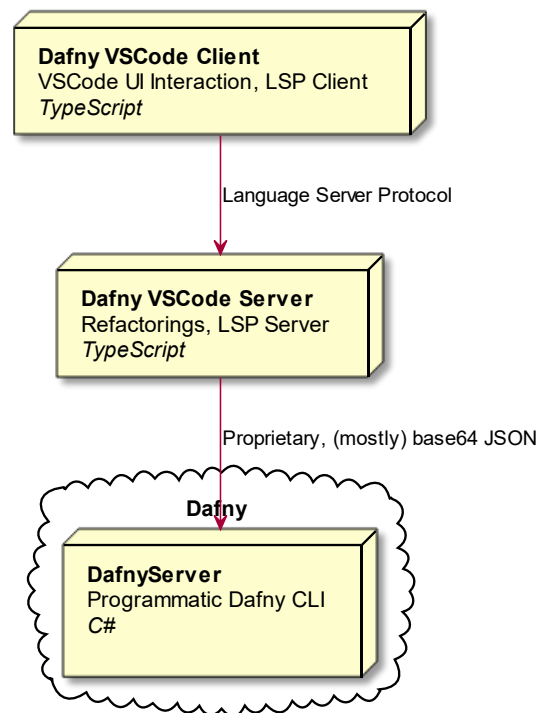


Figure 1 Dafny VSCode Plug-in Architecture Overview

4. Goals of the Project

The primary goal of this project is the reimplementing of the Dafny VSCode Server in C# featuring a tighter integration with the DafnyServer, which is implemented in C# as well. This should make the implementation of further features for the Visual Studio Code plug-in simpler by reducing the complexity of additional API indirection. Furthermore, this will enable integration of the Dafny language analysis features into other integrated development environments.

As the previous solution the reimplementing must be functional on Windows, Linux and MacOS.

Beside the current set of language features available in the Dafny VSCode Server, many extensions are possible as optional goals. For example:

- Debugger (Debug Adapter Protocol)
- Better refactoring support (See thesis)²
- Better contract generation (See thesis)²
- Configuration of compilation arguments
- Display variable types on hover text

² <https://eprints.hsr.ch/603/>

5. Documentation

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

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

6. Important Dates

16.09.2019	Start of the semester and the term project
Until 16.12.2019	Hand-in of the abstract to the supervisor for checking (on abstract.hsr.ch)
20.12.2019, 17.00	Final hand-in of the report through archiv-i.hsr.ch

7. Evaluation

A successful term project counts as 8 ECTS point. The estimated effort for 1 ECTS is 30 hours. (See also the module description ³). The supervisor will be in charge for all the evaluation of the project.

Criterion	Weight
1. Organisation, Execution	1/5
2. Report (Abstract, Management Summary, technical and personal reports) as well as structure, visualization and language of the whole documentation	1/5
3. Content	3/5

Furthermore, the general regulations for term projects of the department “Informatik” apply.

Rapperswil, 16. September 2019

Thomas Corbat

Lecturer

Institut für Software (IFS)

Hochschule für Technik Rapperswil

³ https://studien.hsr.ch/allModules/24386_M_SAI14.html

Table of Contents

1. Abstract	25
2. Management Summary.....	25
2.1 Dafny.....	26
2.2 Initial Solution	27
2.3 Motivation.....	27
2.4 Goals	28
2.5 Results.....	29
3. Introduction.....	29
3.1 Problem Domain.....	31
3.2 Relevance.....	32
3.3 Outlook	32
4. Analysis.....	33
4.1 Existing Thesis	33
4.1.1 Use Cases of the Existing Plugin	35
4.1.2 Extension Points	35
4.2 Visual Studio Code Extensions	36
4.3 Existing Code	36
4.3.1 Client.....	36
4.3.2 Language Server	36
4.3.3 Dafny Server.....	37
4.4 Existing Tests	38
4.4.1 Integration Tests.....	39
4.4.2 End to End Tests	39
4.5 Language Server Protocol	39
4.5.1 Message Types	40
4.5.2 Communication Example	40
4.5.3 Message Example.....	41
4.6 OmniSharp	42
4.6.1 Basic OmniSharp Usage	43
4.6.2 Custom LSP Messages with OmniSharp.....	44
5. Design.....	44
5.1 Basic Design Decisions	44
5.2 Client.....	45
5.3 Server	45
5.3.1 Entrance Point.....	45
5.3.2 Handlers.....	46
5.3.3 Services	46

5.3.4 Dafny Translation Unit.....	46
5.3.5 Buffering	46
6. Implementation	46
6.1 Technologies Overview.....	46
6.2 Architecture Overview.....	47
6.3 Dafny Translation Unit.....	47
6.4 Key Components	47
6.4.1 BufferManager	48
6.4.2 DafnyFile	48
6.4.3 FileHelper.....	48
6.4.4 General LSP Sequence	50
6.5 Features	51
6.5.1 Syntax Highlighting	52
6.5.2 Verification.....	52
6.5.3 Compile.....	52
6.5.4 Autocompletion for Identifiers.....	53
6.5.5 Counter Example	54
6.5.6 Go to Definition	55
6.5.7 CodeLens.....	56
6.6 Testing.....	57
6.6.1 Unit Tests	57
6.6.2 Integration Tests.....	57
6.6.3 System Tests.....	58
6.7 Project Automation	36
6.7.1 Environment.....	36
6.7.2 Prebuild Stage	36
6.7.3 Build.....	36
6.7.4 Tests.....	37
6.7.5 Sonar Scanner	38
7. Results	39
7.1 Syntax Highlighting	39
7.2 Verification.....	39
7.3 Compile	40
7.4 Counter Example	40
7.5 Auto Completion for Identifiers	41
7.6 Go to Definition	42
7.7 CodeLens.....	43
8. Conclusion	44

8.1 Reflection per Feature	44
8.1.1 Verification.....	44
8.1.2 Compile.....	45
8.1.3 Auto Completion for Identifiers	45
8.1.4 Counter Example	45
8.1.5 Go to Definition	46
8.1.6 CodeLens.....	46
8.2 Extension Points of the Previous Thesis	46
8.3 Short-Falling Features of the Previous Thesis	46
8.3.1 Installation, Setup, Marketplace Integration	46
8.3.2 Platform Independence	46
8.3.3 Features	47
8.4 CI Reflection	47
8.5 Planned Extensions in the Bachelor Thesis.....	47
9. Project Management	48
9.1 Meetings	48
9.2 Time Management	48
9.2.1 Project Management.....	50
9.3 Scope	51
9.4 Code Quality Aspects and Metrics.....	52
9.4.1 Code Reviews	52
9.4.2 Backend.....	52
9.4.3 Frontend	53
9.4.4 Test Coverage.....	54
9.4.5 Commit Activities	55
9.5 Infrastructure	56
10. Glossary	57
10.1 Acronyms	57
10.2 Technical Terms.....	57
11. References	58

Appendix

- Developer Documentation
- Project Plan

1. Abstract

Dafny is a formal programming language to prove a program's correctness with preconditions, postconditions, loop invariants and loop variants. In a previous bachelor thesis, a plugin for Visual Studio Code was created to support Dafny. Developers can profit by several features from this plugin. For example, if Dafny cannot prove a postcondition, the code will be highlighted and a counter example can be shown. Other features include code compilation, auto completion suggestions and various refactoring tools. The plugin communicates with a language server. This interface was realized with the language server protocol. This protocol is a specification which allows an easy communication between an IDE and a language server. The language server itself was using an additional JSON-interface to request information from the actual Dafny library.

The goal of this thesis was to rewrite the language server and directly integrate it into the Dafny library to save one of the communication paths. In order to simplify the development of further features, the tighter integration was inevitable. Instead of making complex adjustments to the server API, one can just use the language server protocol to increase functionality. For this, the new language server had to be rewritten from TypeScript to C#.

The old language server could be completely superseded by our new, rewritten server in C#. Since we are using the same project solution as the Dafny library, Dafny's methods and classes are directly accessible. That means, that the interface from the old language server to the Dafny library is now obsolete. Most features of the previous bachelor thesis are supported with the new language server. These include code verification, error highlighting, making suggestions for auto completion, code lens, go to definition, compilation, as well as syntax highlighting. Since the language server is integrated with the Dafny library, new features can be added in a convenient way. Thus, we will continue to work on the scope in the subsequent bachelor thesis.

2. Management Summary

This chapter provides a less technical summary for the thesis. First, the given situation is explained. Next, the motivation and the goals for the project are listed. Finally, the results are presented.

2.1 Dafny

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 we require it to be positive. This is the precondition. In the code, the variable b is assigned the negative of a . Thus, we ensure, that b must be negative, which is the postcondition.

```
method demo(a: int) returns (b: int)
  requires a > 0
  ensures b < 0
{
  b := -a;
}
```

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 and no tests for the code are necessary.

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 disadvantage of their implementation is that there are three distinctive parts of the software.

First of all, there is the actual Visual Studio Code plugin, named “Dafny Client”. The plugin communicates with a language server. This interface uses the so-called language server protocol, an easy way to exchange text document changes or other relevant information.

However, the server has to communicate with Dafny itself, too, to receive the required correctness proofs. This happens in the Dafny library where Dafny code gets compiled for Boogie and checked with Z3. As you can see in the figure below, the interface from the language server to the Dafny library uses yet another communication specification named JSON. Any information transmitted over this interface had to be parsed from plain text.

This results in several disadvantages, since data can be lost in this parsing process. The situation was rather messy and complex. Expanding the plugin with additional functionality was very cumbersome with the initial implementation.

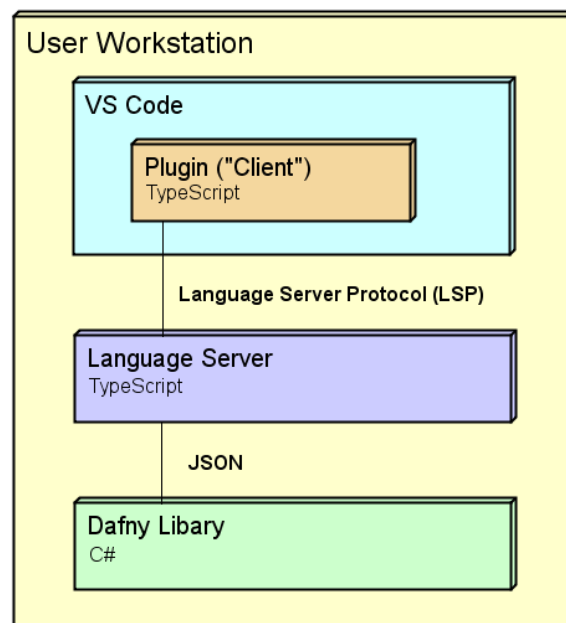


Figure 1 - Old Solution

2.3 Motivation

The figure below shows the new organization that should be achieved in this project. It is much more concise. Thus, there is less room for failures. In software, each communication path is a potential source of errors. Instead of communicating with JSON, the Dafny library can now be directly accessed by the new language server. This way, it is very easy to collect the necessary information firsthand inside the server. There is no longer a loss of information due to parsing.

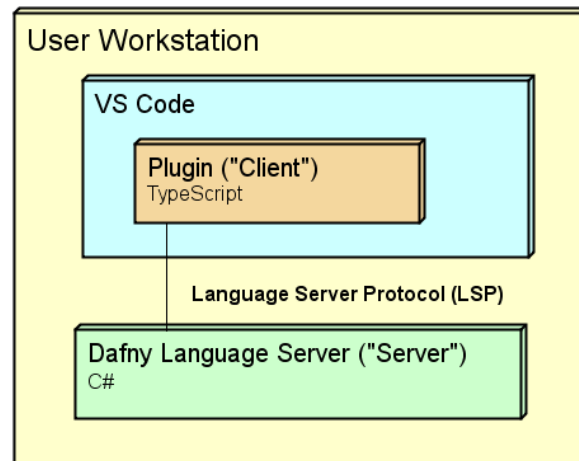


Figure 2 - New Situation

The old solution did not allow an easy extension of the feature-set. Complex adjustments were necessary and two communication paths had to be considered. In order to make this easier, the server integration was inevitable. If one wants to extend the functionality now, a developer can just extend the server and has all functionality of Dafny available.

2.4 Goals

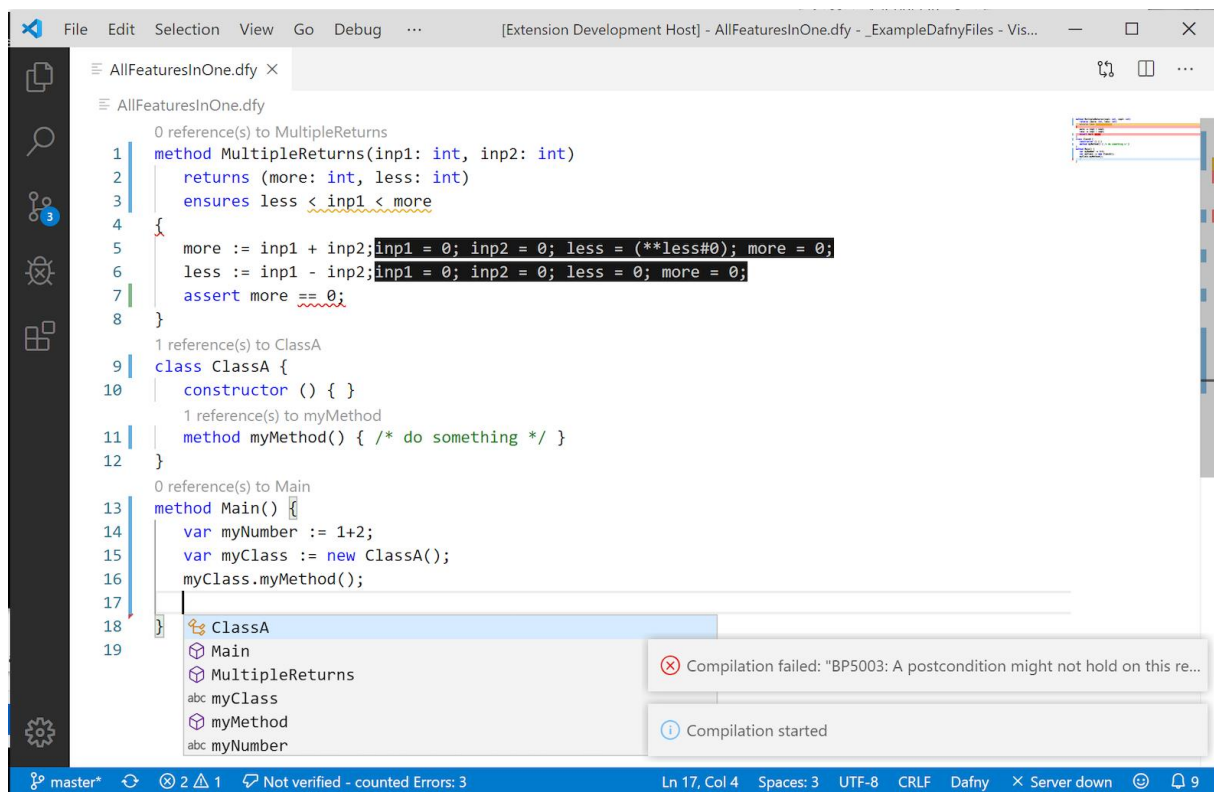
The client is rather trivial and basically just sends requests to the server. Imagine a programmer writing Dafny code. The client just sends his code to the server and asks, what should happen next. The server will then reply and tell for example that line number 5 has to be underlined in red due to a proof which could not be verified. Because of this simplicity of the client, it could mainly be left unchanged compared to the previous bachelor thesis. Just some simple adjustments to the displaying of information had to be done, as well as the connection to the server was subject for reconfiguration.

Our main goal was to make changes on the server side. Since a language server using the language server protocol (LSP) can be implemented in most programming languages, including C#, it was a natural step to rewrite the existing language server in C#. This is to match the language of the Dafny library. It was then possible to integrate the language server directly into the Dafny library. This means that one of the components in Figure 1 could be completely omitted and the JSON parsed communication is no longer necessary.

2.5 Results

The old language server was completely rewritten in the programming language C#. The old language server is no longer in use. Our new language server was then integrated into the Dafny library. This means that Dafny and the language server are within the same software project and they can directly talk to each other, exchanging information and use each other's functionality.

