



School of Computer Science
OST Eastern Switzerland University of Applied Sciences
Campus Rapperswil-Jona

Bachelor Thesis
CUTE EXTENSION FOR VS CODE
Spring Term 2022

Authors:
Christian Bisig, Dominic Klinger

Advisor:
Thomas Corbat

External Examiner:
Guido Zgraggen

Co-Examiner:
Frieder Loch

June 17, 2022

Abstract

Background: CUTE is a lightweight testing framework that offers the possibility to write automated C++ tests. To offer the best possible user experience, a plug-in for the Cevelop IDE exists. This plug-in provides functionality such as a test navigation, green/red bar test outcome visualizations, and difference viewers for assertion failures. Furthermore, it offers convenience features that make C++ testing as easily accessible. The CUTE framework is used in the C++ modules offered at OST Eastern Switzerland University of Applied Sciences.

Purpose: To offer a wider range of tooling choices in the C++ modules, it would be desirable to have the CUTE testing framework integrated into Visual Studio Code. This integration should be implemented in the form of a testing extension, which offers similar functionalities as the Cevelop plug-in and makes sure that C++ testing can be done as easy as possible. Through the integration into a widely known development environment such as Visual Studio Code, CUTE becomes accessible by a larger group of C++ developers.

Methods: In a first step the possibilities to integrate CUTE into Visual Studio Code had to be evaluated. This evaluation primarily focused on the different approaches to create a testing extension for VS Code. In a second step the evaluation focused on the possibilities to elaborate the required information from the test executables and from the test code itself. During this elaboration, prototypes were created for each key functionality. In a second phase of the project, the functionality was implemented based on the findings from this analysis. To make sure that the required functionality is working and can be used during the C++ modules, the newly created extension had to be tested on multiple different levels.

Results: The Visual Studio Code extension created in the scope of this thesis provides all mandatory functionalities to make the CUTE framework usable in VS Code. These functionalities include test discovery, navigation within the test code, creation of test runs that include a single or multiple test cases, and debugging of such test runs. In addition to these minimal required functionalities, convenience features were implemented with the aim to make the usage of the CUTE testing framework for C++ testing as easy as possible. These convenience tools simplify the creation of new test projects, new test suites and new test cases. The extension analyzes the test code and warns the users about potential problems such as unregistered tests.

Conclusion: The Visual Studio Code extension created in the scope of this thesis provides an additional choice of IDE for C++ students or generally while using the CUTE framework. The user interface is familiar to many developers that have worked with VS Code before. This makes using the extension easy right from the beginning and the powerful convenience tools further simplify C++ testing using the CUTE framework.

Management Summary

This chapter contains the management summary of the CUTE extension for VS Code thesis. It should provide an overview of the whole project and the thereby elaborated results. The overview is provided in the form of a summary of the whole project documentation.

Background

CUTE is a testing framework for C++ code, that allows it to write automated tests and arrange them in test suites. CUTE is used in the practical exercise lessons of the C++ modules at the OST Eastern Switzerland University of Applied Sciences. As the CUTE testing framework's only integration is a plug-in for the Cevelop IDE, it is strongly recommended to use Cevelop for the exercises. The Cevelop plug-in offers a set of functionalities such as support to initialize and set up new tests, options to navigate throughout the test code, a green/red bar view, a difference viewer for failing tests and a rerun possibility for single tests.

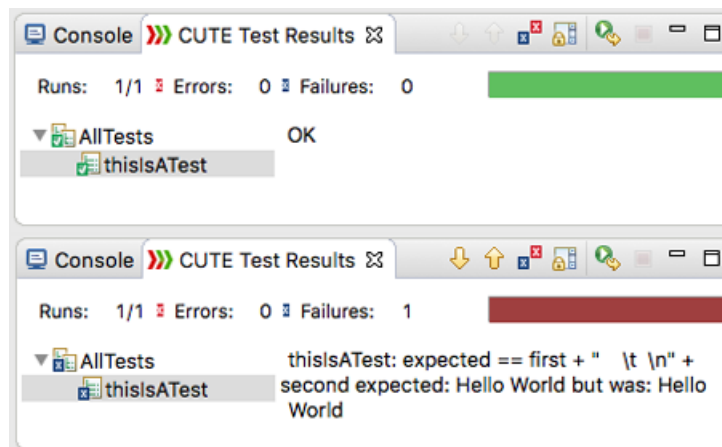


Figure 1: Cevelop CUTE Plug-In

Objective

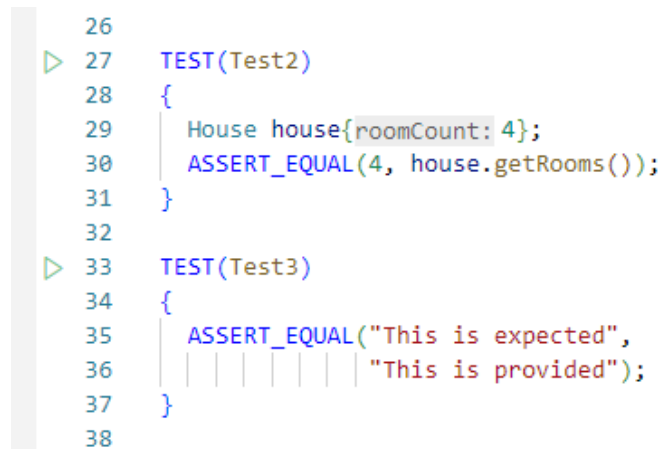
To provide a larger tooling choice, an integration of the CUTE testing framework into Visual Studio Code should be planned and developed in the scope of this thesis. The integration should be made available in the form of a VS Code extension that can simply be installed from the VS Code Marketplace. The CUTE extension for Visual Studio Code should provide similar functionalities as the Cevelop plug-in and make C++ testing as easy as possible. The integration of the CUTE framework into a well-known and widely used code editor such as VS Code, enlarges the group of developers that potentially use CUTE for C++ testing massively even outside the OST Eastern Switzerland University of Applied Sciences.

Proceeding

In the scope of this thesis an extension for Visual Studio Code was implemented to integrate the CUTE testing functionality and make it available to a larger developer community. Multiple different approaches to implement the extension were analyzed in the course of this project. For each key functionality a prototype was developed and later used for a more detailed evaluation of the different options available. The minimal required functionality that is needed to make the CUTE framework usable in VS Code includes the test discovery, the start of test runs containing a single or a selection of tests, and the visualization of the test results. Additional features that were evaluated during the project include support to set up new CMake based test projects and an option to easily create new test suites and test cases. In addition, the possibilities of offering test code analysis that finds potential problems within the test code were investigated.

Evaluation

The evaluation has shown that the best approach to set up the new CUTE extension for Visual Studio Code is to base it on the newly offered VS Code Testing API. This Testing API is a native interface offered by VS Code directly, that allows the registration of test items such as test cases and test suites. Further it offers endpoints that handle the start and management of test runs. The Testing API is available since July 2021 and replaced the before widely used Test Explorer UI third party extension. The native Testing API comes with a user-friendly user interface for the test explorer that is familiar to many developers. Further, a neat integration into the code editor window, that provides further functionalities from directly within the code, is available.

A screenshot of the Visual Studio Code editor interface. The code is written in C++ and uses the CUTE testing framework. It shows two test functions, TEST(Test2) and TEST(Test3), each enclosed in a block. TEST(Test2) contains a struct 'House' with a 'roomCount' member and an 'ASSERT_EQUAL' macro call. TEST(Test3) contains an 'ASSERT_EQUAL' macro call with string literals. The code is color-coded, and the line numbers 26 through 38 are visible on the left margin. The editor's light theme and syntax highlighting are clearly visible.

```
26
27  TEST(Test2)
28  {
29      House house{roomCount: 4};
30      ASSERT_EQUAL(4, house.getRooms());
31  }
32
33  TEST(Test3)
34  {
35      ASSERT_EQUAL("This is expected",
36                  "This is provided");
37  }
38
```

Figure 2: VS Code CUTE Editor Integration

Result

The final version of the CUTE extension for VS Code contains multiple implementations of the test discovery logic. The most performant and reliable information is based on the newly introduced `TEST(...)` macro, that should be used to define tests. This macro allows it to query all tests and their location directly from the test executable, by starting it with the ‘-display-tests’ arguments. The second implementation is based on information from the test code itself. The names and locations of CUTE tests are thereby extracted from the code using language server providers and regex matching of the test code. The reliability and performance of this second implementation heavily depends on the underlying language server provider. The CUTE extension supports the C/C++ for VS Code (CppTools) extension by Microsoft and the Clangd extension as language server providers. To achieve the best possible performance, it is recommended to use the Clangd extension as language server provider.

