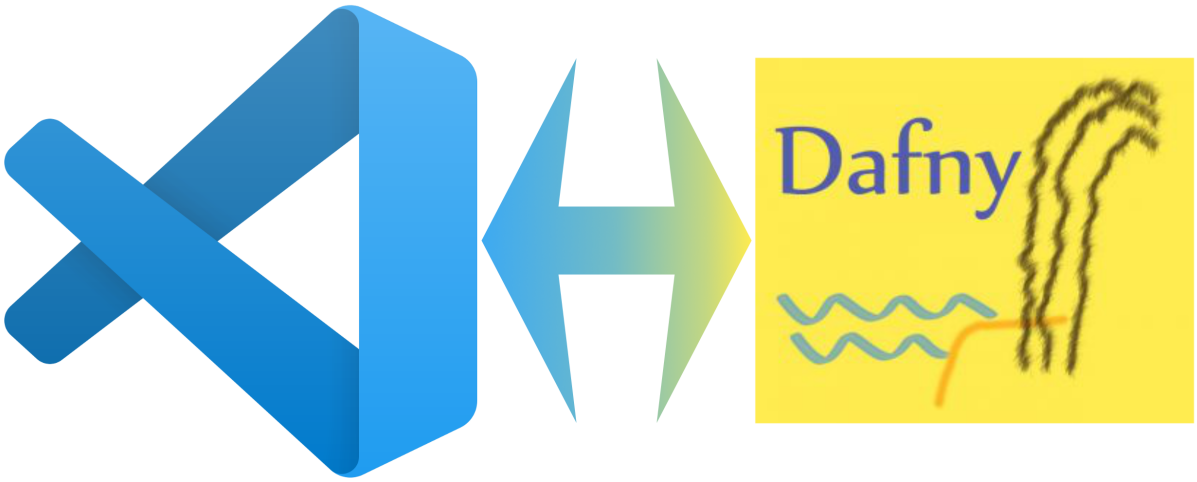

Project: Enhancing Dafny Support in Visual Studio Code

Project Plan

Authors:
Marcel Hess
Thomas Kistler

Supervisors:
Thomas Corbat
Fabian Hauser

Third Reader:
Olaf Zimmermann

Expert:
Guido Zgraggen

Table of Contents

1. Project Overview

During fall semester 2019/2020, Thomas Kistler and Marcel Hess worked on the term project «Dafny Server Redesign». [1] They will now continue this work by the bachelor thesis at hand. Aside the improvement of existing features, new and more complex features can be joined.

1.1 Initial Situation

The preceding project «Dafny Server Redesign» included the creation of a language server for Dafny, the integration with the Dafny library itself, as well as reworking the client for VSCode. The final architecture is shown in the following figure. [1]

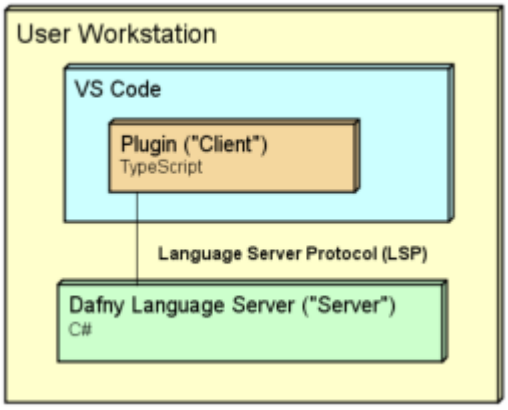

Figure 1 - Architectural Overview

As one can perceive, there is only one component «Server» which includes the Dafny library as well as the language server.

The coded feature set of the language server included, but is not limited to:

- Syntax Highlighting
- Highlighting Logical Errors
- Underlining Violated Promises
- Providing Counter Examples
- Auto Complete Suggestions
- Code Compilation and Execution
- Displaying Status Bar Information

The preceding thesis itself based on the bachelor thesis by Markus Schaden and Rafael Krucker. [2]

1.2 Goal

The goal of this bachelor thesis is the enhancement of the preceding term project.
During the first part of the project, existing features shall be improved in functionality, reliability, maintainability and usability. Some of the features imply flaws which shall be corrected.
The second part includes coding new features. Due to our personal interests we would like to implement a component to manage the symbol table of the user's Dafny code. This component can then be used for easier handling of existing features, and they can then be further improved.

1.3 Scope of Delivery

The final report will at least consist out of the following items:

- Abstract
- Management summary
- Technical report
- Developer documentation
- Project plan
- Personal reports
- Poster
- Legal declarations
- Code in digital form

The thesis will be handed in according to the internal guidelines of HSR.