Aside the rewriting and the integration, we also had to support the features the old server offered. Features are for example the underlining of postconditions that can't be proven, or making suggestions for what the programmer may want to type. A basic feature-set could be implemented. You can see a selection of functions in the following Figure 3.



```
0 reference(s) to MultipleReturns
1 method MultipleReturns(inp1: int, inp2: int)
2   returns (more: int, less: int)
3   ensures less < inp1 < more
4 {
5   more := inp1 + inp2; inp1 = 0; inp2 = 0; less = (**less#0); more = 0;
6   less := inp1 - inp2; inp1 = 0; inp2 = 0; less = 0; more = 0;
7   assert more == 0;
8 }
9
1 reference(s) to ClassA
9 class ClassA {
10  constructor () { }
11  1 reference(s) to myMethod
12  method myMethod() { /* do something */ }
13 }
14
0 reference(s) to Main
13 method Main() {
14  var myNumber := 1+2;
15  var myClass := new ClassA();
16  myClass.myMethod();
17 }
18
19
```

ClassA
Main
MultipleReturns
abc myClass
myMethod
abc myNumber

Compilation failed: "BP5003: A postcondition might not hold on this re..."
Compilation started

Figure 3 – Features Postcondition, Counter Example, Auto Completion, Code Lens and Compilation

The current solution supports the following features:

Postcondition Violations

Whenever a postcondition statement does not hold, the user will be informed.

Counter Example

When a postcondition is violated, the user can request a counter example. The client will then show values under which the postcondition does not hold.

Auto Completions

The plugin will check which names for symbols already appeared in the Dafny source code and will make proper suggestions.

Compilation

The user can create an executable file out of his code. If he wants, he can also select to directly run it inside the editor.

Code Lens

Whenever the user creates so-called functions and classes, he will be informed how often these are used.

Go to Definition

The user may make use of methods he programmed earlier. Often, a programmer is not exactly sure how that method was implemented. To revise it, he can just hit a hotkey and the cursor of the code editor will jump to the definition.

3. Introduction

This chapter outlines the problem domain and the relevance of this thesis.

3.1 Problem Domain

The project aims to integrate the language server, given by the previous bachelor thesis by Markus Schaden and Rafael Krucker [2] into the Dafny library. While there is no direct advantage for the user, future extensions of the software are easier and it is less error prone. Since the project will be continued in the next semester with a subsequent bachelor thesis, the plugin does not have to be released at the end of this thesis. Of more importance is to understand the used technologies, finish the server integration and set a good base for further development.

3.2 Relevance

Dafny is not a widely used programming language, but has its user base. The language has the advantage that the correctness of the code can be proven. Tests are no longer necessary. For crucial applications like used in finance and security, programming languages like Dafny are indispensable. While the existing plugin is easy to use and offers a basic set of features, it is not yet as versatile as the support for other programming languages is. It is therefore of major significance to set a good base, so that the plugin can be extended by more functionality in the future.

To do so, the language server and the Dafny library get united in the new version. This eliminates a communication channel and simplifies the architecture.

3.3 Outlook

With the integrated server, it is relatively easy to support additional text editors aside Visual Studio Code and to add more features. Dafny could thus become more accessible and grow in popularity. This would motivate the usage of formal programming languages.

4. Analysis

Our thesis started by analyzing the existing plugin by Markus Schaden and Rafael Krucker.[4] This covered studying the documentation as well as the code. We then had to find proper tools to rewrite the server in C# and figure out, how we could integrate the new server into the Dafny library.

4.1 Existing Thesis

The main goal of the thesis written by Markus Schaden and Rafael Krucker was to make Dafny more accessible. Unlike other programming languages, the installation of Dafny was very cumbersome, the usage not practical at all. Thus, they implemented a plugin for Visual Studio Code that could simply be installed and updated with the marketplace. The language server and the Dafny library were installed during this process, too. The user did not have to care about the required software in the background. Therefore, the user could instantly start to use the provided features in a convenient way.

The plugin by Markus Schaden and Rafael Krucker communicated with a language server.[2] Both, the plugin and the server, were written in TypeScript. To connect to the Dafny library, they wrote a translation unit, which acted as a console based Dafny server. The language server had to parse the console outputs and return them as JSON to the client. The code is further explained in chapter 4.3

4.1.1 Use Cases of the Existing Plugin

Aside the easy installation, the previous bachelor thesis provided the following features:

- Code Lens
- Code Completions
- Go to Definition
- Rename Element
- Syntax Highlighting
- Counter Examples
- Null Checks
- Bound Checks
- Increase / Decrease / Invariant Guards
- Flow Graphs

To read more about these use cases, please refer to thesis.[2]

4.1.2 Extension Points

Mr. Schaden and Mr. Krucker already stated a variety of extension points.[2] Since we continued to develop the plugin, these possible extensions were also relevant for us. Those included:

- Support of other IDE's, such as Eclipse, Emacs and Monaco
- Debugging
- Widening Scope (improvement of implemented features)
- Contract Generation

4.2 Visual Studio Code Extensions

As we started to analyze the existing code, we noticed that we had a hard time understanding what is exactly happening inside a Visual Studio Code plugin. Thus, we decided to catch up with creating a very basic plugin for Visual Studio Code. VSCode offered some very nice tutorials for this matter. There was a “Your First Extension” tutorial that covers very basic concepts.[5] On the “Extension Guides” page are several more advanced examples available – including some language extension samples.[6] By completing these guides, we learned how to use VSCode for plugin development, how to debug a plugin and how to test it.

4.3 Existing Code

The analysis of the existing code was relatively complex, since the project contained a lot of convoluted source code in various languages. There were three main parts: The client in TypeScript, the language server as well in TypeScript and the Dafny server. The initial architecture is illustrated in Figure 1.

4.3.1 Client

The client, in other words the Visual Studio Code plugin, was written in TypeScript. This is required by the Visual Studio Code Extension API.[7] The main entrance point for the plugin is given by the file `extension.ts`. In this file, the client connects to the language server.

```
const languageServer = new DafnyLanguageClient(extensionContext);
```

Another important file is `commands.ts`, where a variety of commands is registered. For example, if the user wants to compile his file, the proper handler is defined as follows:

```
public compile(...): void {
  [...]
  document.save();
  vscode.window.showInformationMessage(InfoMsg.CompilationStarted);
  this.languageServer.sendRequest<ICompilerResult>(
    LanguageServerRequest.Compile, document.uri)
    .then((result) => {
      vscode.window.showInformationMessage(InfoMsg.CompilationFinished);
      if (run && result.executable) {
        this.runner.run(document.fileName);
      }
    });
}
```

Further commands that were of importance to us, i.e. “show and hide counter model”, were registered in another file by the given code base. We found these commands defined in a different fashion in `dafnyProvider.ts`:

```
private doCounterModel(textDocument: vscode.TextDocument): void {
  this.sendDocument(textDocument, LanguageServerNotification.CounterExample);
}
```

Since no obvious reason was visible for this separation, we decided to define all commands in `commands.ts`.

The client contains further classes with mainly responsibilities for the frontend, such as a class to update the status bar with the amount of errors in the current file.

However, the main job of the client is to register the commands and send them to the language server. This was done in just a few lines of code, as you can see above.

4.3.2 Language Server

The language server part was rather complex since the data has to be extracted from the LSP, then prepared for the API of the Dafny server. Finally, the JSON or plain text responses had to be adapted again and saved back in an LSP compatible format. The whole process acted not just as an adapter, but included a wide range of buffering logic and used many regular expressions for parsing.[8]

4.3.3 Dafny Server

Markus Schaden and Rafael Krucker had to write some parts in C# already.[2] They've built a console server using the Dafny library. This acted as a translation unit between their language server and the Dafny library. They could parse the console output of the Dafny server and read it back in their language server. For this purpose, they used a lot of parsing and regular expressions.[4] Their Dafny console server consisted out of a relatively small main method, which could process a handful of commands like `counterExample`, `verify` or `symbols`. These commands were then redirected to the class `DafnyHelper`, which is the main translation unit to the Dafny pipeline. Aside the `DafnyHelper`, there were further supporting service classes, like the `CounterExampleProvider`, which collects all counter examples Dafny delivers.

While analyzing the code, we noticed the console centered architecture as shown in the following code snippet.[4]

```
public void Symbols() {  
    [...]  
    Console.WriteLine("SYMBOLS_START " + ConvertToJson(symbols) + " SYMBOLS_END");  
    [...]  
}
```

Notice the Keyword "SYMBOLS_START" and "SYMBOLS_END", that were used as parsing stop words back in the language server.

4.4 Existing Tests

This chapter describes the existing test infrastructure. The guiding principle is that all tests have to run automatically within the CI. Manual tests are not carried out at all and are less useful.

4.4.1 Integration Tests

There are a lot of integration tests. Each test consists out of a Dafny file that is handed to the console server. The console output is then compared against a text file, which contains the expected console output. For example, a test result expectation file might look like this: [9]

```
Char.dfy(48,20): Error: assertion violation  
Execution trace:  
    (0,0): anon0  
    (0,0): anon9_Then  
    (0,0): anon10_Then  
Char.dfy(52,20): Error: assertion violation  
Execution trace:  
    (0,0): anon0  
    (0,0): anon9_Then  
    (0,0): anon11_Else  
Char.dfy(63,16): Error: assertion violation  
Execution trace:  
    (0,0): anon0  
    (0,0): anon5_Else  
Char.dfy(81,13): Error: char addition might overflow  
Execution trace:  
    (0,0): anon0  
    (0,0): anon6_Else  
    (0,0): anon7_Then  
Char.dfy(89,7): Error: char subtraction might underflow  
Execution trace:  
    (0,0): anon0  
    (0,0): anon6_Else  
    (0,0): anon7_Then
```

Dafny program verifier finished with 8 verified, 5 errors

4.4.2 End to End Tests

The existing work had a few system tests. Those tests start a Visual Studio Code instance, open a file and then check if certain conditions are met. For example, it is tested if a dedicated file publishes a diagnostic instance with two errors and two proof obligations.

Due to the heavy dependencies of these tests, an automated execution of them is not easy. They require a headless VSCode instance. A better way for end-to-end tests was highly desirable.

4.5 Language Server Protocol

The language server protocol (LSP) is a JSON-RPC based protocol to communicate between an IDE and a language server.[10] In 2016, Microsoft started collaborating with Red Hat and Codenvy to standardize the protocol's specification.[10] 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.[10]



Figure 4 - Communication Benefit of LSP

A big advantage of this is that the IDE specific plugin can be kept very simple. The relevant information is delivered by the language server, which is IDE and language independent. Figure 4 from the VSCode extension guide illustrates the advantages of the LSP. [11]

4.5.1 Message Types

The LSP supports three types of messages.

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

Each message type can be sent from both sides.

4.5.2 Communication Example

The basic idea 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. [12]

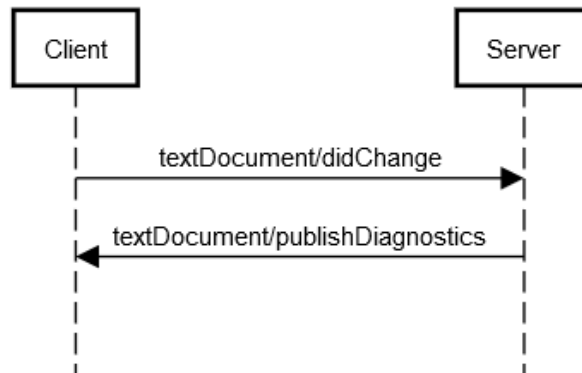


Figure 5 - Example Communication

4.5.3 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.

```
[12:45:29 DBG] Read response body
{
  "jsonrpc":"2.0",
  "method":"textDocument/publishDiagnostics",
  "params":{
    "uri":"file:///D:/[...]/fail1.dfy",
    "diagnostics":[
      {
        "range":{
          "start":{
            "line":4,
            "character":12
          },
          "end":{
            "line":4,
            "character":17
          }
        },
        "severity":1,
        "code":0,
        "source":"file:///D:/[...]/fail1.dfy",
        "message":"assertion violation"
      }
    ]
  }
}
```

4.6 OmniSharp

To work with the language server protocol, we had to use a proper LSP implementation. A quick online research showed that OmniSharp offers support for C#. [13] We could simply install it as a NuGet package. With OmniSharp, many of the parsing and caching problems of the original language server described in chapter 4.3 Existing Code are automatically reduced.

OmniSharp also offers a language server client that we could use later on for our tests.

4.6.1 Basic OmniSharp Usage

Before we started to use OmniSharp in the actual project, we worked our way through a tutorial by Martin Björkström. [14] This tutorial provided us 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.

Since OmniSharp is open source, we could find all available interfaces and thus all available handlers in their git repository. [15] This collection helped us a lot to perceive LSP's possibilities.

4.6.2 Custom LSP Messages with OmniSharp

The current implementation of the problem domain does not only feature premade LSP messages like auto completions or diagnostics, but also custom requests such as "counterexample". Since no example or documentation could be found on custom messages, we contacted Martin Björkström in the OmniSharp Slack channel about this. [16] Martin and his team could instantly provide us with the necessary interfaces that we had to implement.

The server can simply register a handler. The following three items have to be specified:

- Name of the message
- Parameter type
- Response type

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:

```
public class Params : IRequest<Results> { [...] }
public class Results { [...] }

[Serial, Method("messageName")]
public interface IDemoMessage : IJsonRpcRequestHandler<Params, Results> { }

public class MyHandler : IDemoMessage
{
    public async Task<Results> Handle(Params request, CancellationToken c)
    {
        Results r = DoSomething(request);
        return r;
    }
}
```

5. Design

This chapter covers architectural design decision. First, we will have a look at basic decisions that were made. Next, we talk about the client, which is kept as simple as possible. The last part of the chapter covers the server side.

5.1 Basic Design Decisions

The goal of the thesis was to eliminate the language server that was written in TypeScript and integrate it into the Dafny project, which is written in C#. Our aspired component diagram would look as follows.

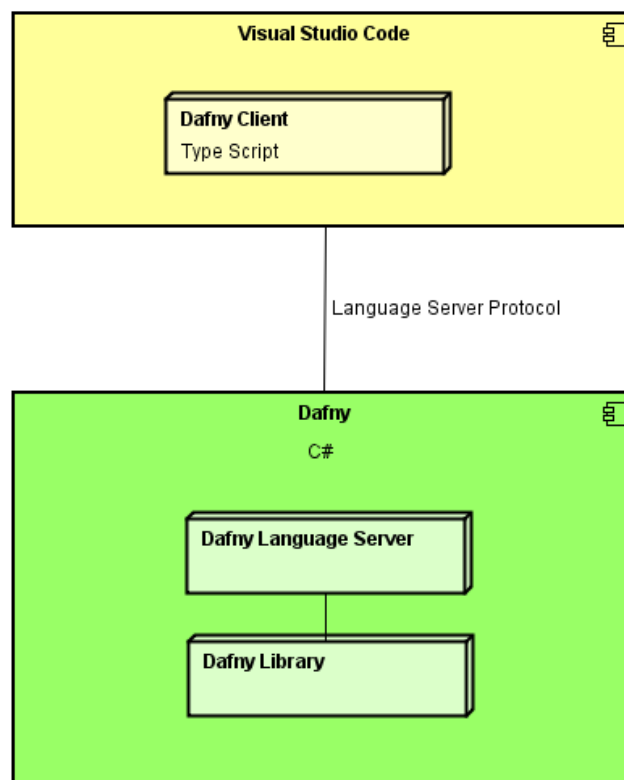


Figure 6 - Aspired Component Diagram

The client was already provided by the previous bachelor thesis. However, the server was meant to change completely. It had to be rewritten in C# and integrated into the Dafny library. A few classes forming a translation unit between the Dafny library and the Dafny server were already programmed by the previous bachelor thesis and could be reused as well.

5.2 Client

The client is supposed to be thin and lightweight. It has basically only four jobs to do:

1. Activate once a `.dfy` file is opened.
2. Connect to the server.
3. Forward any notifications and requests to the server.
4. Process the responses.

The basic plugin configuration could be left as it was by the previous thesis. However, the connection process to the server had to be redone.