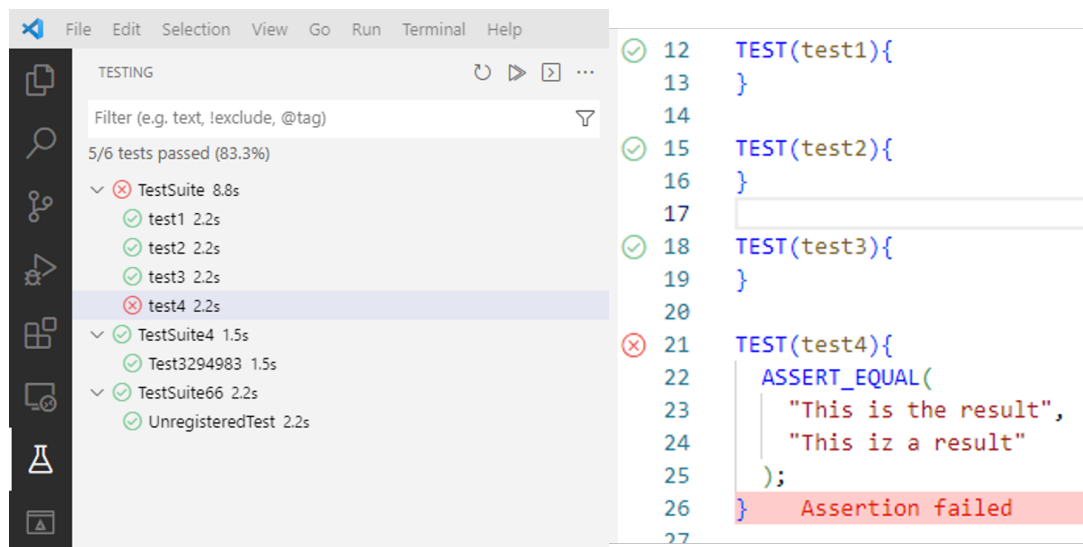


Figure 3: VS Code CUTE Extension UI

Code Generation

The CMake based project generation was implemented in the form of a project template that consists of a library, an executable and a test executable. The template project gets configured in the CMake file according to its structure.

The final CUTE extension for VS Code provides a command to set up new test suites in separate cpp files. The developers can configure the test suite during the setup process. The CMake configuration file is updated based on the newly created test suite file. The CUTE extension further offers a Snippet to create new tests that are declared using the newly created `TEST(...)` macro.

Unregistered Tests

The information from the supported language server providers are used in the released version of the CUTE extension to analyze the test code for potential problems. Thereby, the code is specifically analyzed for tests that are never registered to a test suite or directly to a test runner (unregistered tests) and therefore will never be executed. The CUTE extension does not just warn the users about these unregistered tests, it also provides quick fixes to register the test to an existing test suite or to create a new suite and register the test to that.

```
9
10 Test UnregisteredTest is not registered in a suite and will not be executed (TEST_UNREGISTERED)
11 View Problem Quick Fix... (Ctrl+.)
12 TEST(UnregisteredTest){
13
14 }
```

Figure 4: VS Code CUTE Unregistered Test

Unregistered Suites

The test code is analyzed for test suites that are never called. All tests registered to a suite that is not passed to a test runner, will never be executed. The developer gets a warning about this problem and a quick fix to register the test suite to a test runner in the main function is available.

```
13
14 } Suite TestSuite66 is never referenced and won't be executed (SUITE_UNREGISTERED)
15 View Problem Quick Fix... (Ctrl+.)
16 bool runTestSuite66(cute::runner<cute::xml_listener<cute::ide_listener<>>> runner)
17 {
18 | | cute::suite TestSuite66{};
```

Figure 5: VS Code CUTE Unregistered Suite

Legacy Syntax

The CUTE extension analyzes the code for test declarations which are not based on the new TEST(...) macro. For these test declarations, no implementation location can be provided, when using the executable based test discovery implementation. A quick fix is available to update the legacy test declarations to the new TEST syntax. The developer has the choice of updating all legacy tests at once or rather test by test.

```
13
14 } The TEST(...) macro should be used to declare test cases (LEGACY_TEST_DECLARATION)
15 View Problem Quick Fix... (Ctrl+.)
16 void LegacyTest(){
17
18 }
```

⚠ Some of your tests are declared using the legacy syntax. This might lead to performance problems

Source: CUTE Testing (Extension) Update Mark Ignore

Figure 6: VS Code CUTE Legacy Syntax

Configuration Options

The released version of the CUTE extension for Visual Studio Code offers multiple configuration options, such as the test discovery mode or the legacy code handling. When the default configuration is used, the extension automatically decides which mode to use based on installed language server provider extensions and the used CUTE framework version.

Conclusion

The CUTE extension for Visual Studio Code is available in the VS Code Marketplace and is supported on Windows and Linux. It provides the CUTE testing framework's functionality to the VS Code community. The extension offers all features that are also offered by the Cevelop plug-in. In addition, the VS Code extension offers even more powerful convenience tools based on the language server information. These convenience tools make C++ testing, using the CUTE framework, even easier.

Installation instructions, tutorials and demonstrations are available [here](#).



Figure 7: VS Code CUTE YouTube Instructions

Contents

1	Introduction	10
1.1	Initial position	10
1.1.1	CUTE	10
1.1.2	Cevelop	10
1.1.3	Visual Studio Code	11
1.2	Objective	11
1.3	Team	11
1.4	Supervisor and Examiner	11
2	Requirements Analysis	12
2.1	General Description	12
2.1.1	Product Perspective	12
2.1.2	Product Functionality	12
2.1.3	User Characteristics	12
2.1.4	Dependencies	13
2.2	Use Cases	13
2.2.1	Use Case Diagram	13
2.3	Additional Requirements	18
2.3.1	Reliability	19
2.3.2	Performance and Efficiency	20
2.3.3	Usability	21
2.3.4	Maintainability and Adaptability	21
2.4	System context	23
2.5	Interfaces	23
2.6	Additional Constraints	24
3	Domain Analysis	25
3.1	Structure Diagram - CUTE testing	26
3.2	System Sequence Diagram	27
3.2.1	UC-1: Navigate To Test Case	27
3.2.2	UC-2: Run Tests & UC-4: Analyze TestResult	28
4	Decisions	29
4.1	Fundamental Architecture Decision	29
4.1.1	Build the extension from scratch	29
4.1.2	Test Explorer UI Extension	29
4.1.3	VS Code Testing API	30
4.1.4	C++ TestMate Extesnsion	31
4.1.5	Decision	31
4.2	Test Discovery	31
4.2.1	Executable Based	31
4.2.2	Code Based	34
4.2.3	Decision	34
4.3	Result Format	35
4.3.1	Executable StdOut Result	35
4.3.2	File Based XML Result	36
4.3.3	Decision	36
4.4	Language Server Provider	37
4.4.1	CppTools - C/C++ for Visual Studio Code	37
4.4.2	Clangd	38

4.4.3	Decision	38
4.5	Language Server Integration	39
4.5.1	Custom Language Server Client	39
4.5.2	Use Built-in Visual Studio Code Commands	40
4.5.3	Use Proprietary Language Server Provider Commands	41
4.5.4	Decision	41
4.6	Debugging Libraries	41
4.7	MSYS2	42
4.8	User Guide	42
5	Design	43
5.1	Architecture overview	43
5.1.1	CUTE Extension	43
5.1.2	Supported Extensions	45
5.2	CUTE Extension Components	47
5.3	Class Level Overview	51
5.3.1	Testing Component	51
5.3.2	Tools Component	53
5.3.3	Utilities Component	57
5.3.4	Environment Component	58
5.4	CUTE	60
6	Implementation	62
6.1	Used technologies	62
6.1.1	Extension Folder Structure	62
6.1.2	Extension Manifest	63
6.1.3	Extension Entry File	65
6.2	Quality measures	66
6.2.1	Automated Tests	66
6.2.2	Manual Tests	66
6.2.3	Continuous Integration (CI)	66
6.3	General Extension Logic	67
6.3.1	TEST(...) macro available	67
6.3.2	TEST(...) macro not available	67
6.3.3	Clangd Installed	67
6.3.4	C/C++ for VS Code Installed	68
6.3.5	Clangd & C/C++ for VS Code Installed	68
6.3.6	No Language Server Provider (LSP) Installed	68
6.4	RunHandler Implementation	69
6.4.1	TestRunHandler	69
6.4.2	TestDebugHandler	71
6.5	TestFinder Implementation	72
6.5.1	ExecutableTestFinder	72
6.5.2	CodeTestFinder	74
6.6	CodeAnalyzer Implementation	76
6.6.1	Find Test Runners	76
6.6.2	Find Test Suites	78
6.6.3	Find New Tests	80
6.6.4	Find Legacy Tests	82
6.7	CMakeParser Implementation	84

7	Results	85
7.1	Feature set	85
7.1.1	Test Explorer	85
7.1.2	Code Editor Integration	86
7.1.3	Project Generation	87
7.1.4	Legacy Syntax Converter	88
7.1.5	Assert Failure Analysis	88
7.1.6	Quick Fixes	89
7.2	Feature comparison	94
7.3	Metrics	95
8	Testing	97
8.1	Functionality Testing	97
8.1.1	Test Protocol	113
8.2	Nonfunctional Requirements (NFR) Testing	114
8.2.1	Test Protocol	115
9	Conclusion	116
9.1	Summary	116
9.2	Result evaluation	116
9.3	Reached goals / open work	117
9.4	Future view	117
10	Project Management	118
10.1	Organization	118
10.1.1	Project Contributors	118
10.1.2	Initiator / Supervisor	118
10.1.3	Expert / Examiner	118
10.2	Work Breakdown Structure	119
10.3	Cost Estimate	120
10.4	Time Planning	120
10.4.1	Project Phases	120
10.4.2	Milestones	121
10.4.3	Timeline	123
10.4.4	Epic Estimate	124
10.4.5	Workitems	124
10.5	Risk analysis	125
10.5.1	R1: CUTE does not fulfil requirements for VS Code test environment	125
10.5.2	R2: VS Code Testing API does not fulfil the requirements	125
10.5.3	R3: Incorrect handling of the requirements	126
10.5.4	R4: Wrong architectural decisions	126
10.5.5	Risk Matrix	127
10.6	Quality Management	128
10.6.1	Tools	128
10.7	Development Tools	129
10.7.1	Documentation	129
10.7.2	Extension	129
10.7.3	Development Server	131
10.7.4	Code Style Guidelines	131
11	Glossary	132