2. Organization

This chapter covers the most important organizational aspects of the project in a concise way. It lists the time budget, that SCRUM will be used and when ordinary meetings take place.

2.1 Time Budget

The total working hours for the bachelor thesis is defined to be 12 ECTS or 360 hours per student. The project lasts for 17 weeks, which averages out in a nominal working time of 21.2 hours per week per student. During the last two weeks, no other classes take place and more time resources will be available. Furthermore, this calculation does not include the one-week Easter break.

2.2 Process Management

During the construction phase, we will use SCRUM for an agile development. At the beginning of a sprint, the sprint will be planned and each of the two developers can choose tasks to work on. All occurring tasks will be tracked with tickets in Redmine. Redmine's ticket base forms the product backlog. After the sprint has ended, a sprint review will take place to review the achieved work. At the end of a sprint, a short retrospective will be held to revise procedural aspects, too. There won't be daily reviews, since there are only two team members which can exchange information at any time. The sprints will last for one week only to be as flexible as possible. They will start and end on Friday.

2.3 Meetings

The weekly meetings with our supervisors Thomas Corbat and Fabian Hauser will be held each Thursday at 10:30am.

Thomas Kistler and Marcel Hess have a reserved time frame of 3x8h from Wednesday to Friday each week to work on the project. Whenever necessary, internal meetings can be held within this time frame.

3. Schedule and Scope

The following list provides the reader with an overview of all features that are outlined in this project plan. A more detailed discussion will be held throughout this chapter.

Improvement of existing features

- Syntax Highlighting
- Verification
- Autocompletion
- Counter Example
- GoTo Definition
- Compilation

New but simple features

- Hover Information

New major features

- CodeLens
- Refactorings
- Automated Contract Generation

Functionality for publishing

- Easy installation
- Cross Platform Usage

A nice feature addendum would be debugging, too. Due to its complexity, it will not be considered and may be subject to a dedicated thesis project in the future.

During the preceding term project, first attempts to implement CodeLens have already been made. The implementation is rather inefficient and should be improved. [1] To be able to do so, we got the idea to implement a component that manages the symbol table. This component is strongly connected to the abstract syntax tree (AST) provided by existing Dafny code. The AST can be used to create the symbol table. This component would also ease the implementation of many other features, such as GoTo Definition and Autocompletion. Thus, it wouldn't make much sense to improve features such as GoTo Definition first, and then implement the symbol table afterwards. This circumstance will be considered during scheduling the work load.

The reader will notice that the following schedule is very tight. It may not be possible to realize every idea mentioned in this chapter. Since the time is fixed, the scope has to be variable. Thus, the students will dynamically adjust the scope according to agile project management methods.

To be able to supply a deployable product in the end, the existing code has to be improved, ways for testing have to be found, complex concepts such as a symbol table managing component have to be added. To be able to achieve all of that and not get lost in unimportant details, the project is split into multiple stages which are time boxed. This ensures that we can work on all fields we would like to, so that we can learn as much as possible within this bachelor thesis.

3.1 Project Staging

In this project, we will use a modified variant of the unified process. This means that broadly speaking, the project is split into the following stages:

- Inception
- Infrastructure
- Construction
- Transition
- Conclusion

The inception phase is very short and just contains a few preliminary tasks. During the infrastructure phase, CI, automated testing and automated quality measures will be worked on. The transition stage includes making the plugin ready to publish. The last phase, conclusion, is meant to finish documentation and any open tasks left over from the preceding stages. The major part of the project will take place within the construction phase. This rather large stage is further divided into the improvement of existing features and the addition of new functionality. Details about the individual work packages can be found in the following chapters.

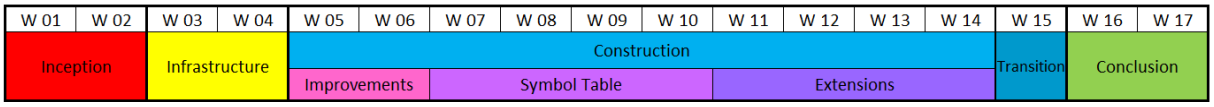

Figure 2 - Project Staging

3.2 Milestones

This subchapter will give a brief overview of the project's milestones. They are defined according to the project stages. There is a milestone at the end of every phase.