During the study of the existing code, we noticed that some design decisions of the previous thesis didn't seem to make much sense. For example, there was a class `commands.ts`, in which most commands were registered. However, other commands like `counterExample` were registered in another class named `dafnyProvider.ts`. To clean things up, we decided to merge all commands into the `commands.ts` file.

The basic client structure is shown in the figure below. A description of each component follows on the next page.

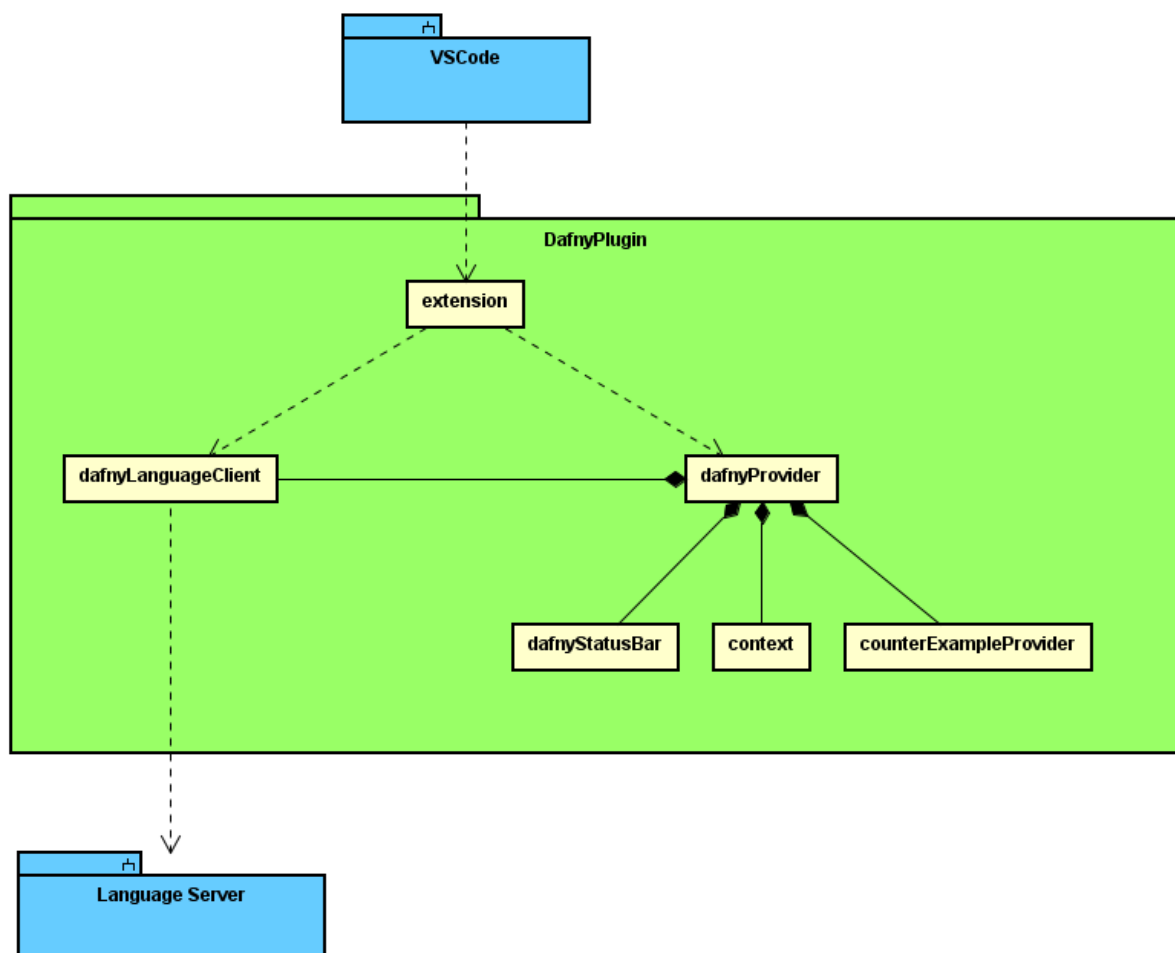


Figure 7 - Class Diagram of the Client (Simplified)

extension.ts

This is the main entrance point. It starts the language client, registers commands, and instantiates `dafnyProvider`.

dafnyLanguageClient.ts

The language protocol client which connects to the server. Messages are sent to the server using an instance of this class.

dafnyProvider.ts

This class is responsible for features that do not base on a command but are always active. This includes sending code verification requests, informing the server whenever a document was changed, opened or closed, as well as keeping track of displayed counter examples.

counterExampleProvider.ts

As the name states, this class is responsible for displaying counter examples. It is a direct member of the `dafnyProvider`. Counter example requests are only sent if the user selects the proper command from the context menu. Thus, the counter example provider is an example of a feature that is not permanently active in the background.

dafnyStatusBar.ts

This class is responsible for handling any changes that will be made to the status bar. This includes showing the error count and such. This class is as well a member of the `dafnyProvider`. It is an example of a class taking care of supplementary tasks that are not core functionality.

context.ts

A class that keeps track of a few global states, such as which file displays which counter examples.

Aside these files, the reader may find a variety of supporting files, such as string resources or interfaces that define types of responses that the server delivers.

5.3 Server

To set up a language server, one needs support for the language server protocol. As stated in chapter 4.6 OmniSharp, we decided to use OmniSharp's implementation. In the main method of our product, we would simply start the language server and register handlers for it. The following figure gives an overview how requests are generally handled. Each component is described in the upcoming subchapters.

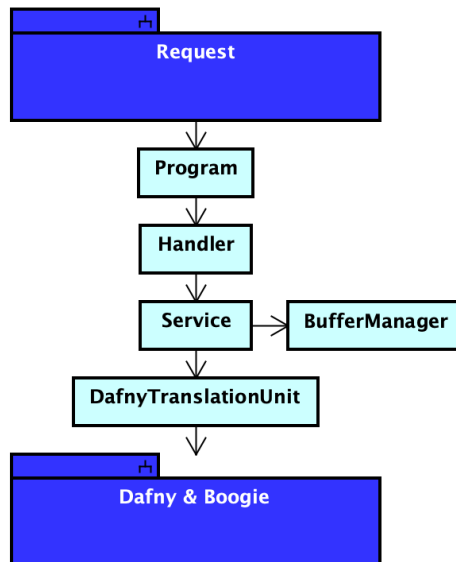


Figure 8 - Schematic Overview of Server Dependencies

5.3.1 Entrance Point

Using the OmniSharp NuGet Package, it is possible to start the server in the Main method of the program. One can then set options, including the request handlers, which are relevant for the architecture.

```
var server = await LanguageServer.From(options =>
    options
        .WithInput(Console.OpenStandardInput())
        .WithOutput(Console.OpenStandardOutput())
        .WithLoggerFactory(new LoggerFactory())
        .AddDefaultLoggingProvider()
        .WithMinimumLogLevel(LogLevel.Trace)
        .WithServices(ConfigureServices)

        .WithHandler<TextDocumentSyncHandler>()
        .WithHandler<CompletionHandler>()
        .WithHandler<CompileHandler>()
        .WithHandler<CounterExampleHandler>()
        .WithHandler<CodeLensHandler>()
        .WithHandler<DefinitionHandler>()
);
```


5.3.2 Handlers

Handlers are classes that are called whenever a request needs to be processed. We decided to keep them simple, provide the necessary skeleton required by OmniSharp, and mainly forward the request to a service. One may argue that this is a “middle man” code smell, as the handler is literally not doing much more than starting a service and delegating the task to it. However, some handlers require a lot of boilerplate code. An example of boilerplate code was already presented in chapter 4.6.2 Custom LSP Messages with OmniSharp. By the partitioning into a handler and a service, we could separate the core logic into an own class, which would make it easier for testing.

Another reason is that services are sometimes used multiple times. A good example is code verification. The code has to be verified either if a document was opened, or when it was changed. The service design decision avoided code duplication a priori.

The following code example shows the usage of a service within the counter example handler.

```
public async Task<CounterExampleResults> Handle(
CounterExampleParams request, CancellationToken cancellationToken)
{
    var file = _bufferManager.GetFile(request.DafnyFile);
    var dtu = new DafnyTranslationUnit(request.DafnyFile, file.Sourcecode);
    var service = new CounterExampleService(dtu);
    return await service.ProvideCounterExamples();
}
```

The actual core logic of the code above just forwards the request to the service. However, the whole handler still contains about 50 lines of code. That is, because the request parameters have to be defined, the response parameters need to be known, and OmniSharp has to know on what request it has to listen as shown in chapter 4.6.2

Thus, the separation between handlers and services is well reasoned.

5.3.3 Services

Services are responsible for handling the actual requests. Most of them will make use of the `DafnyTranslationUnit`, which builds the bridge between our language server and the Dafny library. The services assemble results to the expected format, so that the handler can directly return the result which the service produces. So to speak, the services act as some kind of adapter between the handler and the Dafny translation unit.

5.3.4 Dafny Translation Unit

The `DafnyTranslationUnit` is responsible to preprocess any requests so that the actual Dafny runtime can be involved. Most requests will parse the source code, translate it to `BoogieProgram` instances, and then use the Boogie execution engine to find for example compilation errors or counter examples.

5.3.5 Buffering

The LSP transmits a lot of data. Luckily, it is possible to only transmit changes instead of the whole file. However, also all services need to look up source code often. For this matter, we decided to use a buffer. The buffer is a map containing the file URI as key and a `DafnyFile` as value. `DafnyFile` itself is a class containing all relevant data about the file, including the source code and the symbol table. This would allow us to control access to valid data states, even if the source file is no longer valid as its being edited.

6. Implementation

This chapter gives a brief overview what technologies were used. Afterwards, we discuss major architectural aspects. Then, the feature implementation is described in detail. Next, we talk about how testing was realized. Then, we talk about continuous integration and continuous deployment, which was a distinctive issue during the project, as the project has a lot of external interfaces. Finally, a few metrics are provided.

6.1 Technologies Overview

To manage our project, we used the free tool Redmine. Time tracking, milestone and sprint management, as well as ticketing were done with it.

The server was implemented in C# using .NET Framework, NuGet and OmniSharp. We used Visual Studio with ReSharper to develop the software.

The client is coded completely in TypeScript with Visual Studio Code as the IDE.

Continuous integration and continuous delivery were done using GitLab with Docker container. SonarQube and ReSharper were used to ensure the code quality.

6.2 Architecture Overview

In the `Main` method of our program, we simply start the language server. OmniSharp offers a static method for this. As server options, we need to register our handler components and the `BufferManager`.

The handlers themselves implement proper LSP interfaces such as `ICompletionHandler` and forward requests to services. Both, handlers and services, make use of shared `ContentManager` components. This is illustrated in the figure below. The `DafnyAccess` components on the bottom act as the library layer and are responsible for Dafny parsing and interpretation.

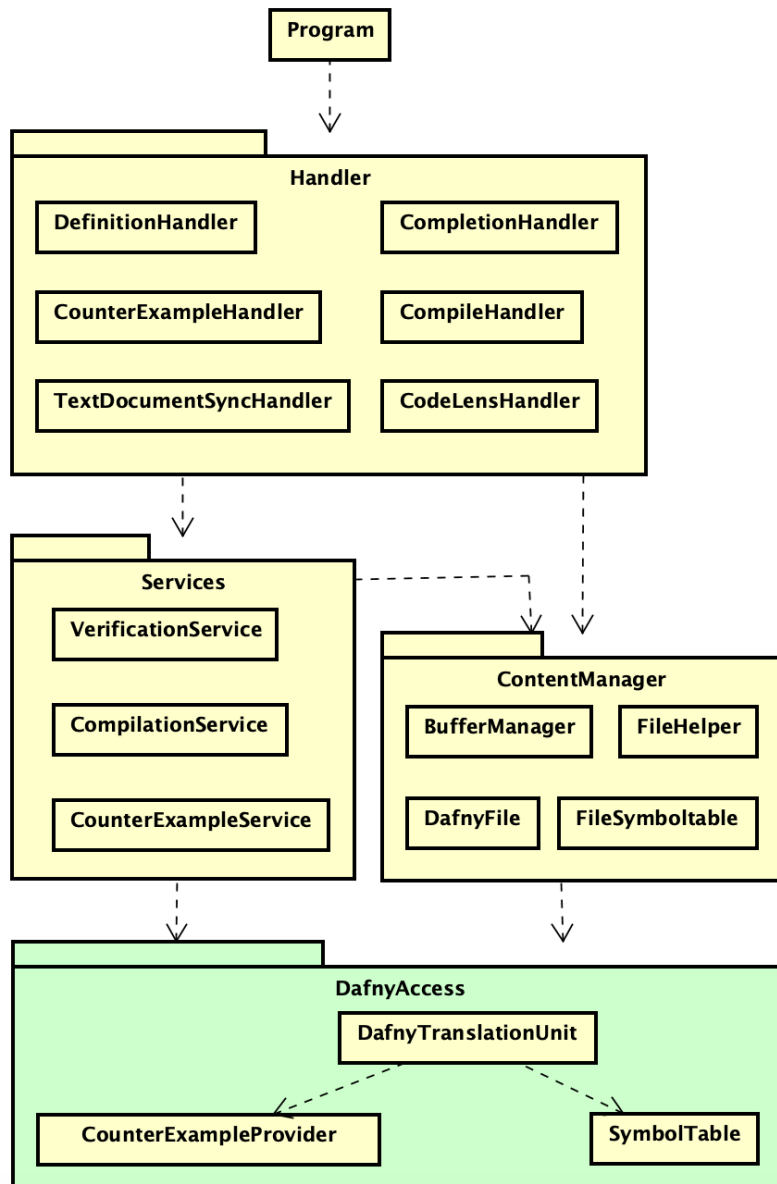


Figure 9 - Package Diagram

6.3 Dafny Translation Unit

As glue code between our Dafny language server and Microsoft Boogie, we created a package `DafnyAccess`. Some parts of it could be taken over from the previous project.[2] The central class in this package is the `DafnyTranslationUnit`. It provides the caller with information like counter examples and symbol tables as you can see in the figure below. Please note that only public fields and methods are shown to keep the general overview clear. In the previous thesis, the name of the translation unit was `DafnyHelper`. We decided that `DafnyTranslationUnit` is a rather fitting name.

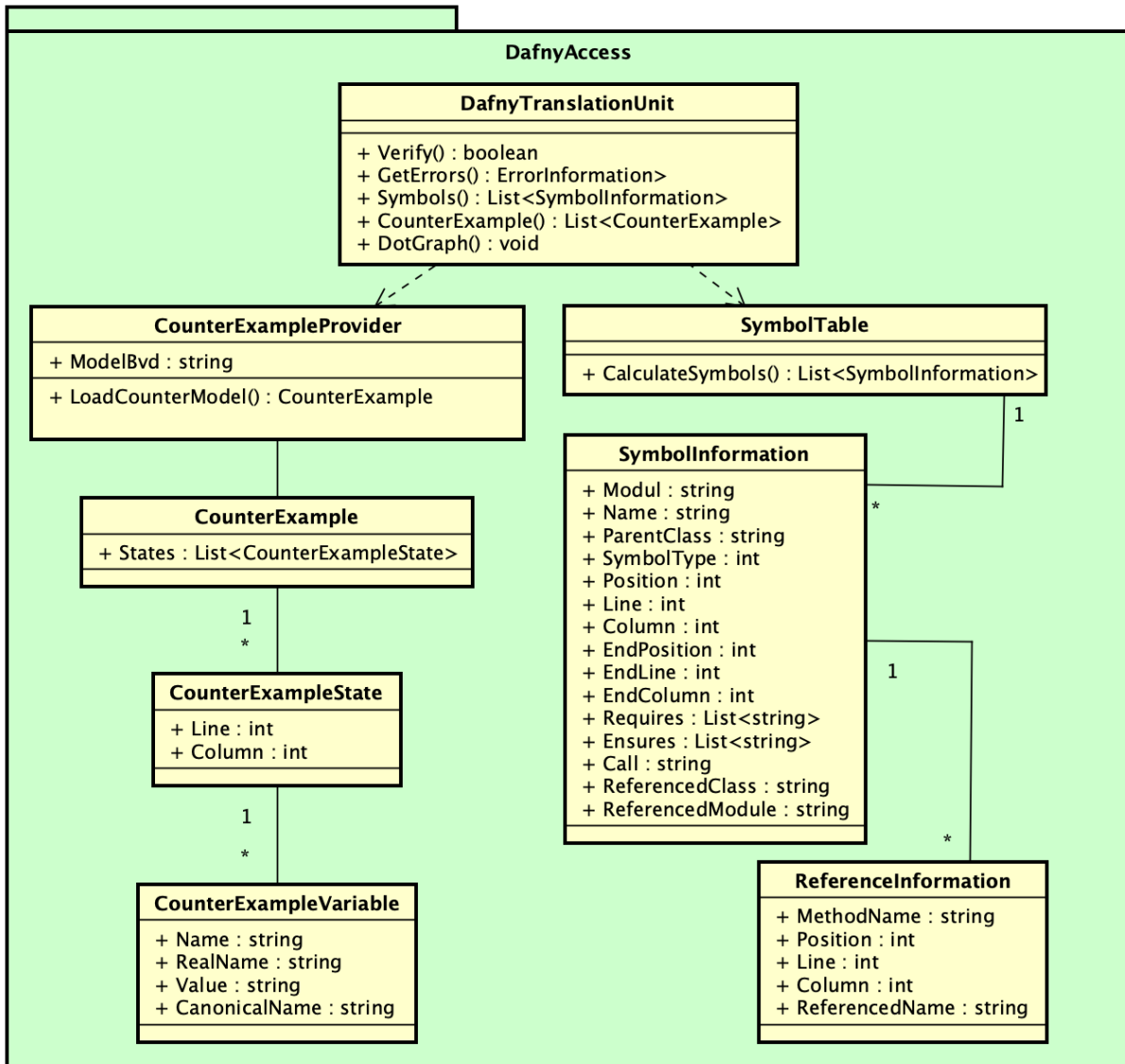


Figure 10 - Package Diagram `DafnyAccess`

We simply copied the `DafnyAccess` package from the previous thesis. This way we could make our adjustments in an isolated environment but still guarantee backwards compatibility. Until the old project is completely superseded, some duplicated code exists as a trade-off.