12 Bibliography	136
13 Content Lists	140
A General Testimonial	142
A.1 Team / Organization / Communication	142
A.2 Meetings	142
A.3 Challenges	142
A.3.1 Personal Experience	143
A.3.2 Dominic Klinger	143
A.3.3 Christian Bisig	143
A.4 Conclusion	143
B Time report	144
B.1 Time stats	144
B.2 Issues	144
C Meeting protocols	146
C.1 Week1	146
C.2 Week2	147
C.3 Week3	148
C.4 Week4	149
C.5 Week5	150
C.6 Week6	151
C.7 Week8	152
C.8 Week9	153
C.9 Week10	154
C.10 Week11	155
C.11 Week12	156
C.12 Week13	157
C.13 Week14	158

1 Introduction

This chapter contains a general overview of this project's background and purpose. To provide such an overview, the situation as it is before the project was started and the prerequisites are explained. This section is followed by an overview of the objectives that are set for the project. The last part of this chapter contains a listing of all project members from the team to the supervisors and experts.

1.1 Initial position

As the practical lessons of the OST Eastern Switzerland University of Applied Sciences' [1] C++ modules are heavily based on CUTE [2] unit tests. Cevolop [3] offers a plug-in for the CUTE testing framework [10] and therefore, it is currently strongly recommended for students to use Cevolop as Integrated Development Environment (IDE) to do their exercises. In order to offer more flexibility in tooling, the decision was taken to develop an extension for Visual Studio Code [4] that offers basic CUTE testing functionality. This additional CUTE extension allows it to future students who attempt the C++ modules to choose between Cevolop and Visual Studio Code as their desired development environment.

1.1.1 CUTE

CUTE [2] is short for C++ Unit Testing Easier. As the name supposes it is a unit testing framework by Peter Sommerlad [10] that aims to make its usability as simple as possible. Amongst others, the OST Eastern Switzerland University of Applied Sciences [1] is using the CUTE testing framework for its C++ classes. CUTE can basically be used as a standalone framework without any IDE integration. But to get the best CUTE testing experience, it is recommended to use CUTE [2] in combination with Cevolop [3], a C++ Development IDE built with Eclipse CDT [6]. Cevolop offers a CUTE Plug-in [5] that allows test navigation, green/red bar results, diff-viewer for failing tests and wizards to create test frames. The listing below contains all available features offered by the Cevolop plug-in. A more detailed description of the Cevolop plug-in's functionality can be found here [5].

- Wizards to initialize and set up new tests
- Test navigator with green/red bar
- Diff-viewer for failing tests
- Rerun functionality for single test (e.g. a failed one)

1.1.2 Cevolop

Cevolop [3] is a free and platform independent Integrated Development Environment (IDE) that can be used for C++ development. Cevolop extends Eclipse CDT [6] (C/C++ Development Tooling) and offers many additional features such as the already mentioned CUTE testing support or additional quick fixes and refactoring options. Cevolop is maintained by the OST Institute for Software [7]. As the CUTE testing framework [10] is neatly integrated into Cevolop it is recommended that students who take the C++ classes at the OST Eastern Switzerland University of Applied Sciences [1] use it to do their exercises.

1.1.3 Visual Studio Code

Visual Studio Code [4] is a free, open source and platform independent source-code editor by Microsoft. Visual Studio Code offers many extensions that provide features such as syntax highlighting, code completion, snippets and tools that allow efficient code refactoring. VS Code was released in 2015 and quickly gained popularity with developers. One of the major advantages of Visual Studio Code is its easy extensibility and customizability. There are many extensions for various languages and applications available. If some use case should not be covered by an existing extension there is a well-documented framework available to write and publish own extensions. These extensions can be easily installed from the marketplace [8]. After the installation each extension will run in its own process, what should minimize the performance impact on the IDE.

1.2 Objective

The goal of this bachelor thesis is to develop an extension for Visual Studio Code [4] which provides basic support for the CUTE testing framework [10]. The feature set should be similar to the features offered by the CUTE plug-in for Cevelp [5]. In a first step it should be analyzed whether the whole functionality needs to be implemented from scratch or whether the CUTE extension can be built on an existing Visual Studio Code extension that provides some baseline functionality. To take this fundamental architectural decision the different possibilities need to be analyzed at the beginning of the project. The final decision should be taken in collaboration with the project's supervisor.

A minimal feature set should include the possibility of compiling a CUTE test project from within the IDE, run test cases and get the test results in some visualized form like a green/red bar. Changes on the CUTE framework [10] itself are allowed if some features require such an adaptation or extension of the framework logic.

A further objective of this project is to setup portable and IDE-agnostic build environments for the module's exercises based on CMake [9]. It should be as easy as possible for students to set up a new project that includes CUTE tests. Cevelp offers the possibility to create CUTE executables, C++ executables and library projects in separate steps. To offer a more convenient way to set up new projects in just one step it would be an option to provide a template project containing all the required files and the corresponding CMake configuration.

The CUTE plug-in for Cevelp offers some additional convenience features such as automatic test registration, creation of new test suites, rerunning of specific tests or suites, comparison of expected and actual values for failed assertions, navigating to specific test cases and further test code analysis. Some of these features heavily rely on C++ code information that might not be available in Visual Studio Code, others might be feasible as optional project goals.

1.3 Team

- Dominic Klinger, dominc.klinger@ost.ch
- Christian Bisig, christian.bisig@ost.ch

1.4 Supervisor and Examiner

- Guido Zraggen - Examiner, zraggen@gmail.com
- Thomas Corbat - Supervisor, thomas.corbat@ost.ch
- Frieder Loch - Co-Examiner, frieder.loch@ost.ch

2 Requirements Analysis

This chapter should provide an overview of the requirements for the CUTE extension. Thereby the requirements should be split according to the different use cases of the product and afterwards be analyzed in a more detailed manner. Based on the requirements defined in this chapter, test cases will be designed. This should ensure that all required features are present in an acceptable form. Following this introduction, this chapter will contain a general description of the Visual Studio Code CUTE extension and a detailed explanation of all functional and nonfunctional requirements. An overview of the defined use cases can be found in the form of a use case diagram. After the use case diagram, a brief description of each underlying use case can be found. In the last section of this chapter the nonfunctional requirements regarding reliability, performance, usability, and adaptability will be described.

2.1 General Description

This section contains a general description of all requirements and expectations towards the CUTE extension for VS Code. These requirements are analyzed from a product perspective, from a functionality perspective, and from a user characteristics perspective. In addition to the thereby analyzed requirements the external dependencies are listed and described.

2.1.1 Product Perspective

The CUTE extension for Visual Studio Code is an application that supports the handling of C++ testing based on the CUTE framework [10]. Currently such an extension, that supports CUTE testing, is only available for Cevelop [3]. To increase the flexibility of choosing the desired tooling for the C++ modules at the OST University of Applied Sciences [1], this new extension should offer a somewhat similar experience and functionality. After this project is finished, an additional way of using the CUTE testing framework [10], in a way that is as simple as possible, should be available.

2.1.2 Product Functionality

The CUTE extension should at least be able to compile a C++ solution that contains CUTE tests and find the there defined test cases and test suites. The extension should then provide a possibility to run the discovered tests and if possible, also just a selection of the discovered tests or suites. Test results should be visualized in some form to let the user know which tests failed and which were successful. Further the extension should provide an easy way of setting up a new project that contains CUTE tests, preferably in a single step in the form of a template project. Additional optional features such as the simplified generation of new test suites, user warnings if test cases should never be called in the code or the debugging of test cases should also be included into the finished product if the available time allows it. To evaluate further features the feature set of the existing Cevelop extension [5] should be consulted. The main goal of the extension is to offer the use of the CUTE test framework [10] in a way that is as convenient as possible.

2.1.3 User Characteristics

The users of the CUTE extension for Visual Studio Code are predominantly students who visit the C++ modules at the OST University of Applied Sciences [1]. These students have different levels of C++ knowledge and experience therefore the complexity of using the CUTE extension should be kept as small as possible. Besides the students there might be users outside of the OST who use CUTE [2] as their C++ testing framework of desire.

2.1.4 Dependencies

The main dependency of this project is the testing framework CUTE [10]. CUTE stands for C++ Unit Testing Easier and provides unit testing functionality for C++ in a header only way. The basic structure of a project being tested is a library, an executable and a test-executable. Both executables link the library part to build the application. Further dependencies might arise throughout the project for example to retrieve language information. Additional information about the dependencies of the CUTE extension can be found in the chapter Software Architecture.

2.2 Use Cases

The users need to be able to perform the following actions using the functionality of the CUTE extension to get the same functionality and user experience as this is the case in the CUTE extension for Cevelop [5].