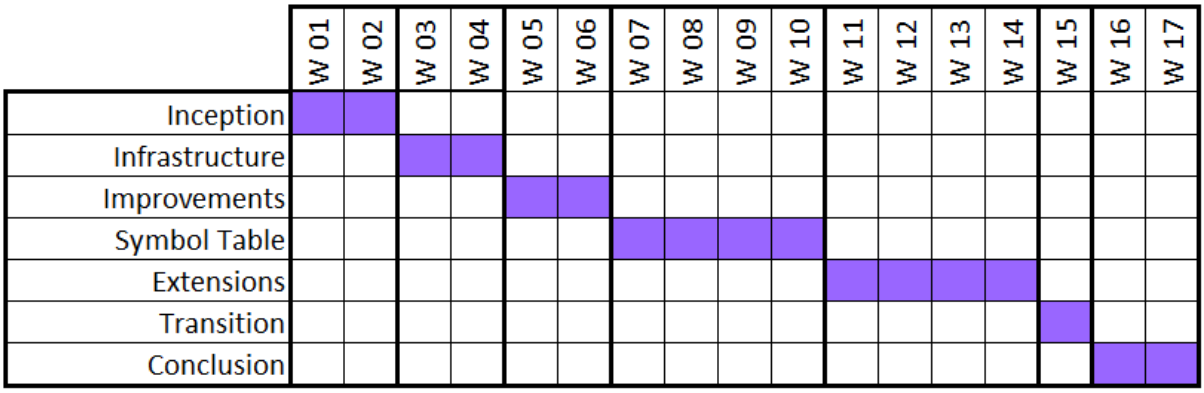

Figure 3 - Milestones

Note that this project will contain no elaboration stage, since most elaborating work had already been completed during the preceding project. There is no need to create a prototype, nor to evaluate new technologies for most aspects. Nevertheless, during the infrastructure stage, problems such as automated testing can be taken care of. [1]

3.2.1 Inception

This stage will only last for two weeks and contains the initial set up. This includes:

- Getting familiar with the project after Christmas break
- Defining the goals for the bachelor thesis
- Authoring the project plan
- Updating IDE's and other software
- Update frameworks and libraries, such as OmniSharp if applicable
- Creating the intermediary presentation

Because the expert's availability is limited, the intermediary presentation will be held at the beginning of the project to present the achievements of the preceding term project and is held at February, 27th 2020.

3.2.2 Infrastructure

This milestone contains revising the existing CI/CD environment. There are some leftovers from the preceding thesis for which improvements are necessary. [1] Aside solid integration tests, system testing shall be considered and SonarQube needs to be installed for all software parts. Testing and quality measures are further discussed in chapters 4 and 5.

3.2.3 Improvements

During the term project in the fall semester 19/20, a solid amount of features was implemented. However, some of them show flaws which limit their usability. [1] To increase the acceptance of the product, these features shall be improved during this stage. Features that require symbol table component to be available are not listed. Aside those, the following ideas are present:

- Verification
  - Show syntactical errors, too, not only logical errors
  - Smarter underlining for better visibility
- Counter Example
  - Simplify presentation
  - Automatic update when the user continues to type
- Compilation
  - Support for compilation arguments
  - Direkt Dafny/Boogie nutzen
- Improve Status Bar

3.2.4 Symbol Table Manager

The goal in this milestone is to implement a component that builds and manages the symbol table for the Dafny code. This would make many other features possible, such as renaming. Existing features such as CodeLens and GoTo Definition will also profit by this component. To acquire the required knowledge base, the students will attend the lecture "Compilerbau" by Prof. Dr. Luc Bläser at HSR.

3.2.5 Extensions

In the follow-up milestone, the symbol table manager shall be used at its glance for the following features and improvements:

- Autocompletion
  - Smarter suggestions, for example show only classes after the keyword «new»
  - Place cursor according to user's desire
  - Show placeholder arguments for methods and constructors
- GoTo Definition
  - Be more flexible to where the cursor position is
- CodeLens
  - Finish all work in progress
- Refactorings

3.2.6 Transition

Before the code will be frozen, we will invest one week in final work such as:

- Ensure easy installation
- Ensure cross platform compatibility

3.2.7 Conclusion

At the very end of the project, the documentation will be authored and completed, the final presentation has to be prepared and a poster will be created. The last two weeks function also as a buffer for unforeseen consequences.