6.4 Key Components

The ContentManager package offers functionality, which is used by many components on higher layers. Especially the BufferManager is a very important class for many components. In the following figure, you see a general overview of the ContentManager package. Please note that only public fields and methods are shown to keep the diagram simple.

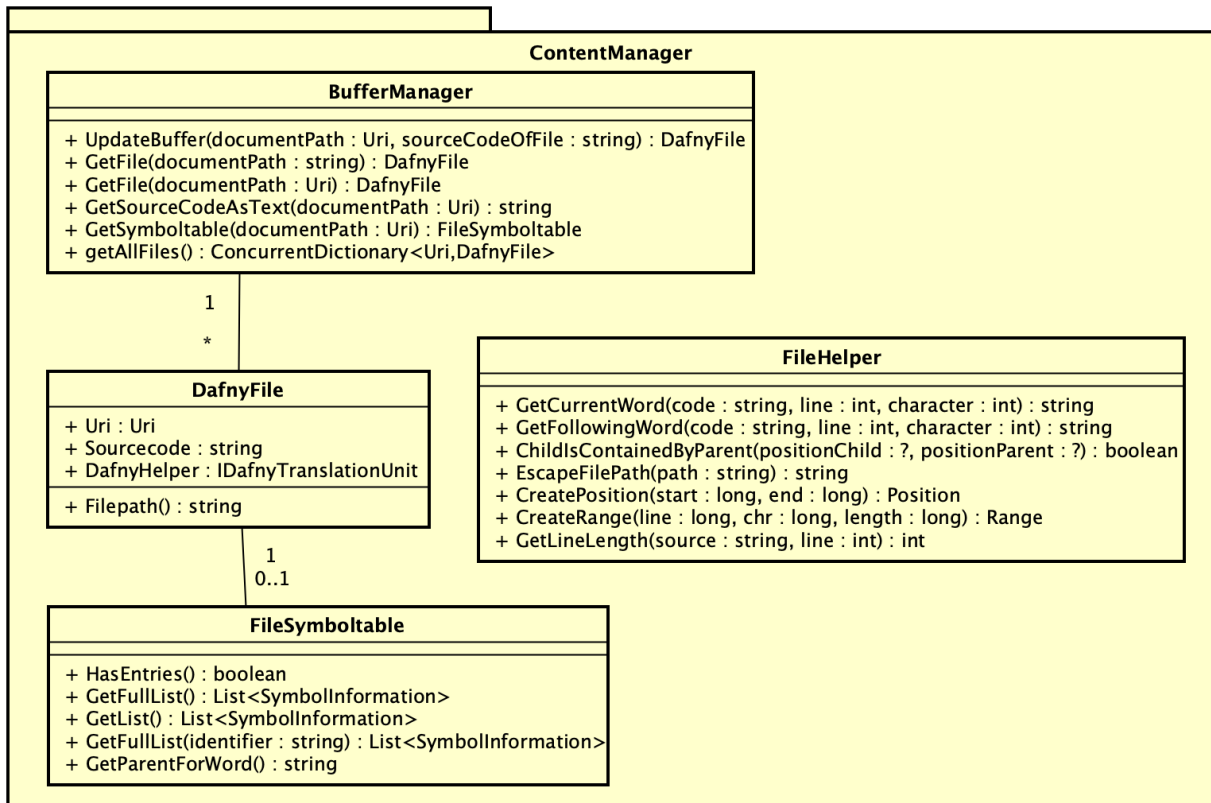


Figure 11 - Class Diagram ContentManager

6.4.1 BufferManager

The BufferManager is a very central component for the whole server side. It caches information about all Dafny files which are being edited by the user. This includes the source code as well as the FileSymboltable. The BufferManager is updated every time a user opens or changes a Dafny document in Visual Studio Code.

6.4.2 DafnyFile

Instead of buffering just the source code with the BufferManager, we decided to create an own class to represent a Dafny file. Many information is stored in this component, including the source code, the symbol table and a reference to the DafnyTranslationUnit.

6.4.3 FileHelper

The FileHelper is mostly used by components that need positional data from Dafny source code. To keep this component completely decoupled and to avoid any direct dependencies to the DafnyFile, the source code has to be handed as an argument. This also means that the FileHelper can be easily tested. Most methods in this class are static.

As you can notice in Figure 11, the method `ChildIsContainedByParent` has question marks as argument types. That is because this method suffers under the code smell “Long Parameter List”. The corresponding refactoring is still pending, but we decided to use the short version in the diagram for clarity.

6.4.4 General LSP Sequence

As mentioned in the previous chapter, each time a user opens or edits a Dafny file in Visual Studio Code, an LSP message is sent to the server with the file changes. The `BufferManager` then updates its buffer for the specific file.

Whenever any other request is sent, a registered handler will be chosen to process the request. This may or may not use information from the `BufferManager`. Once the result is available, the result is sent back to the client.

The following figure contains a sequence diagram which illustrates this process.

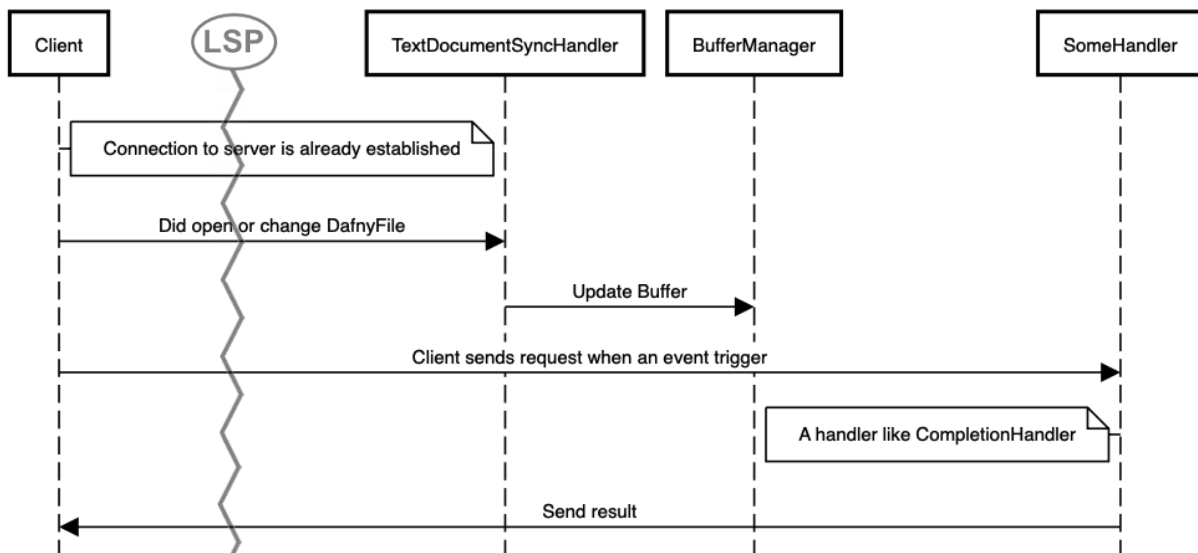


Figure 12 – General LSP Sequence Flow for File Changes and Requests

There are several other events that can be sent from the client to the server and as well from the server to the client. For a deeper explanation, please read the following chapter, which describes each feature in more detail.

6.5 Features

All covered use cases are documented in this chapter. For each feature, we will present how we implemented it and why.

6.5.1 Syntax Highlighting

Syntax highlighting is one of the few exceptions that is not implemented through the language server. It is covered inside the client and worked out of the box from the previous bachelor thesis.[2]

To achieve syntax highlighting for Dafny in Visual Studio Code, one has to configure which keywords Dafny supports, how to match them in the code and how they are classified.

For every Visual Studio Code plugin, the grammar can be defined in the file package.json. You can see the relevant part of the file right below:

```
"languages": [
  {
    "id": "dafny",
    "aliases": [ "Dafny", "dafny" ],
    "extensions": [ ".dfy", ".dfyi" ],
    "configuration": "./language-configuration.json"
  }
],
"grammars": [
  {
    "language": "dafny",
    "scopeName": "text.dfy.dafny",
    "path": "./syntaxes/Dafny.tmLanguage"
  }
],
```

In Dafny.tmLanguage, several regular expressions are defined. Those tell Visual Studio Code how to match keywords and how to classify them for proper coloration. The following code snippet of Dafny.tmLanguage is an example how control keywords are identified.

```
<dict>
  <key>match</key>
  <string>\b(class|trait|datatype|codatatype|type|newtype|function|ghost|var|const|method
|constructor|colemma|abstract|module|import|export|lemma|as|opened|static|protected|twostat
e|refines|witness|extends|returns|break|then|else|if|label|return|while|print|where|new|in|
fresh|allocated|match|case|assert|by|assume|reveal|modify|predicate|inductive|copredicate|f
orall|exists|false|true|null|old|unchanged|calc|iterator|yields|yield)\b</string>
  <key>name</key>
  <string>keyword.control.dafny</string>
</dict>
```

To read more about defining a grammar in VSCode, please refer to the official guide. [17]

6.5.2 Verification

Diagnostics are language server items, that mark errors and warnings in the code. They contain a document location, an error message, and an error severity. Underlining faulty code was the first feature that we implemented. Our starting point for this problem was the method `Verify` inside the `DafnyTranslationUnit`.

The method `Verify` is of a simple structure. It first sets a bunch of environment variables. Then, the methods `Parse`, `Resolve` and `Translate` work on the Dafny source code and translate it into a `boogieProgram`. The `Boogie` method finally checks if there are any logical errors in the code, such as postcondition violations.

```
public bool Verify()
{
    ServerUtils.ApplyArgs(args, reporter);
    return Parse() && Resolve() && Translate() && Boogie();
}
```

To capture any errors which are found, a delegate is passed as an argument to `Boogie`'s execution engine. This delegate adds all errors to a list.

```
public List<ErrorInformation> Errors { get; } = new List<ErrorInformation>();
[...]
```

```
ExecutionEngine.InferAndVerify(boogieProgram, ps, name, e => Errors.Add(e), time);
```

Out of this `ErrorInformation`, we could extract all necessary information for the UI, such as the location and the error message. All that was left to do was to transform the errors from `ErrorInformations` to `Diagnostics`. The diagnostics could then be sent back to the client using the `PublishDiagnostics` notification, which is conveniently specified by the LSP.[12]

In addition to that, we are also sending the total sum of errors to the client as a notification. This allows the client to update the status bar easily. Thus, the status bar always displays the total sum of errors. The client does not need to keep any information about errors for that.

The verification process is called every time the document changes. This is a lot of work for the `Boogie` compiler, but since everything runs asynchronously, no problems arose so far.

Diagnostics contain a property named `RelatedInformation`. This property was used by `Boogie` to specify where the origin of a postcondition violation is. Unfortunately, `OmniSharp` defined the location of this related information as a string, instead with the class `Location`. We therefore couldn't use the related information firsthand. Instead, we just created an own, new `Diagnostics` that marks the related location as a warning. This bug has already been reported to `OmniSharp`.[18]

6.5.3 Compile

VSCoDe plugins can offer commands that the user can execute by opening the context menu (right click), pressing a shortcut or simply by entering them into the command prompt (F1).

The previous bachelor thesis offered a hand full of such commands. Aside server restart options, there was the `compile` command. The previous thesis implemented this by sending a `compile` notification to the server. The server then started `Dafny.exe` and handed the opened file as an argument, aside some other arguments.

In our thesis, we wanted to call all Dafny components directly. Just to launch an exe and read out the output didn't seem very clever. However, in consultation with our supervisors, we decided to keep it this way. Thus, only a few lines of code were necessary to implement the `compile` feature.

```
string processOut = "";
process.OutputDataReceived += (sender, args) => processOut += args.Data;
```

As you can see, we collected the console out of `Dafny.exe` in a simple string. To evaluate the string, we had to use regex parsing.

After the file has been compiled, a notification is sent back to the client. This is either an error notification or a confirmation. The client has the option to directly run the compiled file inside Visual Studio Code, too.

6.5.4 Autocompletion for Identifiers

The autocompletion has two different precision modes. In general, a valid Dafny file can be parsed and returns access to a symbol table. This table will be cached in the buffer. Once an autocompletion request is sent, the whole symbol table will be parsed as autocompletion entries and they are returned to the client.

In case there is an autocomplete request for a specific symbol (e.g. for a reference of class "A"), the server filters the symbol table for symbols that have the same parent symbol class (in this case all symbols that contain the parent symbol class "A").

As an additional requirement, every symbol must be in range of the parent class.

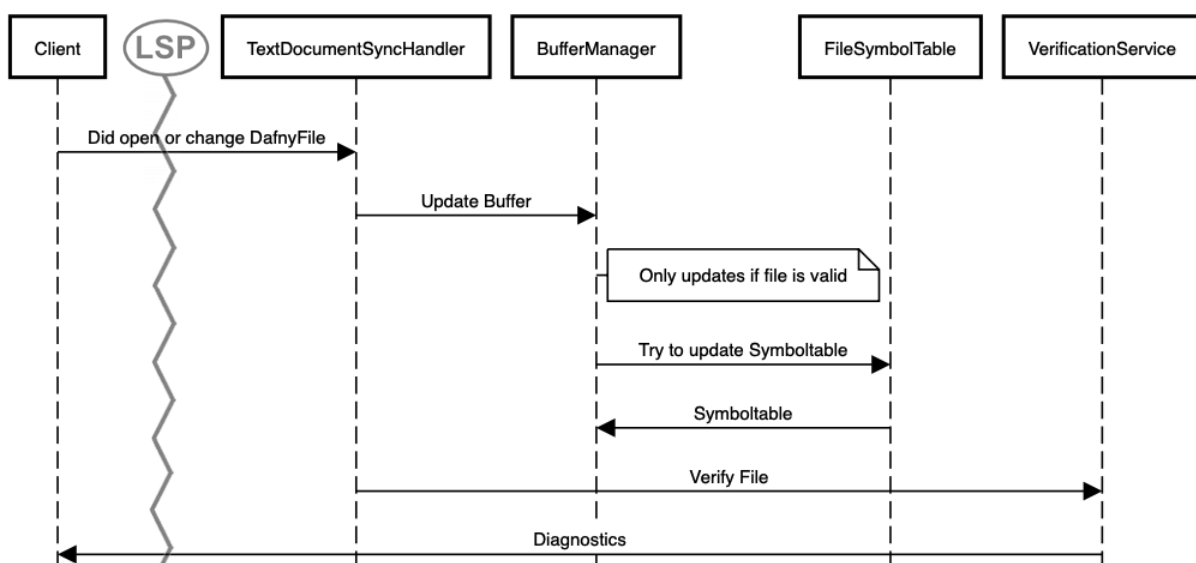


Figure 13 - Sequence Diagram of Building a Symbol Table

As you can see in Figure 13, each time the BufferManager updates a file entry, it tries to generate a new symbol table for that specific file. If successful, the symbol table will be cached. Once a file is invalid (e.g. when the user is starting to type), the symbol table can't be generated. To still provide completion results to the user, the last valid version of a file's symbol table of the buffer can be used. This is illustrated in the Figure 14 below.