2.2.1 Use Case Diagram

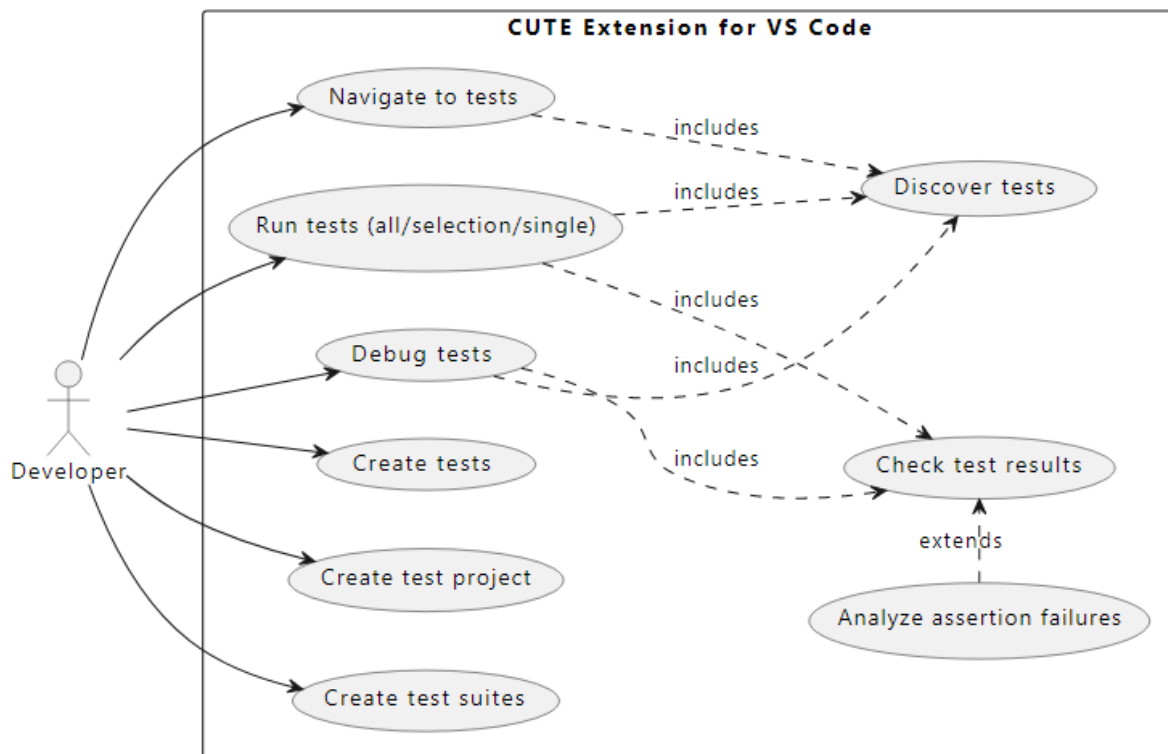


Figure 8: Use case diagram

Use Case UC-1: Navigate to tests

Scope:	CUTE Extension: Code Navigation
Primary Actor:	C++ Developer / CUTE Tester
Preconditions:	The tests of a project and their implementation locations are discovered
Description:	Users should have a possibility to easily navigate to the implementation location of a test case. The user should have the ability to access a list of tests, select a specific test out of that list and then automatically be navigated to that test's location. This feature should simplify the navigation through the project that is being tested.

Use Case UC-2: Run tests

Scope:	CUTE Extension: Test Execution
Primary Actor:	C++ Developer / CUTE Tester
Preconditions:	The tests of a project and their implementation locations are discovered
Description:	Users should be able to execute a project's test cases. The user should get an overview of all tests in a specific project and then run them from there. The user should have the choice of running all discovered tests or just a subset of them e.g., tests belonging to a specific test suite. The user should also be able to run single test cases. It should be possible to start a test run either from the test explorer or directly from within the code at the implementation location of a specific test. If multiple tests should be part of a test run, these should be executed in parallel to improve the performance.

Use Case UC-3: Debug tests

Scope:	CUTE Extension: Test Execution
Primary Actor:	C++ Developer / CUTE Tester
Preconditions:	The tests of a project and their implementation locations are discovered, debugging extension is installed
Description:	Users should be able to debug a project's test cases. The user should get the choice of whether to simply run a test or rather attach a debugger to the test execution. From the user experience point of view there should be no big difference on how to start a test run, whether with or without debugger attached. Therefore, all the above-mentioned locations to start a normal test run should also be available to start a test debug run. If multiple tests should be selected at the start of a debug run, these should be executed serial and not parallel as it should be the case in a normal test run.

Use Case UC-4: Check test result

Scope:	CUTE Extension: Result Analysis
Primary Actor:	C++ Developer / CUTE Tester
Preconditions:	A test is executed and the test result is available
Description:	After a test or a selection of tests was executed, the user should be able to get a clearly arranged overview of the outcome of each test case. This overview should be implemented as a red / green bar experience similar to what the CUTE extension for Cevelop is offering. It should be easily distinguishable which tests ran through successfully and which failed.

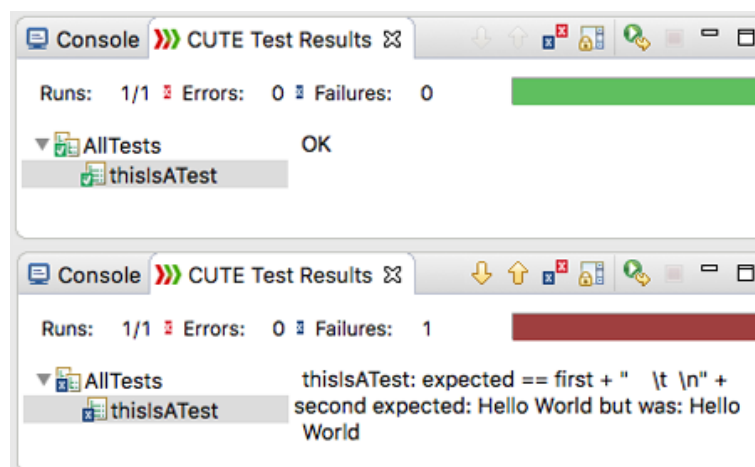
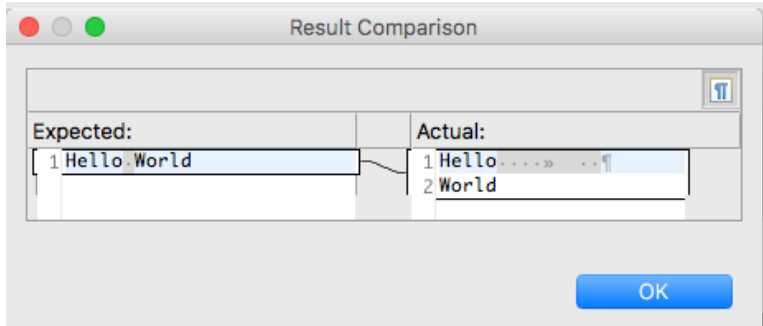


Figure 9: Cevelop Green/Red-Bar View [11]

Use Case UC-4.1: Analyze assertion failures

Scope:	CUTE Extension: Result Analysis
Primary Actor:	C++ Developer / CUTE Tester
Preconditions:	A test is executed and the test result is available. The test failed because of an assertion failure.
Description:	<p>Users should be able to quickly determine why a test run failed. If a test failed because an assertion was wrong, the user should be able to easily analyze the difference between the actual and the expected result. The currently available Cevalop extension offers a difference view that allows the comparison of the expected and the actual result with difference visualization support. Below the difference viewer of the Cevalop extension can be seen.</p>  <p>Figure 10: Cevalop Assert Difference Viewer</p>

Use Case UC-5: Create Test Cases

Scope:	CUTE Extension: Test development
Primary Actor:	C++ Developer / CUTE Tester
Preconditions:	A test project is set up and a test file is available.
Description:	<p>Users should be able to create additional test cases as easy as possible. The CUTE extension should provide support when a user wants to create a new test case. Newly created tests should be listed in the test explorer after the project was recompiled. Afterwards all other functionalities should be available for the new test case.</p>

Use Case UC-6: Create Test Suites

Scope:	CUTE Extension: Test development
Primary Actor:	C++ Developer / CUTE Tester
Preconditions:	A test project is set up. The CUTE framework is available in the workspace.
Description:	Users should have the possibility to easily create additional test suites. The CUTE extension should provide support when a user wants to create a new test suite. After the creation of an additional test suite, the user should have the ability to add tests to this suite. Newly created test suites should be listed in the test explorer after the project was recompiled.

Use Case UC-7: Create Test Projects

Scope:	CUTE Extension: Test development
Primary Actor:	C++ Developer / CUTE Tester
Preconditions:	An empty workspace is opened in VS Code.
Description:	Users should be able to create a new test project without much effort. The initialization of a new project consisting of a library, an executable and a CUTE test executable should be simplified by the CUTE extension. Preferably a solution that just requires a single step to set up such a new project including a corresponding CMake file should be part of the CUTE extension.