3.3 Additional Potential Extensions

Depending on how quick progress will be made during construction phase, the following features can also be implemented:

- Display hover information
- Support for multiple files
- Auto contract generation

3.4 Work Packages

To coordinate our tickets and for time reporting, we will use Redmine. The first tickets have already been created:

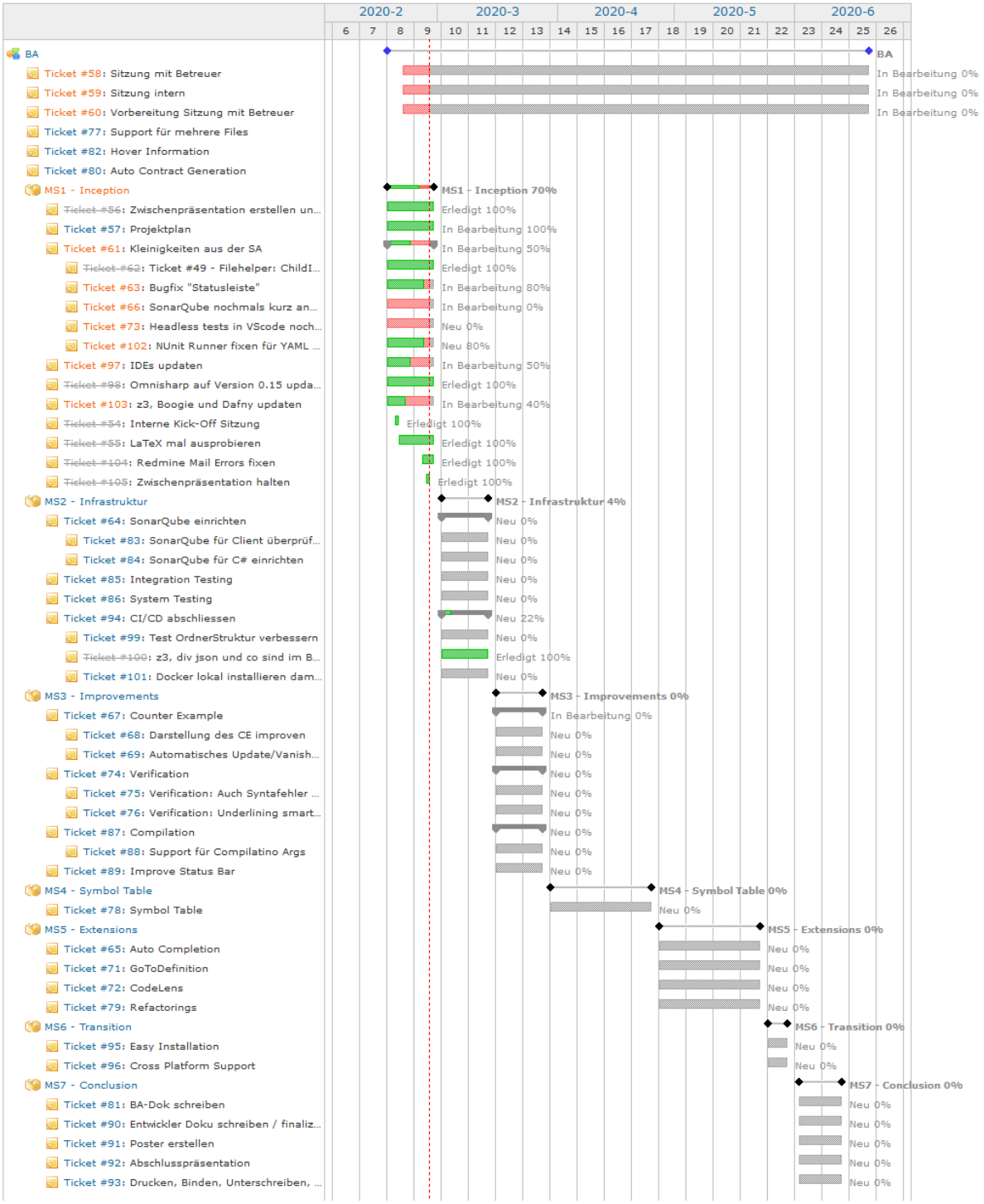

Figure 4 - Gantt Diagram

4. Infrastructure

Each member of the team will use his own notebook for the major part of the work.

The following technologies will be used during the research project:

Project planning and time reporting:	Redmine
Build server / CI server:	GitLab
Version control system:	Git
Host for Git:	GitLab
Runtime environment:	.NET Framework 4.6.1

IDEs:

- Microsoft Visual Studio
- Microsoft Visual Studio Code

Code quality and guidelines:

- ReSharper
- SonarQube

4.1 CI

The source code will be on GitLab. We will use GitLab's integrated pipeline system to compile the code, run the unit tests, and calculate quality metrics.