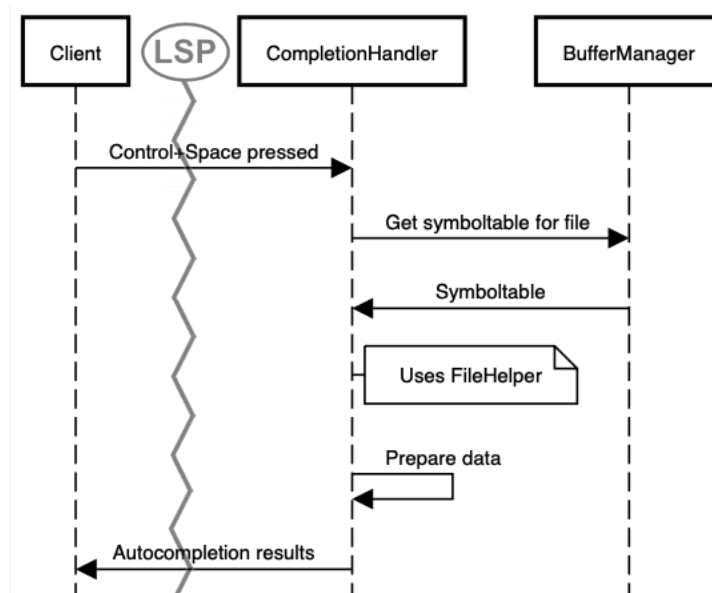


Figure 14 - Sequence Diagram for Autocompletion

6.5.5 Counter Example

The implementation of the counter example feature was not quite as easy, but we still could get inspiration from the previous project, which also offered this feature. The big challenge was to extract the counter example from the way Dafny provided it. Assembling the information to a simpler data structure - in the end just to a string that is displayed, was the main task of this feature.

Similar to chapter 6.5.3 counter example is also initialized by a user command. The client will first off save the document, and then send a request named `counterExample` together with the file URI to the server.

On the other side, the server has a proper handler registered and will forward the request to the counter example service. The service itself will make use of the `DafnyTranslationUnit`, which offers a method `CounterExample`. This method will parse the document, and then apply a server argument `"/mv"`. With this argument, the Dafny pipeline will save a model file which contains the counter examples for each file.

A further class, `CounterModelProvider`, will then read out the model file and extract the relevant information. Extracting all counter models requires a lot of string parsing and conversions, which we could take over from the previous bachelor thesis. Back at the counter example service, the counter models are assembled into a clean list and then returned to the client.

The client will finally display the counter model as a `TextDecoration` element. To be able to dispose and thus hide the counter model, it was necessary to implement a counter model provider at the client side, which keeps track of all displayed text decorations. It is also responsible to create and show text decorations which it receives from the server.

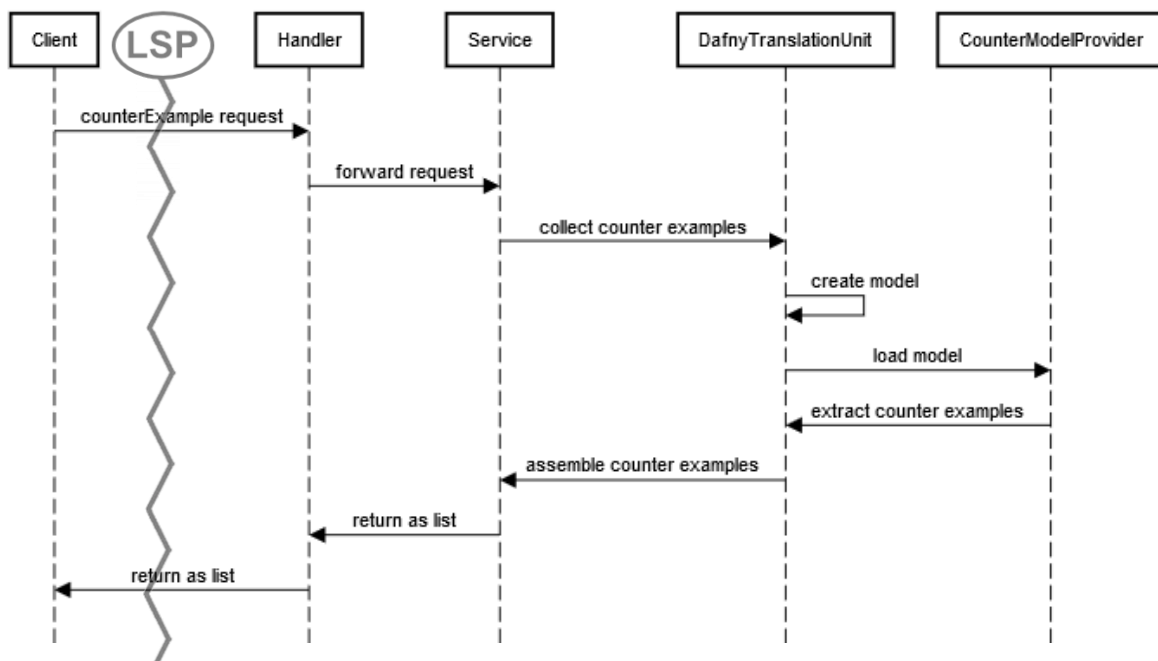


Figure 15 - Sequence Diagram of a Counter Example Request

By simply applying the `"/mv"` argument and running the execution engine, it is necessary to read out the model file which the `"/mv"` argument creates. In this thesis, it was actually meant to bypass such string parsing actions. The reason why we kept the long way around `model.bvd` is, that this way, no changes or deep digging were necessary inside Dafny or Boogie itself. By just reading out the model file, the components can be left as they are, they stay isolated and less coupled, and the counter examples can still be accessed relatively quickly.

6.5.6 Go to Definition

The basic implementation of this feature is kept very simple. It is implemented under the premise that the first match of a symbol name is its definition. If one desires the location of a provided symbol name, the server iterates through the cached symbol table and returns the first match as the definition. The current state of the this feature is still in development. It needs refactoring and test cases.

6.5.7 CodeLens

CodeLens is still under development, too. The current state of the feature shows, how many times a class, function or method is used. However, the user can't click on this information to see previews of the usages.

The implementation is not done in a very graceful way. Using two nested loops, it is checked if two symbols are linked. Whenever this is the case, the counter for the number of references is increased. The current implementation results in a complexity of $O(n^2)$. This could cause performance issues since CodeLens gets triggered on every file change. Thus, the efficiency should be improved by the usage of proper data structures, for example from graph theory.

There is currently another disadvantage. Since the CodeLens feature uses the buffered symbol table, the line information can be outdated, whenever the buffer is not up to date. This happens every time a user is inserting new lines or removes them. The line, on which the reference count should be displayed, is wrong in this case.

This problem is currently also occurring in other features, but with CodeLens it is very noticeable.

To fix this, we planned to add an additional shift information into our file buffer. Whenever the code is invalid, it should store any offsets between the actual and the cached file. This would resolve this issue.

6.6 Testing

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

6.6.1 Unit Tests

Whenever one of our components contains logic, we outsourced the logic into isolated components as far as possible and used interfaces wherever needed. Thus, we could implement isolated unit tests with dependency injection.

In general, one can say that handlers and services mostly do simple method forwarding and they do not contain much logic. More complex logic can be found in “ContentManager” and this code gets heavily tested by unit tests.

Unit tests were only done for the server, not for the client. As the client is mainly responsible for forwarding requests and displaying the results, we abstained from unit testing UI tasks.

6.6.1.1 Simple Unit Tests

The server offered a few methods that could be easily unit tested. This includes mainly helper or library methods, such as creating positional values, for example underlining ranges, or checking if a symbol is within a range. Since these methods had no dependencies at all, testing them was easy. They are mainly found in the FileHelper test class.

As usual, we tested not only successful cases but meaningful inputs from different equivalence classes. Bad cases which are supposed to throw exceptions are tested, too.

6.6.1.2 Unit Tests for Compilation

Since the compilation feature mainly starts `Dafny.exe`, the first attempt for unit testing was to check the last-access-timestamp, which is provided by the operating system. While this worked perfectly fine locally, the tests would not pass during the CI process. Servers, and also many advanced users on their personal computers, often have the feature of saving the most recent access time disabled. This way, they can save a lot of disk writing. This feature has a big performance impact, especially for servers, which handle a lot of disk access.

Compilation was then tested by simply writing a wrapper class for C#'s `Process` to keep track of the process state. This way, we could test if the process had been correctly launched.

6.6.1.3 Unit Tests with Fakes

Many services were not as easy to test, since they depend directly on the `DafnyTranslationUnit`, which itself depends on the Dafny library. Our idea was to create an interface for the `DafnyTranslationUnit` and inject the dependency wherever it is used.

```
public interface IDafnyTranslationUnit
{
    bool Verify();
    List<ErrorInformation> GetErrors();
    List<SymbolTable.SymbolInformation> Symbols();
    List<CounterExampleProvider.CounterExample> CounterExample();
    void DotGraph();
}
```

In our tests, we could simply inject a fake to check if the methods are working correctly. However, achieving this goal was not always as easy as it may sound. For example, Dafny provides a really complex data structure for its counter example results. There are multiple lists nested in each other. Creating a fake result was therefore quite sketchy and not as easy to do. However, with a few assisting methods in the fake class, usage was finally quite simple as the following code sample shows.

```
FakeDafnyTranslationUnitForCounterExamples fake =
    new FakeDafnyTranslationUnitForCounterExamples();

fake.AddCounterExampleToFake(col1, row1, vars1, vals1);
fake.AddCounterExampleToFake(col2, row2, vars2, vals2);

var service = new CounterExampleService(fake);
CounterExampleResults results = service.ProvideCounterExamples().Result;

Assert.AreEqual(2, results.CounterExamples.Count,
    $"Counter Example should only contain 2 counter examples.");

CompareCounterExampleWithExpectation(results.CounterExamples[0], col1, row1, vars1, vals1);
CompareCounterExampleWithExpectation(results.CounterExamples[1], col2, row2, vars2, vals2);
```

Here, a fake instance for the method counter example is created. The fake allows to add counter examples by just calling the proper `addCounterExampleToFake` method. The fake can then be used to create the counter example service, which normally uses a real translation unit instead of the fake. The comparison of the final result is again assisted with the extracted method `CompareCounterExampleWithExpectation`.

6.6.1.4 Tests for Trivial Methods

Another problem was that tests would quickly become trivial and insignificant, because many classes just forward requests. In these situations, we found integration tests more meaningful and focused rather on these instead of strict unit tests.

6.6.2 Integration Tests

The project features two types of integration tests. On the one hand, there are tests checking if our handlers and services work correctly. However, these tests do not inject fake objects but work with real files, existing on the file system and passing those as arguments. Creating those tests was easy as there was nothing special to consider.

On the other hand, we tried to implement integration tests by creating a dedicated client instance with the OmniSharp client interface.[19] The client connects to the server and can then make enquiries, such as asking for a `counterExample`.

While we could set up the basic concept of these tests, they could not be completed. Inside the log, responses are visible, but the proper variables will just stay null. This is possibly due to a bug on OmniSharps side. We contacted them about this but did not get an answer in time. For example, the following request

```
var completions = client.TextDocument.Completions(  
    filePath: aDfyFile,  
    line: 2,  
    column: 5,  
    cancellationToken: cancellationSource.Token  
).Result;
```

will actually produce the response as expected

```
[12:45:29 DBG] Read response body{"jsonrpc":"2.0","id":"2","result":[{"label":"a (Type:  
Method) (Parent:  
_default)","kind":2,"deprecated":false,"preselect":false,"insertTextFormat":0,"textEdit":{"  
range":{"start":{"line":2,"character":5},"end":  
{"line":2,"character":6}},"newText":"a"}}]}
```

but the variable "completions" just stays null. There would be the option to test according to the log output, but this would be extremely tedious, and we decided to await OmniSharp's answer on this issue.[16]

6.6.3 System Tests

System tests were provided within the previous bachelor thesis. They made use of a headless instance of VSCode. Basically, the tests will open some Dafny files and check if proof obligations and errors are reported as expected. Those tests have a huge amount of dependencies. Most notably, a X-Server [20] has to be started within the CI to be able to launch VSCode later on for the test. Understandably, not many of those tests were implemented. During development, a further problem arose - the VSCode instance would require some launch arguments that cannot be set - and we decided together with our supervisors to skip these tests for now.

6.7 Project Automation

As a version control system, we used git. CI and CD was realized with GitLab. Whenever a commit was made, the project was built, tested and evaluated automatically.

Setting up the project automation turned out to be very exhaustive and took over-proportionally long.

Our GitLab CI/CD has four stages as shown in the figure below. Three of them run each time a git push has been received. The first stage, Prebuild, does only run when needed as discussed in chapter 6.7.2 The following subchapters describe what happens in each stage.

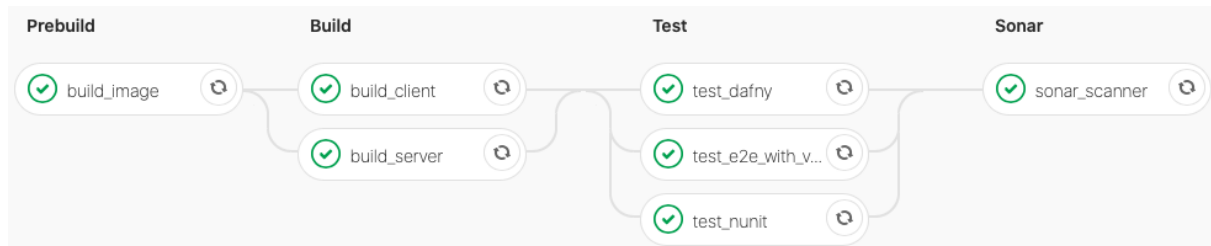


Figure 16 - GitLab CI/CD

6.7.1 Environment

GitLab uses a Docker container in which the build process takes place. Since our project has several dependencies, setting up this environment was not easy at all and caused a lot of problems. The Dafny solution references Microsoft Boogie, which is not realized with a NuGet package, but by given, relative path references. The user must clone Boogie's repository and set up a strict path structure. Boogie itself uses further dependencies, namely the correctness proofer Z3.exe, which has to be in a specific folder, too. All these references had to be set up in Docker, which we could only do thanks to the help of our co-supervisor Fabian Hauser.

6.7.2 Prebuild Stage

The Prebuild stage contains the `build_image` pipeline. It is responsible for the initial setup of the Docker container. This pipe runs only if the Docker or GitLab configuration has changed. You can read more about this in the developer documentation.

The Prebuild stage is responsible to set up Node, Z3, Go and Boogie. The Docker container also needs the Sonar scanner for the later `sonar_scanner` stage.

6.7.3 Build

The Build stage contains two pipelines. `build_server` builds the server using `msbuild` and `nuget restore`. In the second stage `build_client`, which runs in parallel, the client gets prepared. We use the command `npm install` to download all needed Node packages. Node modules are cached for later stages.

After all Node modules are ready, TypeScript files are built into the `VSCodePlugin/out` folder as you can see in the `.gitlab-ci.yml` file.

Because later stages will need those output files, we save them into an artifact.

`msbuild` and `nuget` itself were provided by the Docker container and did not cause a lot of trouble. However, keeping the folder structure correct and providing all requirements like `Z3.exe` was quite tricky.

The client on the other side could just be built using `npm`, which was also provided within the Docker container. This process caused no trouble at all and was quite straight forward.

6.7.4 Tests

The test stages include three parallel pipelines.