Use Case UC-8: Discover Tests

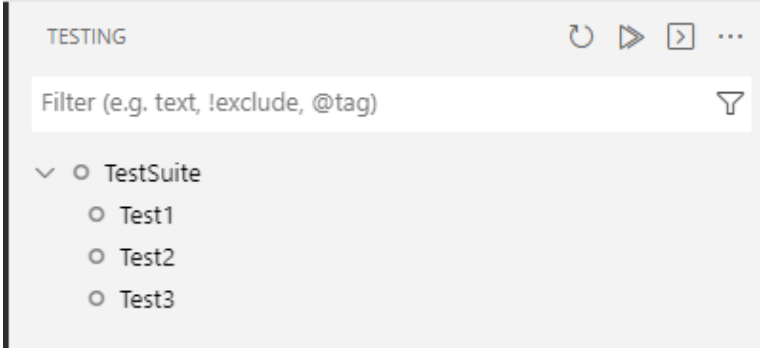
Scope:	CUTE Extension: Test development
Primary Actor:	C++ Developer / CUTE Tester
Preconditions:	An empty workspace is opened in VS Code.
Description:	<p>Many of the above-mentioned use cases depend on a listing of the test suites and test cases belonging to a project. Therefore, the CUTE extension should provide the functionality to discover these test suites and test cases within a workspace. The discovered tests should then be presented in a clearly arranged manner in the test explorer view.</p> 

Figure 11: Test Explorer View

2.3 Additional Requirements

In this section of the chapter requirements analysis an overview of the additional quality requirements that should be fulfilled by the CUTE extension can be found. These additional requirements include the nonfunctional requirements (NFRs), given interfaces that need to be used and further surrounding conditions. The functional requirements are split into the categories of reliability, performance, usability, and adaptability. These nonfunctional requirements were analyzed based on the ISO 25010 standard [12].

2.3.1 Reliability

This section contains the non-functional requirements regarding the product's reliability. Each of the below defined non-functional requirements should help with improving the CUTE extension's reliability.

NFR-1: Maturity

Business Goal:	The users should be notified in an understandable way if an unexpected error occurs. The problems thereby should become reproducible. With this information the problems can then in a first step be analyzed and afterwards be mitigated.
Scenario:	During the usage of the CUTE extension for Visual Studio Code an unexpected error occurs, and the extension stops working without any useful information. The problem is possibly not reproducible and therefore cannot be mitigated.
Reaction:	Users are informed in an understandable way if a malfunction should occur during the usage of the CUTE extension. The users get a hint of what might has gone wrong. The user notifications should contain all the information needed to fully understand the problem that has occurred.
Measure:	The extension should always be able to notify the users if something went wrong. The messages should always contain a reason for the problem and the component that has caused the malfunction. The most common problem sources should be covered with specific exception handlers.

NFR-2: Fault Tolerance

Business Goal:	The CUTE extension for Visual Studio Code should be able to handle malformed input such as malformed XML-results from the test executable or tests missing their implementation location when queried from the test executable. The CUTE extension should also be able to work if none of the supported debugging extensions is installed.
Scenario:	An incomplete response from a test executable or a missing extension on which some functionality is based crashes the extension whereafter Visual Studio Code needs to be restarted or the extension needs to be reloaded manually by the user.
Reaction:	Malformed input from any component is checked and errors are captured, and the user gets notified with an understandable message. If no debug extension should be available this option should not be made available. If the test execution does not return the location of a test implementation, the navigation functionality should be disabled.
Measure:	The extension should never crash based on malformed test results or missing debug extensions. There are safeguards in place to handle missing extensions, for example missing debug extensions or missing language information provider. The user should always be notified about the currently available feature set.

2.3.2 Performance and Efficiency

This section contains the non-functional requirements regarding the product's performance and efficiency. The below defined requirement should ensure that the CUTE extension for VS Code is usable without lagging. This should help to provide a good user experience.

NFR-3: Time Behavior

Business Goal:	The CUTE extension should not have any negative performance impact on any other part of the development experience using Visual Studio Code. Test discovery should be possible in a useful time frame. Test run start, stop and debugging should be possible immediately. This should lead to a flawless user experience.
Scenario:	The CUTE extension slows down the C++ development process using Visual Studio Code which leads to a bad user experience. Tests are discovered too slow which makes the testing extension unusable. The test execution is slowed down by the CUTE extension.
Reaction:	The test execution should be parallelized which should lead to an improved time behavior when running tests. Test discovery should always try to use the most performant option (direct querying of the test executable) and notify the user if this option should not be available.
Measure:	The test discovery using the most performant option should not take longer than 10 seconds. The execution of the test executable through the CUTE extension should not lead to a more than 1.5x increased time behavior compared to directly running the test executable.

2.3.3 Usability

This section contains the non-functional requirements regarding the product's usability. The below defined requirement should make sure that the extension is easily usable.

NFR-4: Operability

Business Goal:	The CUTE extension should provide an easy to learn experience that uses controls that are familiar to the average developer. Transition from Cevlop to Visual Studio Code should be possible without any additional learning required. The project, test suite and test case generation should be possible without having to write the whole C++ code by hand.
Scenario:	The test information is distributed and hidden in some Visual Studio Code menus. The controls used are different compared to any other test extension for Visual Studio Code. The user cannot figure out how to run and debug tests.
Reaction:	The user experience should be as similar as possible to other Visual Studio Code test extensions and also not be far away from the Cevlop extension. Users can intuitively use the CUTE testing extension in Visual Studio Code.
Measure:	The CUTE extension uses familiar controls like the Visual Studio Code test explorer, which is used by other test extension. There should be no instructions required for a user to be able to run tests.

2.3.4 Maintainability and Adaptability

This section contains the non-functional requirements regarding the product's maintainability and adaptability. The below defined requirements should make sure that the extension is well maintainable.

NFR-5: Modifiability

Business Goal:	The CUTE extension should be designed in a manner that it is easily extendable and modifiable. After changes or extensions have been made to the project, there should be a possibility to ensure that everything still works as intended and none of the existing functionality has been compromised.
Scenario:	After a small change in a component, errors occur when using functionality that was working before. These errors make the CUTE extension unusable.
Reaction:	Tests are executed automatically after changes were made to the CUTE extension project. These tests should guarantee that no breaking changes were made during the modification. These tests should always be triggered from the continuous integration pipeline on GitLab [13].
Measure:	Automatic tests allow it to check if a modification has broken some existing functionality before this modification is merged back into the master branch. No modification that has a negative impact on existing functionality should be merged into the master branch. The code coverage should be at least 80%. In addition, code reviews in the form of a pull request should be required before being able to merge some changes back into the master branch.

NFR-6: Modularity

Business Goal:	The CUTE extension should be designed and implemented in a modular way such that single components can be easily exchanged without having to change any other components. If for example the result format should be changed from the currently used XML format to something else, then only the result analyzer component should have to be exchanged without any further code changes needed. The same principle needs to hold if the extension should be extended for example by an additional language server provider.
Scenario:	The change from XML based test results to JSON based results requires a reimplementaion of the test execution logic.
Reaction:	The components from which the extension is built are clearly separated as independent as possible. The coupling throughout the project should be kept as low as possible. There is a clear separation of responsibilities between the components.
Measure:	<p>The modularity of the project should be checked via the Sonarqube [14] code metrics. This should ensure that each component can be exchanged without having to change any unrelated code or components. The functionality of such a replacement should have no influence on the functioning of other components in the application. The following Sonarqube metrics were defined and should be followed:</p> <ul style="list-style-type: none">• Duplicated Lines: Max. 5%• Maintainability Rating: Min. A• Reliability Rating: Min. A• Test-Coverage: Min. 80%

2.4 System context

The following listing shows all the parts that the CUTE extension for Visual Studio Code context consists of.

- Compiler (GCC [15], Clang [16], MSVC [17])
- CMake [9]
- Visual Studio Code [4]
- CUTE Framework [10]
- Language Servers [18] (Clangd [19], CppTools [20])

Even though the assignment allows it to modify the CUTE framework for example for some additional features, the backwards compatibility needs to be preserved. No changes can be made that lead to the breaking of already existing and currently working projects.

2.5 Interfaces

As already mentioned in the context section, the main interface, this CUTE extension for Visual Studio Code has to work against is the CUTE framework [10]. More specifically the CUTE extension works against compiled executables that contain CUTE tests. To run all tests contained within an executable, it is sufficient to simply start the executable without any parameters. If only a specific test from the executable should be executed, this can be specified via the following arguments:

```
1 #!/if the test does not belong to a test suite  
2 Test.exe TestCase  
3  
4 #!/if the test belongs to a test suite  
5 Test.exe "TestSuite#TestCase"
```

Listing 1: CUTE Executable Interface

The test results returned by a CUTE executable depend on the configuration of the CUTE runner within the test project. Either the result can be retrieved over the standard output of the executable or in an XML file which is written to the current working directory of the environment that started the CUTE test executable. The format of the standard output result can be seen in the next listing:

```

1  #beginning TestSuite 2
2
3  #starting TestCase1
4
5  #success TestCase1 OK
6
7  #starting TestCase2
8
9  #failure TestCase2 /path/test.cpp:28 TestCase2: Error Message
10
11 #ending TestSuite

```

Listing 2: CUTE Executable StdOut Result

The format of the XML result for the same tests can be found in the listing below. The thereby produced XML result format has some similar to the JUnit XML [21] result format.

```

1  <testsuites>
2    <testsuite name="TestSuite" tests="2">
3      <testcase classname="TestSuite" name="TestCase1"/>
4      <testcase classname="TestSuite" name="TestCase2">
5        <failure message="path/test.cpp:28 TestCase2: Error ↵
6          Message">
7            TestCase2: Error Message
8          </failure>
9      </testcase>
10 </testsuite>
    </testsuites>

```

Listing 3: CUTE Executable XML Result

2.6 Additional Constraints

It is allowed to extend or adapt the CUTE testing framework [10] if necessary. These changes first have to be discussed with the project supervisor. The main goal is to always keep the backwards compatibility when implementing such CUTE framework extensions or modifications. Existing projects should not be broken due to any framework change.