GitLab will notify all developers whenever a pipeline fails, for example if tests fail or the committed code is not compiling. Each member has to ensure that the pipelines run successfully.

4.2 CD

No continuous deployment will be used. The code base will be published only once at the end of development via the VSCode Marketplace.

5. Quality Aspects

This sub chapter covers quality aspects of this project, namely code reviews and style guidelines.

5.1 Code Reviews

Critical and difficult parts of the code shall underlie a code review. For this purpose, GitLab supports merge requests. We can use these each time a branch is merged into the master branch. We will especially keep in mind that not only one student works on a specific feature and the other student has no clue about it. Regular information and knowledge exchange will be enforced during the code reviews and sprint reviews.

5.2 Code Style Guidelines

We will apply the following coding guidelines:

- The naming of classes, methods and variables shall be in English only. Comments and the documentation shall also be in English.
- The code in C# shall basically fit the default guidelines of ReSharper and SonarQube. However, if we find that the readability of the code gets worse, exceptions and custom rules are allowed.

Generally, each developer has to ensure that the following guidelines are met:

- Good readability of the code
- Style guides as mentioned above redeemed
- Simplicity of the solution
- Documentation updated
- Proper tests written
- All tests are green

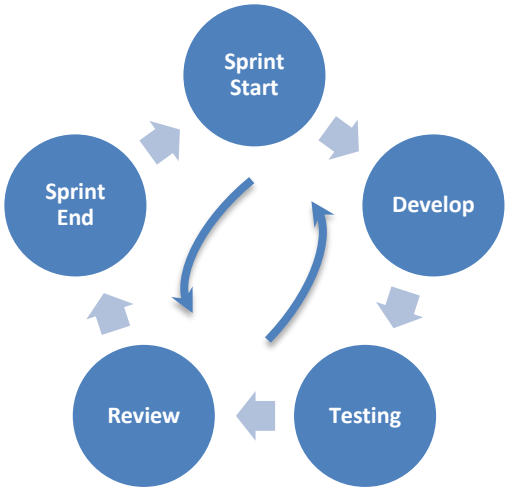

Figure 5 - Iterative Workflow

6. Testing

This chapter gives a brief overview of the planned testing infrastructure and guidelines. The chapter is divided into a section for each testing level.

6.1.1 Unit Testing

Simple classes, especially such that have none or few dependencies shall be properly unit tested. The tests shall be meaningful and their input shall cover different equivalence classes. We are going to use proper mocking to run automated unit tests on classes with dependencies, wherever possible. The targeted test coverage shall be at least 80%. However, if the goal is not met because trivial code paths such as constructors are not tested, no dummy tests for those code paths shall be added just to meet the goal.

The tests will be separated from the other parts of the project inside a specific project folder.
The developers can run these unit tests directly inside Microsoft Visual Studio. The unit tests will also be automatically performed by the CI process each time a commit is made and pushed to GitLab using the existing nUnit runner setup from the previous thesis.

6.1.2 Integration Testing

It is planned to use Omnisharp's Lanuage Client [3] to simulate communication between a client and the language server. The client can send all messages that the system under development supports and also receive the responses. Thus it is easy to test them for correctness.
This can be done directly inside Visual Studio in C#. Prior trials using this approach have shown that issues may arise, such as failures to parse the server's response. Those issues seem hard, or even unreal, to resolve since no obvious reason could be found. However, further investigation on this approach is worthwhile and shall take place. [1]

The advantage of using Omnisharp's client for testing is that it bases on the same technologies as the unit tests do. This would allow an easy integration into the CI system. Since they could be coded inside Visual Studio, we also had direct access to all methods and classes which makes writing tests very convenient. [1]

6.1.3 System Tests

The integration tests cover any code written in C#. The only code that is left to test is client code written in TypeScript. The preceding bachelor thesis used tests with a headless VSCode instance to test basic functionality of the client, which is very tedious, but necessary to test basic client functionality. [1]

7. References

[1] Hess, Marcel and Kistler, Thomas, “Dafny Language Server Redesign,” HSR Hochschule für Technik Rapperswil, 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] “OmniSharp.Extensions.LanguageClient 0.15.0.” [Online]. Available: https://www.nuget.org/packages/OmniSharp.Extensions.LanguageClient/. [Accessed: 26-Feb-2020].

Note: This document is based on the previous term project «Dafny Server Redesign».