The `test_dafny` pipeline runs the previously written Dafny tests. We included those tests into our project to be sure we do not change the old Dafny console server by accident.

In `test_nunit` we run our own unit and integration tests for the language server part.

We also have the pipeline “`test_e2e_with_vscode_instance`”, that runs a headless Visual Studio Code instance to run automated end to end tests. Unfortunately, this is not working yet.

6.7.4.1 Client

To start the client tests, one has to install all Node dependencies using `npm install`, then build the typescript code with `npm run vscode:compile-typescript`, and finally the tests can be run with `npm run tests`.

These are exactly the commands the CI pipeline executes. To read more about this, refer to the developer documentation.

6.7.4.2 End to End

For the available end to end tests, an X-Server had to be started to provide the proper UI infrastructure, and a headless VSCode instance is used. As mentioned earlier, those tests could not be run at the end of the project due to a launch argument that could not be set.

6.7.4.3 Server

It seemed quite reasonable to use `mstest` to write unit and integration tests in Visual Studio. However, as it turned out, it was not that easy to install `mstest`, the test runner for default .NET Framework tests, inside the CI environment. `mstest` uses quite a lot of dependencies. One has more or less to install a full instance of Visual Studio to get access to `mstest`.

As an alternative, `nUnit` was considered. However, `nUnit` is only available for .NET Core projects by default. Referring between .NET Core and .NET Framework is something that does not work, thus we had to come up with a little hack and manually overwrite the project type of our `nUnit` test projects from .NET Core to .NET Framework. Surprisingly, this worked without any issues.

The `nUnit` test runner was neither available from scratch and had to be installed manually. At first, it was not clear what the easiest way of doing that is. However, in the end we found the test runner as a NuGet package. Therefore, we could simply install it by installing the corresponding NuGet package.

Finally, we had a runner and test projects that were able to refer the other projects without any issues.

6.7.5 Sonar Scanner

Finally yet importantly, we execute the Sonar scanner and publish the information to SonarCloud. At this point, this is only working for the client part.

According to the community forum, the C# code base can only be analyzed with the scanner for MSBuild and needs two different projects on SonarCloud. The codebase has to be analyzed twice (one for the TypeScript codebase, one for C#) in order to get proper analysis results. [21]

For reasons of time, we have not yet made this separation. In order to ensure the code quality nevertheless, we have taken other measures. More details about this are stated in chapter 9.4 Code Quality Aspects and Metrics.

7. Results

This chapter presents all features that were implemented. The chapter is kept less technical and gives an overview of the achieved scope of the project.

7.1 Syntax Highlighting

As described in chapter 6.5.1 the syntax highlighting is realized through a given Dafny grammar file. The following screenshot shows how syntax highlighting looks inside Visual Studio Code.

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

Figure 17 - Syntax Highlighting

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

7.2 Verification

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

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

This is the postcondition that might not hold. The error: A postcondition might not hold on this return path.

Peek Problem No quick fixes available

Figure 18 - Postcondition Violation

The logical verification is permanently active. The user just has to type Dafny code and it will be verified. The current version only supports logical errors, not syntax errors. Whenever an assertion or postcondition violation appears, the code block is underlined in red. The actual postcondition is underlined in yellow. The user can hover over the error to get additional information about the problem.

7.3 Compile

The user has two options to initiate compilation: He can just compile his software or he can additionally run it on a console. The commands are available as hotkeys, in the context menu or via the VSCode command bar. The client will save the document, send its URI, and the server will report back if compilation was successful. If any errors are present in the code, they are reported as a window notification. This includes syntax errors. If the compiled program can be run, a PowerShell instance is started directly inside Visual Studio Code. As an example, our code from Figure 18 would produce the following two notifications:

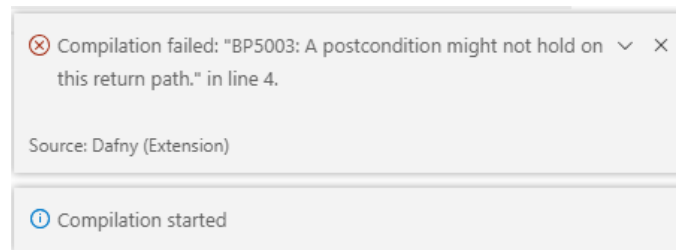


Figure 19 - User Notification on Compile Errors

7.4 Counter Example

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

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

Figure 20 - Demonstration of Counter Examples

The example in Figure 20 demonstrates a counter example. The precondition is commented. Without this requirement, the postcondition can be violated with $y = 0$.

7.5 Auto Completion for Identifiers

Whenever a valid interim result of a Dafny file is sent to the server, the file symbol table is created. This table is then stored in the buffer as discussed in chapter 6.5.4 Autocompletion for Identifiers. As an example, if one would have a Dafny code snippet like below, the best and only match would be method `m`. This is exactly what is suggested for the user as you can see.

```
class C {
  constructor() { }
  method m() {
    // do something
  }
}
method Main() {
  var acc := new C();
  acc.
```

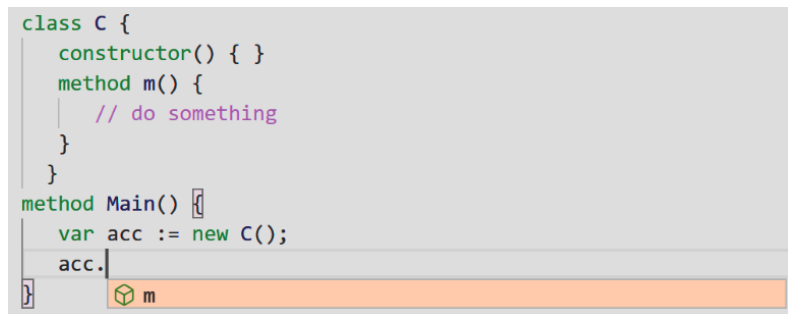


Figure 21 - Auto Completion for Instance of Class C

In addition to the previous plugin, our new version is now also able to provide general completion suggestions if one is not typing a word. In this case, just all available symbols are proposed. This is illustrated below. Please also note that every symbol type (i.e. methods, classes and variables) has its own icon.

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

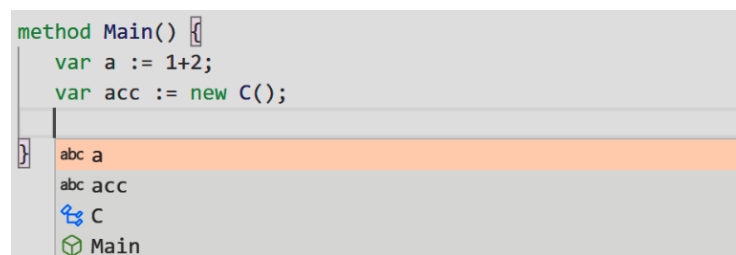


Figure 22 - General Suggestions with Symbol Types

Besides the general release build for production, there is also a developer build available. The behavior is slightly different. Once a debug version of the server is built, additional information like the type and the parent of the symbol become visible. This can be very helpful for future feature development.

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

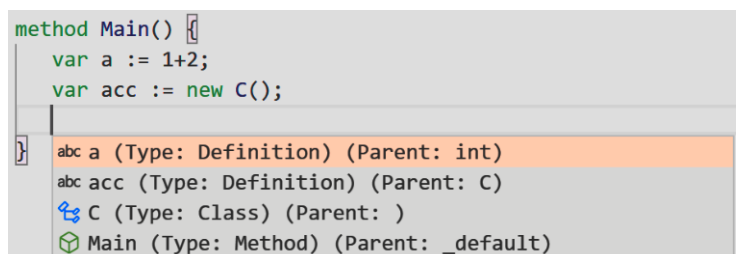


Figure 23 - Auto Completion with Debug as Server Mode

7.6 Go to Definition

The usage of the “Go to Definition” feature is pretty easy. A user can simply use the context menu or hit F12 to go to the definition of the used symbol. In the following figure, you can see how a user would like to jump to the definition of the method `m`.

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

0 reference(s) to Main
method Main() {
  var acc := new C();
  acc.m();
}
```

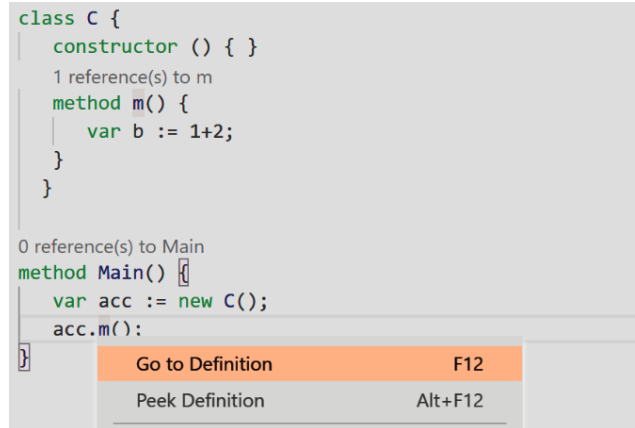


Figure 24 - Go to Definition via Context Menu

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

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

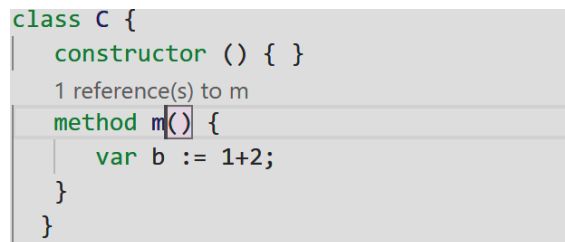


Figure 25 - Cursor Position after go to Definition

7.7 CodeLens

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

```
2 reference(s) to C
class C {
|   constructor () { }
|   1 reference(s) to m
|   method m() {
|       // do something
|   }
}