3 Domain Analysis

This chapter should provide an overview of the domain model that makes up a CUTE testing environment. Based on the findings of this analysis in a later step model classes for the CUTE extension were defined. These model classes later built the foundation of the whole extensions design, as that design was built around the model classes. This chapter analyzes the domain of the CUTE extension for VS Code. In the first section the problem domain is analyzed and described. In the second part the high-level dynamics of selected use cases are explained. Underneath that domain analysis some system sequence diagrams can be found. These system sequence diagrams should provide a coarse overview of the systems interaction with the users and external components. A more detailed description of the single components and the interaction between them within the extension can be found in the chapter Design.

3.1 Structure Diagram - CUTE testing

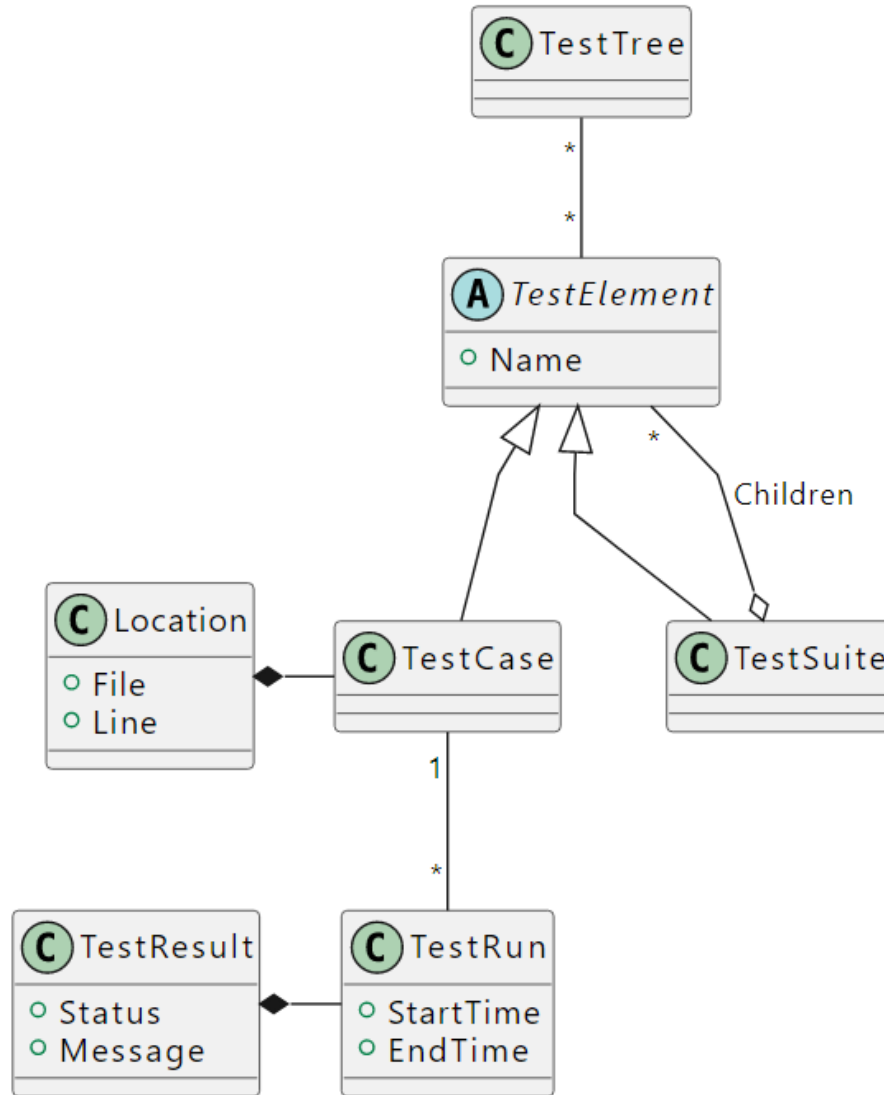


Figure 12: Structure diagram

The above shown diagram includes the domain classes of the CUTE extension for Visual Studio Code. More specifically it contains the model classes that the CUTE framework testing logic, which is required to cover the use cases towards this VS Code extension. The testing logic is based on the **TestTree** object, that can contain multiple **TestElement** objects. From this **TestTree** object the visualizations in the test explorer will later be built. All tests of a CUTE project need to be registered in the **TestTree** object, either directly or contained in a **TestSuite** object. The **TestSuite** – **TestCase** relationship was designed using the composite pattern [62]. Both these classes are derived from the abstract **TestElement** class, what later should simplify the sharing of common functionality. The **TestSuite** owns a list of such **TestElements** as its children. This allows the nesting of test suites and test cases in an arbitrary depth. Further each **TestCase** owns a **Location** object, that contains the file and line information of the test's implementation. Each **TestCase** can have multiple **TestRuns** with the corresponding **TestResult** of that specific run.

3.2 System Sequence Diagram

This section contains an explanation of the dynamics of some selected use cases. Thereby a sequence diagram was created for each selected use case. For each of these diagrams an explanation is available.

3.2.1 UC-1: Navigate To Test Case

The below shown sequence diagram visualizes the flow through the system that gets triggered if a user opens the test explorer and decides to navigate to a specific test case. Before the user can decide to which test case he wants to jump, the project test cases need to be loaded into the test explorer. The test loading gets triggered by the test explorer. After the CUTE extension gets triggered to reload the tests it either does that directly, using the test executable, or via language server information that it receives from a code analyzer component. After the tests are loaded into the test explorer, the user can choose to which test implementation he wants to navigate. This user action in the test explorer notifies Visual Studio Code to open a certain location within a specific file.

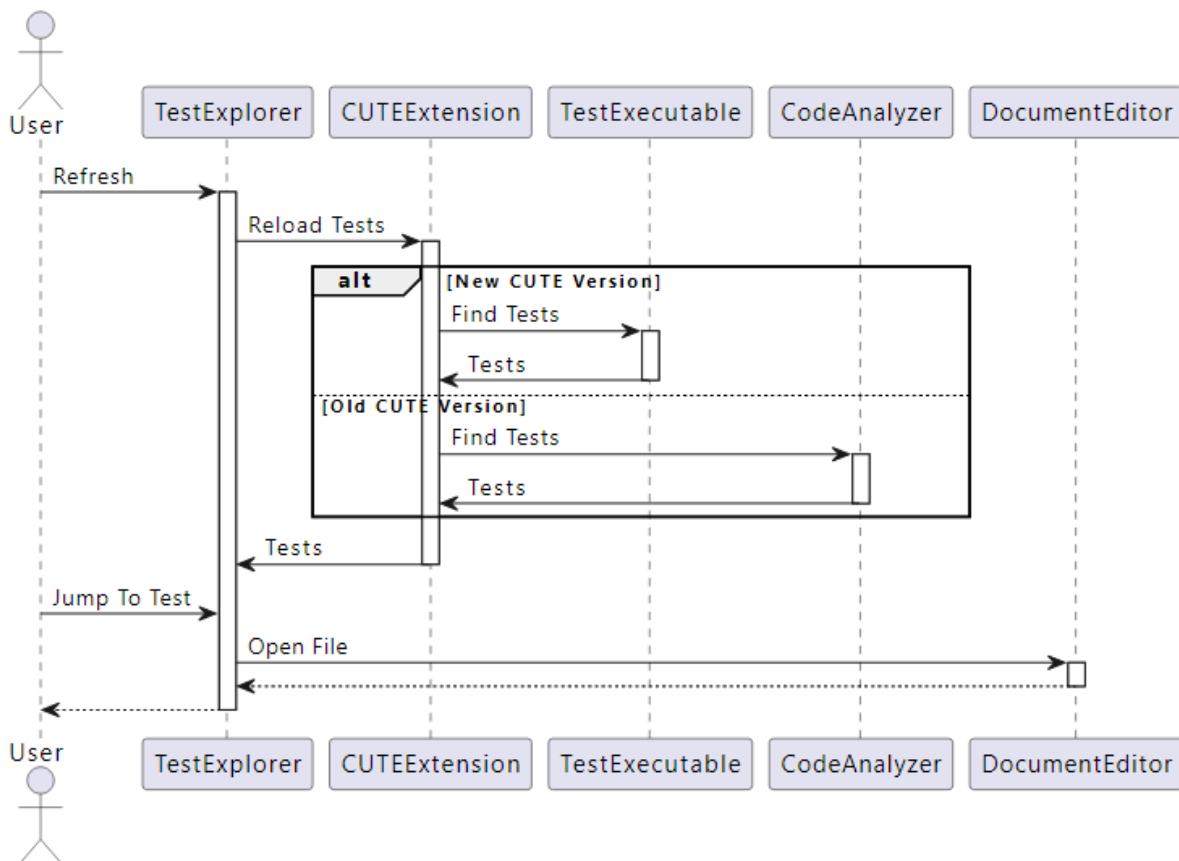


Figure 13: Test discovery sequence diagram

3.2.2 UC-2: Run Tests & UC-4: Analyze TestResult

The below shown system sequence diagram visualizes the flow through the system which gets started when a user chooses to start the execution of some tests. Before he is able to do that in the test explorer, the tests need to be discovered. This discovery process works in the same way as described in the system sequence diagram above. Due to reasons of readability this refreshing of the test explorer is not included in the sequence diagram below. After the users decided which tests he wants to run, the extension starts the test executable with the corresponding start arguments. After these test executables have finished a test run, the CUTE extension analyzes the result and stores that result in a usable way. The user can then get a diff viewer for assertion failures to get a better understanding of why a test failed.