0 reference(s) to Main
method Main() {
|   var acc1 := new C();
|   var acc2 := new C();
|   acc2.m();
}
```

Figure 26 - CodeLens Example

Unfortunately, clicking the grey references is not yet supported. This should open a box in which the corresponding uses are shown, as it was the case in the original plugin.

8. Conclusion

In this chapter, our work is reflected. We will have a critical look at the implemented features and how they can be improved. Features that could not be implemented will also be mentioned. Finally, we will discuss how the plugin can be extended in the future.

8.1 Reflection per Feature

In this chapter, we will have a critical look at our features. Many of them are not realized in a robust way or they come with certain flaws. This chapter should make clear that we are well aware of those flaws and that we may revisit our implementation to improve them.

8.1.1 Verification

This feature has two rather major flaws. The first one is that Boogies verification is not responding all possible errors that can occur. It actually only checks for logical correctness, but not for other syntax errors. If syntax errors exist, Boogie won't even be involved and the parsing process fails. We couldn't collect these error messages in time, since they require deeper digging into the Dafny pipeline.

Currently, syntax errors, such as a forgotten semicolon or the usage of an undeclared variable won't cause underlines. However, if you try to compile the code, a proper error message is shown.

d: > Eigene Dokumente > VisualStudio > SA > dafny-server-redesign > anExmapleDafnyFile.dfy

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

⊗ Compilation failed: "semi expected" in line 7.

ⓘ Compilation started

Figure 27 - Flaw of the Error Highlighting

As you can see in the figure above, the user forgot to write a semicolon at the end of line 6 (the compiler reports the error in line 7, because it expected a semicolon before the statement “assert” in line 7). Although there is an assertion error in line 7 as well as a postcondition violation in line 3, none of them are reported. Since the document is invalid and cannot be parsed, the verification service is not involved. Compilation is currently the only way to get notified about the error.

The second flaw of this feature is that Boogie only reports a tiny range of the source code as erroneous, such as a ‘==’ or a ‘{’. Since this is very hard to see, we decided to underline not only these one or two characters, but the whole line wherever an error occurs. However, if a line consists of a single character, such as a ‘{’, this is obviously hard to see and should be done better.

8.1.2 Compile

The compilation feature works actually quite reliable. Due to its simplicity it is quite robust. It starts Dafny .exe with the opened file as an argument. However, the feature displays quite a lot of notifications. If the compilation task is not taking very long, the user may get up to three notifications in a short amount of time. A possible update could limit the notifications in such a way that they are only displayed if compilation takes long enough.

8.1.3 Auto Completion for Identifiers

The feature for auto completion already works pretty well in a basic scenario. It could still be improved to increase the usability even further.

There are two ways how suggestions can be made smarter. A simple and independent addition would be to check for keywords. For example, if there is a “new” keyword in front of the cursor, only classes could be suggested to the user.

Another extension would be the adjustment of the syntax tree parsing in deeper Dafny layers. If more information was available, better suggestions could be made. For example, if each symbol also contained its scope, the suggestions could be filtered efficiently by only showing symbols which are available at the cursor's position.

Another approach for more usability would be to improve code insertion. For example, if a method has been selected for autocompletion, no brackets are included after the method name. Similarly, a new keyword could automatically be inserted in front of classes if there is none yet. Also, placeholders for arguments could be generated for constructors and methods. A user would only have to press TAB to swap between the argument placeholders. This is for example known from other IDEs like XCode. The following figure illustrates this feature.

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

Figure 28 - Auto Completion in XCode

8.1.4 Counter Example

As you can see in Figure 20 - Demonstration of Counter Examples, the counter example provider reports the counter examples in a little bit a harsh format. There are more brackets than necessary. Also, in line 5, the variable less is already reported, although its value is not yet known. Displaying the counter examples could definitely be made more user friendly.

However, a nice feature of the design element that was used to display the counter example is that it moves dynamically as the user continues to type. That means, if he inserts blank lines on top, the counter examples will also move downwards and stick to the corresponding code line.

Another flaw of this feature is that as the user corrects his code, the counter example will not automatically adjust or vanish if the postcondition violation has been resolved. The counter example is fixed and has either to be generated again or to be hidden explicitly by the user. A simple solution for this would have been to just hide the counter example whenever the user starts to type again. However, the user may actually want to keep track of the counter example. This is why we didn't implement it this way.

8.1.5 Go to Definition

As mentioned in chapter 6.5.6 Go to Definition, this feature is still in an early stage of development.

Since the definition for a symbol is just defined as the first occurrence in a file, the feature will not work over multiple Dafny files. Support across multiple dependent Dafny files is still a general problem. Although conceptually the BufferManager was designed for this, there is no specific implemented support in a feature for this dependency logic.

The way of evaluating the target symbol should be increased in smartness for better usability. The feature could be made more user friendly if it was less dependent of the cursor position. For example, if no keyword can be found on the right side of the cursor, the server could also try to find a keyword on the left side. This would provide more robustness and increase usability.

8.1.6 CodeLens

The current state of this feature is still in progress as mentioned in chapter 6.5.7 Indeed, there are still several open construction sites to complete this feature. Please read the mentioned chapter “CodeLens” for more details about the performance, the buffer issue and the actual CodeLens popup window, which is missing in the current version.

8.2 Extension Points of the Previous Thesis

In the previous bachelor thesis, a variety of features was named for possible extension points. This included debugging, various refactorings and contract generation.[2] Unfortunately, we were unable to spend time on any of these features, because we were busy enough working on existing use cases. Thus, these features will still be possible extensions for the upcoming bachelor thesis.

8.3 Short-Falling Features of the Previous Thesis

Some features that were implemented in the previous bachelor thesis could not be taken over to the new plugin. Unfortunately, we spent too much time on CI issues and couldn't implement as many features as we wanted. This chapter gives a brief overview of those.

8.3.1 Installation, Setup, Marketplace Integration

After a short consultation with our supervisors, we decided not to spend any time on the installation process. This is because we will continue to develop the plugin. A release would not make much sense at this point, because some features still need improvement. Thus, we will focus on an easy installation towards the end of the upcoming bachelor thesis.

8.3.2 Platform Independence

Although we tried to keep this requirement in focus, it was really hard to strictly follow this guideline. The previous bachelor thesis offered some nice concepts, like global variables storing if Mono is to be used, but we kept our focus on a single platform during development. However, the infrastructure to make the plugin runnable under Linux and macOS is present and before a release, it is realistic to re-guarantee platform independence without big effort.

8.3.3 Features

Since we were very busy working on the problems which continuous integration caused, we couldn't implement all features of the previous bachelor thesis. We also focused on refactoring towards the end instead of implementing additional features in low quality. Thus, we could not implement renaming, null and bound checks, flow graphs and invariant guards. In the subsequent bachelor thesis, we plan to implement them.

8.4 CI Reflection

If there is one thing which slowed this thesis down, it was continuous integration. As stated in chapter 6.7 Project Automation, it caused a lot of trouble and extra work. Since there was no preconfigured testing environment, just setting up the infrastructure for basic automated unit tests was quite strenuous. Additionally, each setup had to be done once for the client and once for the server with completely different prerequisites. This made CI a very hard task for us as this is not our main strength.

A legit question is, if the choice of GitLab was reasoned well enough. GitLab bases on Linux. This was welcomed for the client side, but the server could have profited way better from a Windows environment. The installation of `msbuild` and `mstest` would have been much simpler. With GitLab, we had to bypass a lot of these issues, for example by using the nUnit test runner as a NuGet package installation, or by relatively complex Docker configurations.

8.5 Planned Extensions in the Bachelor Thesis

Aside the improvement of the current features, there are several ideas for additional use cases. One of them is debugging. However, debugging is very complex and may be a bit too difficult for the limited time range in a bachelor thesis. There are also several refactoring options, like rename variable, extract method, and such. A nice feature would also be the automated generation of contracts, which constructs pre- and postconditions by itself. We could also implement array boundary checks and null checks.

Another point of extension is to provide support for more IDEs, not only Visual Studio Code. This would only require writing a separate client part. The server stays untouched.

9. Project Management

This chapter reflects organizational aspects. We will compare relevant propositions from the project plan to the actual state. First, we talk about time management. Then, the scope is analyzed. Finally, some code aspects will be discussed.

9.1 Meetings

The weekly meetings with our supervisors Thomas Corbat and Fabian Hauser took place each Wednesday at 1pm. They were moved to Friday afternoon for the last three weeks due to appointment conflicts. Internal meetings with only Thomas Kistler and Marcel Hess were held on a regular basis on Friday afternoon.

All meetings were very informative and could be completed in a reasonable time frame. They were generally very productive. The overall time spent for discussion and information exchange was relatively small compared to other activities. Therefore, more time was available for production.

9.2 Time Management

The project had a time range of 14 weeks. Each of the two students had to spend about 16.5 hours per week on the project. Time tracking was done in Redmine. On average, we were able to keep ourselves within the timeframe as you can see in Figure 29. Both team members worked about 235 hours in total.

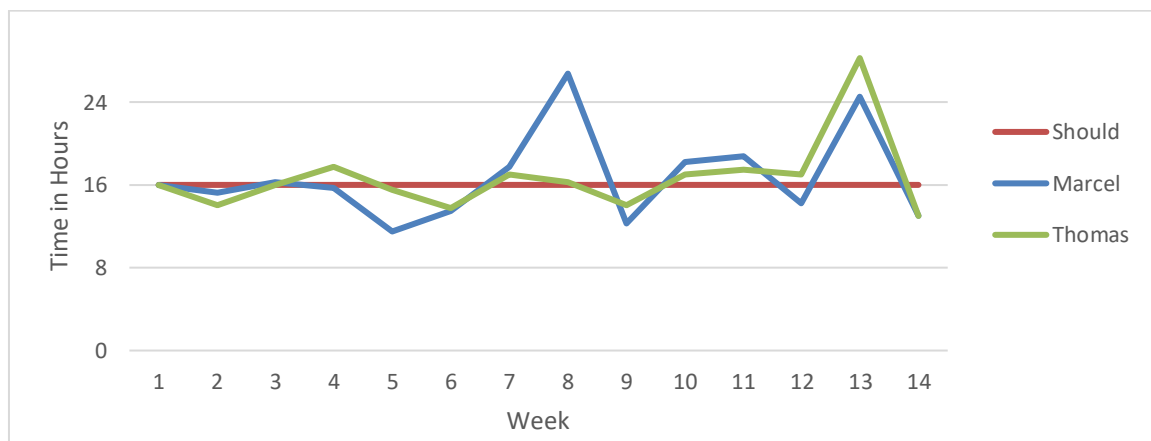


Figure 29 - Time Tracking

In the first few weeks, we were struggling getting started and getting the whole project to run. Once we were settled in, we were able to become more productive and spend more time on feature development.

In the following diagram, you can see the invested time by activity. The largest part was development, followed by documentation and meetings. We rate this basic time split as well taken.

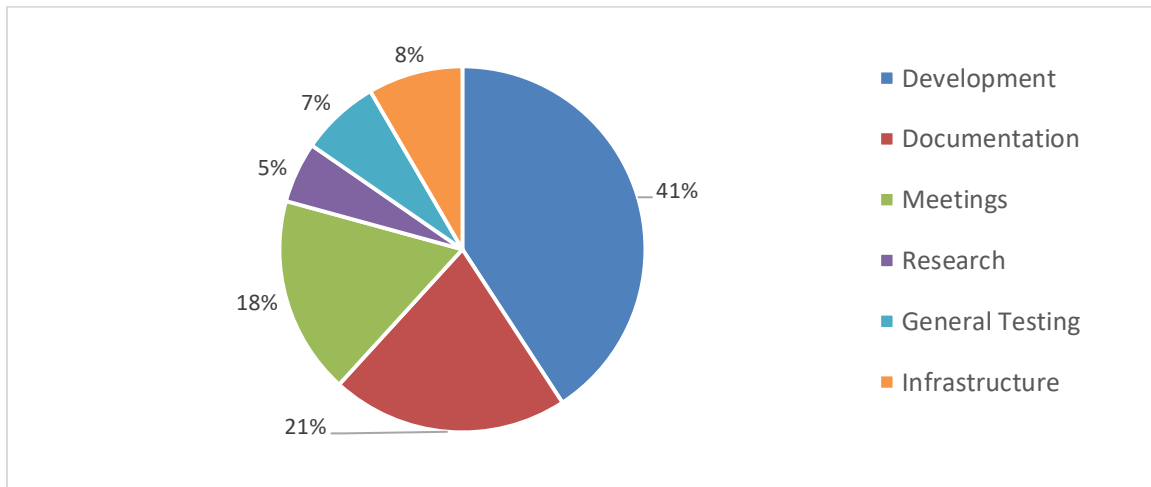


Figure 30 - Time by Activity

The most time-consuming development activity was CD/CI. We had a lot of trouble getting all tests to run in our CI and we struggled until the end with dependencies. This took us much more time than expected. Other major development activities can also be seen in the following figure.

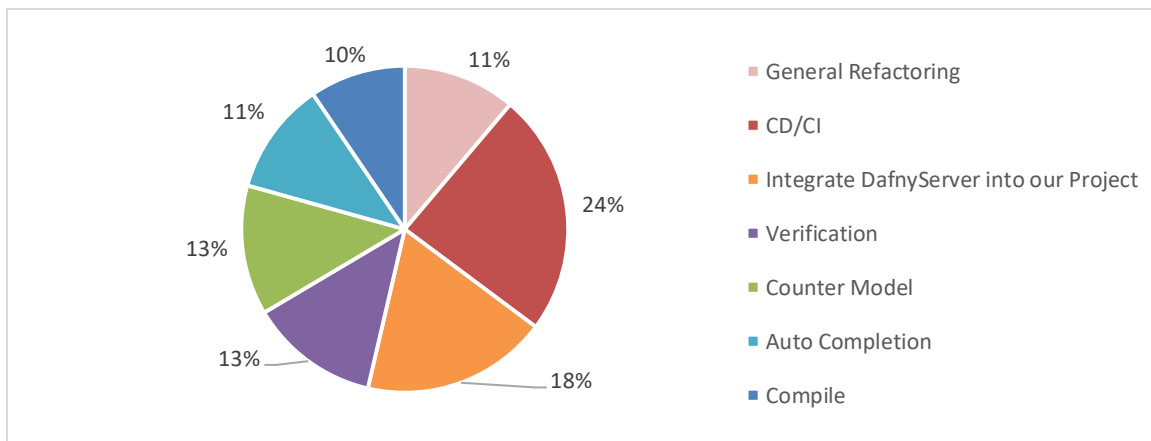


Figure 31 - Most Time-Consuming Development Activities

9.2.1 Project Management

As planned in the project plan, we used a mix between unified process and SCRUM. The project was broadly split into an inception, elaboration, construction and a transition phase. Each of the periods was split into sprints lasting two weeks. At the end of every sprint, a milestone was defined. At the start of the project, it was not easy to plan our milestones. We did not know which features are easy to start with and which are rather difficult to implement. Thus, we kept the milestones very generic with the naming “First Features” and “Further Features”. The planned milestones are shown in the following Figure 32.

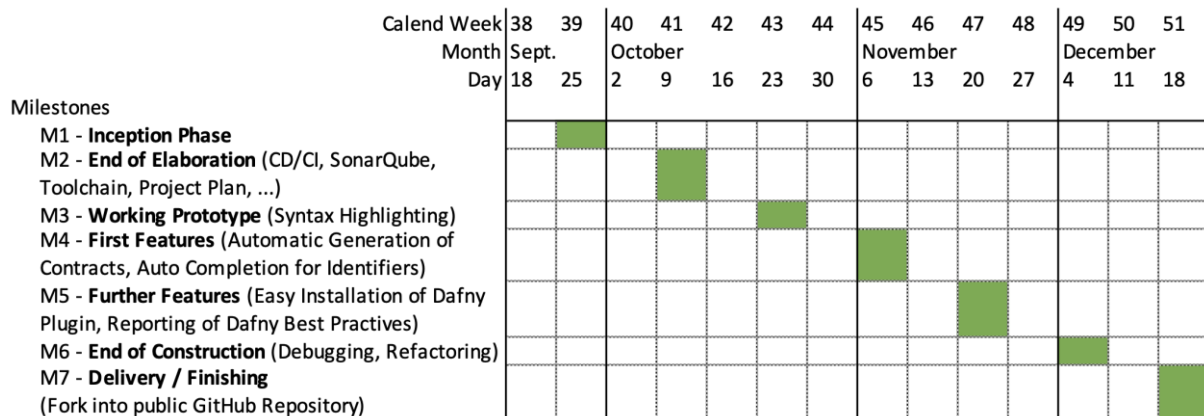


Figure 32 - Planned Milestones

In general, we were able to stick to this plan pretty well. However, the features which we implemented varied a lot from the initial plan. CD/CI has kept us busy through almost the very end of the project as you will see in Figure 33 - Actual Milestones. It should have been completed after the elaboration phase.

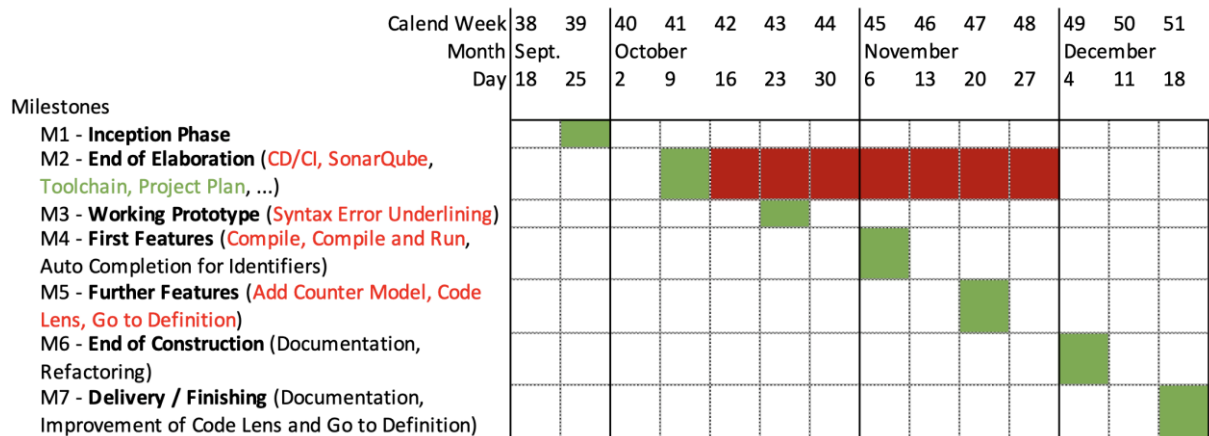


Figure 33 - Actual Milestones

The last milestone “End of Construction” was supposed to be a reserve time slot. We could use it well for refactoring, since we decided to put quality over quantity. Therefore, the features “Syntax Highlighting” and “Code Lens” were only implemented in a very basic way. At the end, we had a little time left to make small enhancements, but the features still need improvement.

9.3 Scope

In the project plan, we stated that we wanted to implement the same use cases the previous thesis offered. Those included:

- Easy installation of the plugin
- Syntax highlighting
- Reporting of Dafny best practices violations
- Automatic generation of contracts
- Auto completion for identifiers

During development, these targeted features were changed quite heavily. Figure 33 shows all deferred features marked in red. In the beginning of the project, we did not know exactly which features are suited to start with. As we got familiar with the used technologies, we could make more precise time cost estimates. As stated in chapter 8.3.1 the automated plugin installation was postponed. Syntax highlighting worked out of the box. Best practice violations and contract generation have also been postponed to the bachelor thesis. Auto completion could be implemented. However, we could also implement compilation, code verification, counter examples, go to definition and a basic version of code lens. These were not even listed in the original project plan.

For a further description in brief and fully dressed format of these use cases, we refer to the bachelor thesis of Markus Schaden and Rafael Krucker.[2]

9.4 Code Quality Aspects and Metrics

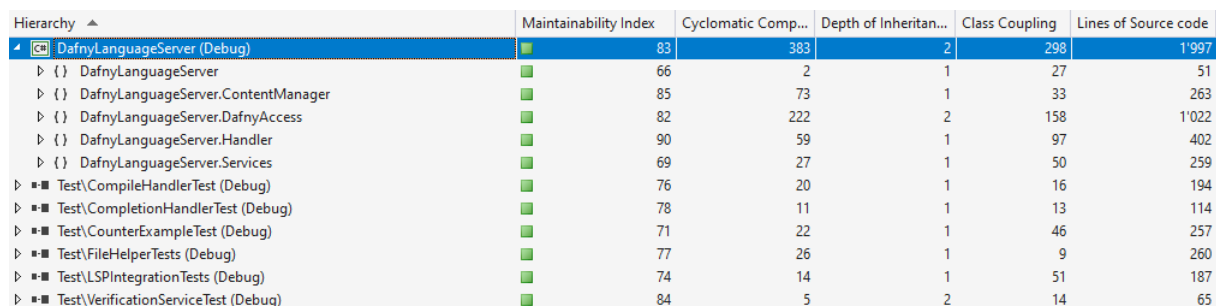
This chapter gives a brief overview of the most important enumerations about the project.

9.4.1 Code Reviews

To double check the code, we used GitLab's merge requests to do simple code reviews. Usually, one of the students coded a feature. Once finished, he created a new merge request. The second student could then study the code and make suggestions for improvements before the merge took place. There were also several code reviews head to head to improve the code quality in general.

9.4.2 Backend

This subchapter refers to the backend. In the following figure, one can see the lines of code and some other metrics, that will be covered in the following sections.



Hierarchy	Maintainability Index	Cyclomatic Comp...	Depth of Inheritan...	Class Coupling	Lines of Source code
DafnyLanguageServer (Debug)	83	383	2	298	1'997
DafnyLanguageServer	66	2	1	27	51
DafnyLanguageServer.ContentManager	85	73	1	33	263
DafnyLanguageServer.DafnyAccess	82	222	2	158	1'022
DafnyLanguageServer.Handler	90	59	1	97	402
DafnyLanguageServer.Services	69	27	1	50	259
Test\CompileHandlerTest (Debug)	76	20	1	16	194
Test\CompletionHandlerTest (Debug)	78	11	1	13	114
Test\CounterExampleTest (Debug)	71	22	1	46	257
Test\FileHelperTests (Debug)	77	26	1	9	260
Test\LSPIntegrationTests (Debug)	74	14	1	51	187
Test\VerificationServiceTest (Debug)	84	5	2	14	65

Figure 34 - Server Metrics

9.4.2.1 Lines of Code

The backend consists of about 2000 lines of code (LOC). An additional 1000 lines of code were written for tests. However, a lot of code from the package `DafnyAccess` was taken over from the previous bachelor thesis. Not all of the 1000 LOC in this package were written by ourselves. We guess that we wrote about 1500 LOC. According to the module “Software Engineering”, Prof. Dr. D. Keller taught that each developer can deliver about 500 LOC per month. We were two people, worked for 2.5 months at 40%, and thus are well above this rule of thumb.

9.4.2.2 Other Metrics

The maintainability index is a prediction from Visual Studio that states how easy the code is to remedy. It is generally around 80%. The cyclomatic complexity in the second column is very low for all packages, except for `DafnyAccess`. The very large file `SymbolTable.cs` contains a lot of nested conditional and loop statements, which causes this high value. One task for the bachelor thesis could therefore be to refactor this class. The class coupling is rather high in this file, too, but low for most other classes and packages.

9.4.3 Frontend

The client consists of about another 1000 LOC, but most of them were taken over by the previous thesis. Many of them are also just string constants and such. A bunch of test files and scripts are also present in the frontend. The summary is shown in the following figure.

language	files	code	comment	blank	total
JSON	6	1,560	0	5	1,565
TypeScript	20	916	69	182	1,167
Dafny	15	190	0	29	219
Markdown	3	163	0	76	239
Ignore	1	10	0	0	10
Batch	1	1	0	0	1

Figure 35 - Client LOC

9.4.3.1 SonarQube

The SonarQube report for the frontend in SonarCloud is pretty satisfying.[22] There are some abnormalities as you can see in Figure 36. They are easy to justify though.

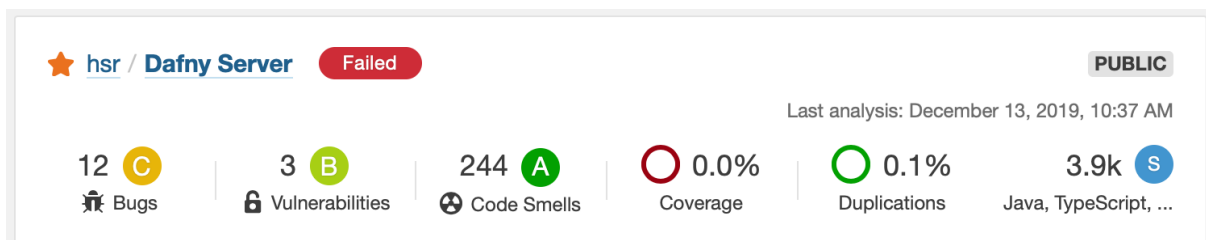


Figure 36 - SonarCloud Report for the Client

There is no longer any test coverage since all unit test cases were removed together with the rest of the language server part that used to exist in TypeScript.

The discovered bugs, vulnerabilities and code smells result from analyzed Java files that were found in the Dafny project. As soon as we split the client from the server for SonarQube as mentioned in chapter 6.7.5 Sonar Scanner, those negative numbers should disappear.

9.4.4 Test Coverage

Test coverage varies a lot by package. As stated in chapter 5.3.2 Handlers, we separated handlers and services. The services could therefore be very well tested as you can see in Figure 37. They come with a very high test coverage.

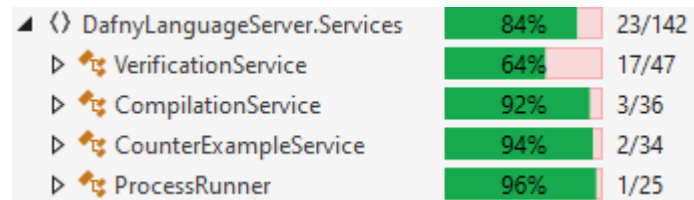


Figure 37 - Test Coverage for Package Services

The package ContentManager, which has no dependencies at all, reached even a coverage of almost 100% as you can see in Figure 38. The tests for this class are also as meaningful as it gets.

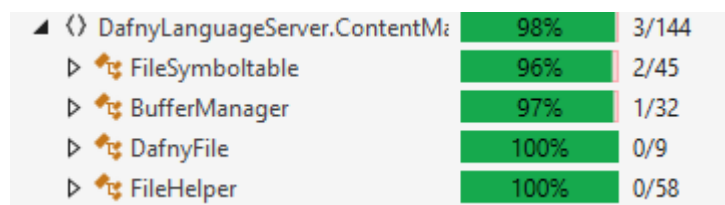


Figure 38 - Test Coverage for Package ContentManager

The DafnyAccess package has a lower test coverage as you can see in Figure 39, since we took over a lot of code from the previous bachelor thesis. Because we are now using this code ourselves, we would actually have to test it. But the classes have heavy dependencies on Boogie, for which we couldn't achieve a proper level of decoupling in time.

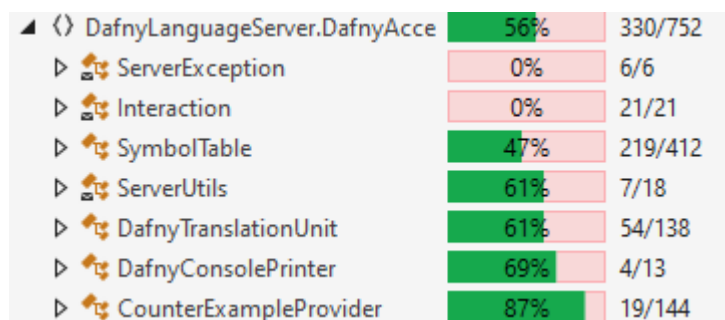


Figure 39 - Test Coverage for Package DafnyAccess

As reasoned in chapter 6.6.1.4 Tests for Trivial Methods, the handlers, which mainly just forward requests, were not subject to tests. This also applies for the Main method. They are getting covered with end to end tests.

With the `Handlers` package excluded, we reached a coverage of about 66%. This is a bit lower than the aspired 80%. But most of this is due to the very large `SymbolTable.cs` file, which contains about 200 lines of untested code as you can see in Figure 40.

Symbol	Coverage (%)	Uncovered
▲ Total	66%	356/1038
▲ DafnyLanguageServer	66%	356/1038
▲ () DafnyLanguageServer.DafnyAcce	56%	330/752
▶ ServerException	0%	6/6
▶ Interaction	0%	21/21
▶ SymbolTable	47%	219/412
▶ ServerUtils	61%	7/18
▶ DafnyTranslationUnit	61%	54/138
▶ DafnyConsolePrinter	69%	4/13
▶ CounterExampleProvider	87%	19/144
▲ () DafnyLanguageServer.Services	84%	23/142
▶ VerificationService	64%	17/47
▶ CompilationService	92%	3/36
▶ CounterExampleService	94%	2/34
▶ ProcessRunner	96%	1/25
▲ () DafnyLanguageServer.ContentM:	98%	3/144
▶ FileSymboltable	96%	2/45
▶ BufferManager	97%	1/32
▶ DafnyFile	100%	0/9
▶ FileHelper	100%	0/58

Figure 40 - Total Coverage (without Handlers)

9.4.5 Commit Activities

Figure 41 shows the amount of commits per day. There are a few peaks. These are mostly reasoned by commits to test new CI configurations. They often just contain single changes to the `.yaml` configuration file. One could use a local Docker container to reduce the commit count, but it was often easier to just push changes to GitLab and test it right there.

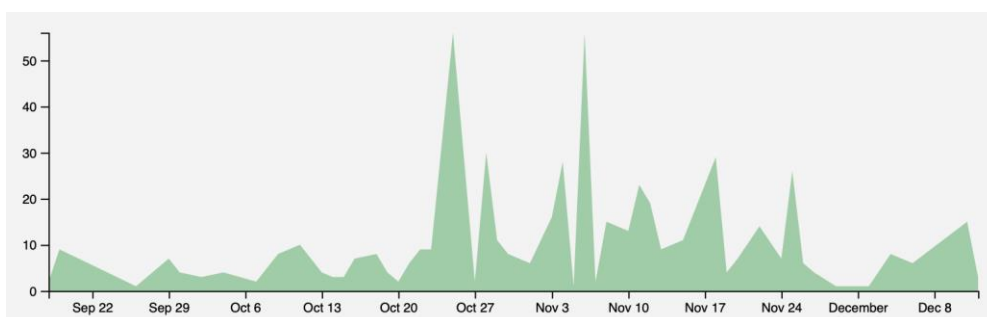


Figure 41 - Commits to Master, Excluding Merge Commits

Mostly, we worked on Mondays, Wednesdays and Fridays for this thesis. This can be well seen in Figure 42.

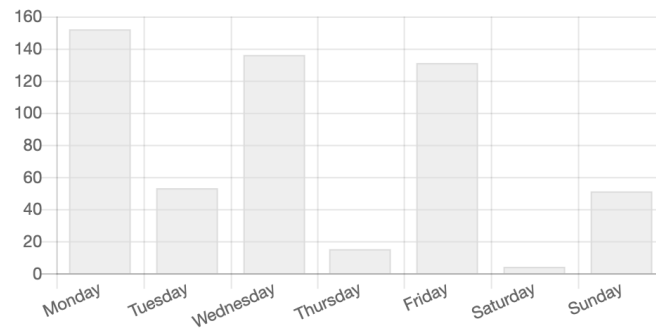


Figure 42 - Commits per Weekday

Once our CI ran stable – unfortunately this was only the case for the last month – we endeavored to push only good code quality to GitLab that would pass the CI process. This is well visible in Figure 43. Successful pipelines are marked in green, failed ones in grey.

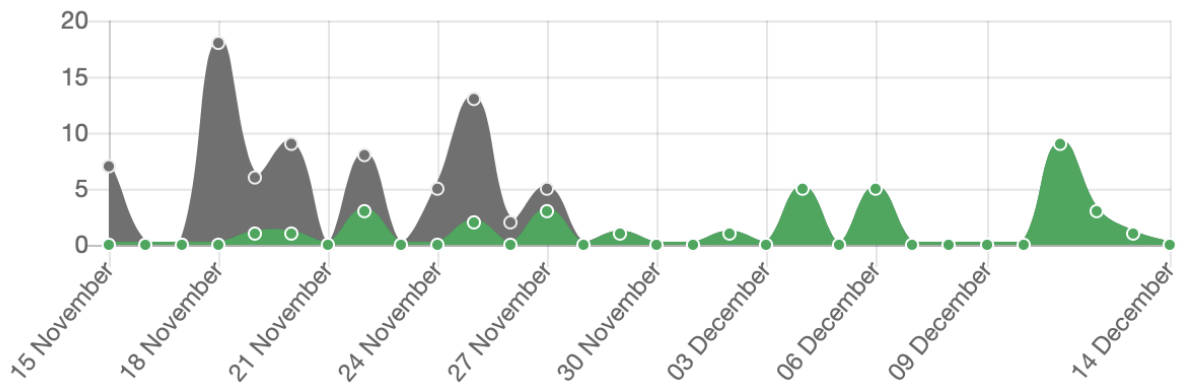


Figure 43 - Pipeline Success Rate

9.5 Infrastructure

Contrary to our original plan, we did not use the provided Windows server at all. Since we switched from .NET Core to .NET Framework, a Windows VM was used instead of macOS. Other than that, we were in line with the project plan. We could use the right IDE's, use the planned tools for code quality and work with GitLab for the CI process.

10. Glossary

10.1 Acronyms

LSP	Language Server Protocol A protocol for the communication between an IDE and a language server
VSCoDe	Visual Studio Code Text editor
IDE	Integrated Developer Environment A text editor for programmers with a variety of additional features, such as syntax highlighting or the option to compile the code
AST	Abstract Syntax Tree Structured internal format for source code
CI	Continuous integration Automatic compilation, execution of tests and quality measurements upon code changes
CD	Continuous deployment Automatic distribution of the product
LOC	Lines of Code Common measure for project size
VM	Virtual Machine Running a virtual computer inside your computer, for example Windows on macOS
JSON	JavaScript Object Notation Specification to serialize data in plain text
JSON-RPC	JavaScript Object Notation Remote Procedure Call Protocol to exchange data using JSON
UI	User Interface Surface of a program which the user can interact with
URI	Uniform Resource Identifier String to identify a resource, such as a file or a webserver

10.2 Technical Terms

X-Server	Toolkit to create graphical user interfaces and interact with them.
Headless	Instance of a program without a user interface.
OmniSharp	Organization that programmed the C# implementation of the LSP.
NuGet	A library with software-packages for C#.
Unified Process	Method for software project management.
SCRUM	Method for software project management.
exe	Executable program file for Windows.

11. References

- [1] "Dafny," *Wikipedia*, 14-Nov-2019. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=Dafny&oldid=926221042>. [Accessed: 16-Dec-2019].
- [2] Krucker, Rafael and Schaden, Markus, "Visual Studio Code Integration for the Dafny Language and Program Verifier," HSR Hochschule für Technik Rapperswil, 2017.
- [3] "HSR Correctness Lab." [Online]. Available: <https://www.correctness-lab.ch/>. [Accessed: 14-Dec-2019].
- [4] "DafnyVSCode/Dafny-VSCode," 20-Nov-2019. [Online]. Available: <https://github.com/DafnyVSCode/Dafny-VSCode>. [Accessed: 05-Dec-2019].
- [5] "Your First Extension." [Online]. Available: <https://code.visualstudio.com/api/get-started/your-first-extension>. [Accessed: 05-Dec-2019].
- [6] "Extension Guides." [Online]. Available: <https://code.visualstudio.com/api/extension-guides/overview>. [Accessed: 05-Dec-2019].
- [7] "Extension API." [Online]. Available: <https://code.visualstudio.com/api/index>. [Accessed: 14-Oct-2019].
- [8] "DafnyVSCode regexRessources.ts," *GitHub*. [Online]. Available: <https://github.com/DafnyVSCode/Dafny-VSCode/blob/develop/server/src/strings/regexRessources.ts>. [Accessed: 15-Dec-2019].
- [9] "Test/dafny0/Char.dfy.expect," *GitLab*. [Online]. Available: <https://gitlab.dev.ifs.hsr.ch/dafny-lang/dafny/blob/master/Test/dafny0/Char.dfy.expect>. [Accessed: 15-Dec-2019].
- [10] "Language Server Protocol," *Wikipedia*, 02-Dec-2019. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Language_Server_Protocol&oldid=928869971. [Accessed: 05-Dec-2019].
- [11] "Language Server Extension Guide." [Online]. Available: <https://code.visualstudio.com/api/language-extensions/language-server-extension-guide>. [Accessed: 15-Dec-2019].
- [12] "LSP Specification." [Online]. Available: <https://microsoft.github.io/language-server-protocol/specifications/specification-3-14/>. [Accessed: 28-Oct-2019].
- [13] "Langserver.org." [Online]. Available: <https://langserver.org/>. [Accessed: 05-Dec-2019].
- [14] "Martin Björkström - Creating a language server using .NET." [Online]. Available: <http://martinbjorkstrom.com/posts/2018-11-29-creating-a-language-server>. [Accessed: 14-Oct-2019].
- [15] "OmniSharp/csharp-language-server-protocol," *GitHub*. [Online]. Available: <https://github.com/OmniSharp/csharp-language-server-protocol>. [Accessed: 05-Dec-2019].
- [16] "Omnisharp Slack Channel." [Online]. Available: <https://app.slack.com/client/T0RE90CRF/C804W8JHE>. [Accessed: 05-Dec-2019].
- [17] "Syntax Highlight Guide." [Online]. Available: <https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide>. [Accessed: 14-Dec-2019].
- [18] "DiagnosticRelatedInformation Location should be an Location object · Issue #175 · OmniSharp/csharp-language-server-protocol," *GitHub*. [Online]. Available: <https://github.com/OmniSharp/csharp-language-server-protocol/issues/175>. [Accessed: 22-Oct-2019].
- [19] "OmniSharp.Extensions.LanguageClient 0.14.0." [Online]. Available: <https://www.nuget.org/packages/OmniSharp.Extensions.LanguageClient/>. [Accessed: 14-Dec-2019].
- [20] "XSERVER." [Online]. Available: <https://www.x.org/releases/X11R7.7/doc/man/man1/Xserver.1.xhtml>. [Accessed: 14-Dec-2019].
- [21] "SonarCloud for C# Framework Project," *SonarSource Community*, 27-Nov-2019. [Online]. Available: <https://community.sonarsource.com/t/sonarcloud-for-c-framework-project/17132>. [Accessed: 11-Dec-2019].
- [22] "SonarCloud." [Online]. Available: <https://sonarcloud.io/dashboard?id=dafny-server>. [Accessed: 14-Dec-2019].