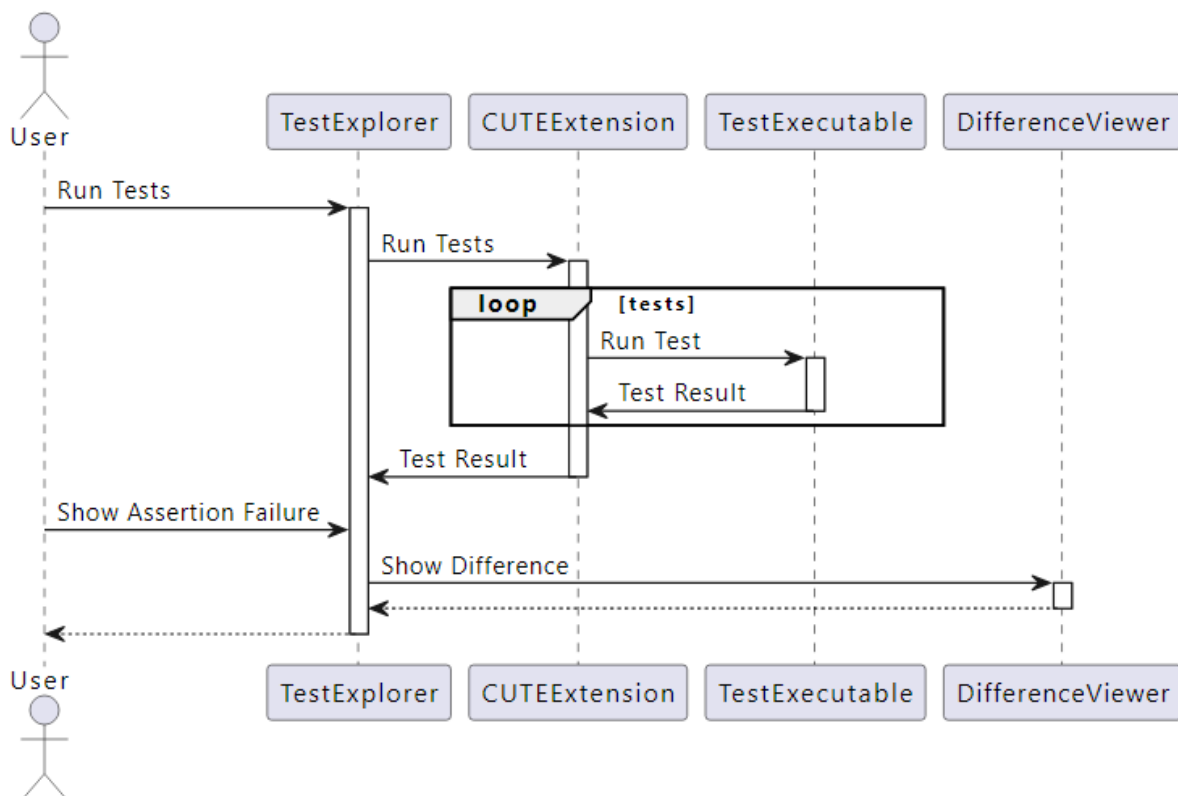


Figure 14: Test run sequence diagram

4 Decisions

In this chapter, the decisions made during this project are documented. These fundamental decisions set the framework in which the CUTE extension was built and therefore had a big influence on the architecture and design of the Visual Studio Code extension that was built in the context of this bachelor thesis. For each decision that was made, the different available options are explained, and the final decision is justified over the following pages.

4.1 Fundamental Architecture Decision

The first decision that had to be made, was the one whether to build the CUTE extension from scratch or rather use already existing extensions, which provide some useful basic functionality, as a baseline for the project. As examined during the risk analysis, this decision states a major risk to the project's success. Therefore, many options were evaluated through the analysis of existing testing extensions for Visual Studio Code.

4.1.1 Build the extension from scratch

Building the CUTE extension from scratch provides full flexibility in terms of technology and design. On the other hand, this approach involves a bigger amount of UI planning and implementation which requires a large amount of usability testing. To ensure a user experience that requires as little learning as possible the controls need to be similar to existing and well-known testing tools like different Visual Studio Code testing extensions or also the Eclipse JUnit tools [22]. In the end, the produced solution would have to be close to existing solutions from a user experience point of view. Therefore, the question whether it would make sense to build on already existing functionality and especially user interfaces, is justified.

Pros

- Full flexibility
- No constraints regarding architecture and design

Cons

- Additional effort required for usability testing
- Higher risk of not achieving the NFR regarding the usability

4.1.2 Test Explorer UI Extension

The Test Explorer UI [23] provides an extensible user interface for running tests in Visual Studio Code. This extension offers easy options to add support for a custom testing framework, such as CUTE in this case. There is an extensible list of existing test adapters [27] available, which could be used as inspiration for the CUTE test adapter. There are also templates and instructions available on how to set up such a custom test adapter for the Test Explorer UI extension. The main benefit of using the Test Explorer UI extension is consistency in the user interface area, as this extension is widely used and accepted by developers. The implementation of a test adapter for CUTE tests includes all the logic of discovering tests, running tests, and analyzing the result for each test run. The further evaluation revealed that this extension is deprecated since July 2021 and that there is no plan to add further major features.

Pros

- Proven user experience
- Neat integration into VS Code
- Wide range of example test adapters available [24]
- Templates and instructions available [25]

Cons

- Deprecated since July 2021

For C++ the following test adapters are currently available:

- | | |
|--------------------------------------|-------------------------------|
| • C++ TestMate [40] | • CppUTest Test Explorer [32] |
| • CMake Test Explorer [28] | • Boost.Test Explorer [33] |
| • CppUnitTestFramework Explorer [29] | • Acutest Test Explorer [34] |
| • Bandit Test Explorer [30] | • CppUnit Test Explorer [35] |
| • catkin-tools [31] | |

4.1.3 VS Code Testing API

With version 1.59 Visual Studio Code [36] introduced the proprietary Testing API [37] in July 2021. This Testing API offers interfaces to publish and run tests and show their run results in a visualized format. This API offers the users the possibility to run tests from the Test Explorer view, from decorations within the source code and from commands. Further this Testing API offers richer displays of outputs and diffs. Testing extensions for specific testing frameworks now can simply code against the offered API and profit from the offered functionality and user interfaces. There are official examples and tutorials available on how to create a custom testing extension that works against the official new Testing API. This API is the reason that the earlier described Test Explorer UI is deprecated. There are official migration instructions that explain how to adapt existing Test Adapters for the Test Explorer UI in a way that they can be used with the new Testing API.

Pros

- Official Testing API [37]
- Rich user experience with low learning curve
- Example implementations available
- Already adapted by some of the above-mentioned Test Adapters

Cons

4.1.4 C++ TestMate Extension

C++ TestMate [39] is a widely used C++ testing extension for Visual Studio Code that provides support for GoogleTest [42], Catch2 [41] and DOCTests [43]. Further it provides some basic support for Google Benchmark [44] and there is an open pull request including the implementation for CppUTest [45] support. C++ TestMate is one of the earlier mentioned testing extensions that was converted from the Test Explorer UI to the official VS Code Testing API. Using the open pull request for the CppUTest support a similar functionality could be implemented to support the CUTE testing framework. In addition to the benefits of using the pure Testing API as basis, this approach would offer further configuration possibilities and preconfigured parts such as the test debugging support for multiple debugging extensions. On the other hand, the CppUTest support pull request is open for a while now and no real progress can be seen what makes it unclear if and when it will be merged and officially be available.

Pros

- Additional configuration possibilities
- Preconfigured functionality and extension support
- Example available on how to integrate an additional testing framework

Cons

- No control over the integration
- Unexpected delays and problems possible
- Depending on third party extension

4.1.5 Decision

Based on the above visible evaluation of the different approaches to implement the CUTE extension, the decision was made to use the official VS Code Testing API. The functionality that needs to be implemented using this approach can be in some ways similar to the implementation of the C++ TestMate extension, that now also directly builds on the Testing API. Further information on the final implementation can be found in the chapters Design or Implementation.

4.2 Test Discovery

One of the functionalities that need to be implemented when using the official Testing API, is the test discovery. The CUTE tests need to be discovered somehow and registered in a `vscode.TestController` [46] instance afterwards in order to make use of the test explorer functionality. The tests and suites need to be registered as `vscode.TestItem` [47]. In order to make the full testing experience available, these `vscode.TestItem` objects need an id, a file uri and a range within that file. The different approaches to discover the test cases and test suites with the required properties are discussed in the following sections.

4.2.1 Executable Based

The analysis of the C++ TestMate extension revealed that it uses the executables to discover test cases and test suites. All the there supported testing frameworks produce executables that offer the possibility to query all the contained tests, suites, and their corresponding locations. The TestMate extension checks which executable in the output directory (configurable setting) contains what kind of tests by starting the executable with a `-h` or `--help` argument. The result then reveals the executable contains tests and if yes which testing framework they are based on. After the test executables are found they are started again with the framework dependent flag to list the contained tests. Below a part the identification logic used in the TestMate extension can be seen.

```

1  const runWithHelpRes = await this._shared.taskPool.scheduleTask(async () => {
2      if (checkIsNativeExecutable) await c2fs.isNativeExecutableAsync(this._
      _execPath);
3      return this._spawner.spawnAsync(this._execPath, ['--help'], this._
      _execOptions, this._shared.execParsingTimeout);
4  });
5
6  const frameworkDatas: Record<
7  FrameworkId,
8  Readonly<{
9      priority: number;
10     regex: RegExp;
11     create: (sharedVarOfExec: SharedVarOfExec, match: RegExpMatchArray) =>
      AbstractExecutable;
12 }>> = {
13     catch2: {
14         'priority': 10,
15         regex: '/Catch2? v(\d+)\.(\d+)\.(\d+)\s?/',
16         create: (sharedVarOfExec: SharedVarOfExec, match: RegExpMatchArray) =>
17             new Catch2Executable(sharedVarOfExec, parseVersion123(match)),
18     },
19     'gtest': {
20         priority: 20,
21         regex:
22             '/This program contains tests written using .*(--(\w+)list_tests.*List
the names of all tests instead of running them/s',
23         create: (sharedVarOfExec: SharedVarOfExec, match: RegExpMatchArray) =>
24             new GoogleTestExecutable(sharedVarOfExec, match[1] ?? 'gtest_'),
25     },
26     'doctest': {
27         priority: 30,
28         regex: '/doctest version is "(\d+)\.(\d+)\.(\d+)"/',
29         create: (sharedVarOfExec: SharedVarOfExec, match: RegExpMatchArray) =>
30             new DOCExecutable(sharedVarOfExec, parseVersion123(match)),
31     },
32     'gbenchmark': {
33         priority: 40,
34         regex: '/benchmark \[--benchmark_list_tests=\{true\\|false\\}\]/',
35         create: (sharedVarOfExec: SharedVarOfExec) => new
GoogleBenchmarkExecutable(sharedVarOfExec),
36     },
37     'google-insider': {
38         priority: 50,
39         regex: '/Try --helpfull to get a list of all flags./',
40         create: (sharedVarOfExec: SharedVarOfExec) => new GoogleTestExecutable(
sharedVarOfExec, 'gunit_'),
41     },
42 };

```

Listing 4: Test Executable Identification [48]

The following code snippet shows the discovery of GoogleTest test cases as it is done by C++ TestMate. Thereby the executable is called again with the argument to list the contained tests. This call produces an XML file containing the tests and suites with their locations within the project.

```

1 const args = this.shared.prependTestListingArgs.concat([
2   `--${this._argumentPrefix}list_tests`,
3   `--${this._argumentPrefix}output=xml:${cacheFile}`,
4   ]);
5
6   this.shared.log.info('discovering tests', this.shared.path, args, this.
7   shared.options.cwd);
8   const googleTestListProcess = await this.shared.spawner.spawn(this.shared.
9   path, args, this.shared.options);
10  const loadFromFileIfHas = async (): Promise<boolean> => {
11    const hasXmlFile = await promisify(fs.exists)(cacheFile);
12    if (hasXmlFile) {
13      const xmlStream = fs.createReadStream(cacheFile, 'utf8');
14      await this._reloadFromXml(xmlStream, cancellationToken);
15    }
16  }

```

Listing 5: Executable based test discovery [49]

As the CUTE framework does not offer any help flag and no options to list the tests with their location, this approach requires some changes to the framework itself. Two additional flags have to be added and a solution to find a tests location needs to be discovered as CUTE currently does not keep the locations information of a test.

Pros

- Most reliable approach as the infos directly come from the executable
- Most performant approach as no additional information is required
- Chosen approach in C++ TestMate extension

Cons

- Requires changes of the CUTE framework
- Uncertainty that the location information can be gathered without breaking backward compatibility

4.2.2 Code Based

A different approach to discover test cases and their location is to use language information provided by some language server provider. This approach does not require any changes of the CUTE framework, as it is purely based on the source code. To discover test cases for example, all the references to the CUTE macro definition in the `cute_test.h` header file could be searched. An option to make the discovery more reliable is to also include the constructor calls of the test struct that is also defined in the `cute_test.h` header. The code snippet below shows the CUTE macro and the two constructors for which the references need to be found.

```
1 #define CUTE(name) cute::test((&name), (#name))
2
3
4 template<typename VoidFunctor>
5 test(VoidFunctor const & t, std::string sname = demangle(typeid(VoidFunctor).name()))
6     : name_(sname)
7     , theTest(t)
8 {
9 }
10
11 // separate overload to allow nicer C++11 initializers with {"name",lambda}
12 template<typename VoidFunctor>
13 test(std::string sname, VoidFunctor const & t)
14     : name_(sname)
15     , theTest(t)
16 {
17 }
```

Listing 6: `cute_test.h` test instantiation

Pros

- Does not require any CUTE framework changes
- Definitely keeps backwards compatibility and works on existing projects

Cons

- Slower as the information needs to be searched and combined (e.g. references)
- Depending on Language Server Provider implementations
- Uncertainty that all required language information is available

4.2.3 Decision

After prototypes for both the above-mentioned approaches showed promising results, the decision was made to set the executable based approach as preferred option but to also implement the code-based approach as a backup solution for older CUTE projects and executables. This decision was made with prior consultation of the project supervisor. The success of both prototypes has minimized the risks R1 and R2 massively as it was clear now, that all the required functionalities on both the CUTE framework and the Visual Studio Code side can be implemented. Further details on the exact implementation of both approaches can be found in the chapters Design and Implementation.

4.3 Result Format

Besides the test discovery also the test result analysis needs to be implemented, when using the official Testing API. The CUTE framework offers different result output options. One offered possibility is to write the test results into the standard output of the process that runs the test executable. Another option available is the output of the result as XML file. If this second approach is chosen, the test executable writes the result file to the current working directory. The desired option can be chosen by configuring the CUTE test runner accordingly. In the next sections, both approaches will be evaluated, and the final decision will be justified based on the evaluation results afterwards.

4.3.1 Executable StdOut Result

The writing of the result into the standard output is the simplest approach, as it does not involve any file system interactions. Using this approach, the CUTE extension could simply start the executable, then wait for the output and finally parse that to check the test outcome. This approach could lead to problems when running tests from the same executable in parallel. The following code listing shows the result format on the console for a successful and for a failed test run.

```
1      #beginning TestSuite 1
2
3      #starting TestCase1
4
5      #success TestCase1 OK
6
7      #ending TestSuite
```

Listing 7: CUTE Executable StdOut Success

```
1      #beginning TestSuite 1
2
3      #starting TestCase2
4
5      #failure TestCase2 /path/test.cpp:28 TestCase2: Error Message
6
7      #ending TestSuite
```

Listing 8: CUTE Executable StdOut Failure

Pros

- Easy to implement
- No file system interaction required

Cons

- Possibility of problems when executing multiple tests in parallel
- Information lost after the process is finished

4.3.2 File Based XML Result

Writing the result into a file in the XML format, that is somewhat similar to the JUnit XML result format has the advantage, that there are already existing parsers [51] available to extract the required information from result file. Using this approach, the CUTE extension needs to do an additional step of reading the result file from the file system. This approach easily allows multiple test runs happening at the same time, as the results of each run can be stored into a different file or folder. If a problem should occur during the result parsing, the information will not be lost, what helps with the nonfunctional requirement regarding maturity. Using the results from the file, it is possible to reproduce the problem and fix it based on the findings. The listing below shows results of a successful and a failed test run in the XML format.

```
1      <testsuites>
2          <testsuite name="TestSuite" tests="1">
3              <testcase classname="TestSuite" name="TestCase1"/>
4          </testsuite>
5      </testsuites>
```

Listing 9: CUTE Executable XML Result Success

```
1      <testsuites>
2          <testsuite name="TestSuite" tests="1">
3              <testcase classname="TestSuite" name="TestCase2">
4                  <failure message="path/test.cpp:28 TestCase2: Error↵
5                      Message">
6                      TestCase2: Error Message
7                  </failure>
8              </testcase>
9          </testsuite>
      </testsuites>
```

Listing 10: CUTE Executable XML Result Failure

Pros

- Easily parallelizable
- Existing XML parser available
- No information loss if the result parsing failed

Cons

- Additional steps required to write and read the result to and from the file

4.3.3 Decision

Based on the findings from the evaluation, the decision was made to use the file-based approach, that stores the results in XML format in the file system. This approach allows easier parallelization of test runs and better stability in the case of a parsing problem, as the information stay available, and the problem could be reproduced.

4.4 Language Server Provider

To implement the code-based test discovery and some further convenience tools such as warnings of unregistered tests, a language information provider is needed. Visual Studio Code offers multiple such language information provider for C++ [52] in the form of extensions, that contain language servers and their matching client. The language servers should provide the functionality to find document symbols, find references throughout a workspace for a specific symbol and find the definition of a method in order to be able to support the CUTE extension use cases. The evaluation of the most popular language server providers for C++ [52] can be found in the next sections.

4.4.1 CppTools - C/C++ for Visual Studio Code

The CppTools extension is maintained by Microsoft and adds language support for C and C++ to Visual Studio Code. Besides the IntelliSense [53] features, that are mostly based on language server information, this extension offers additional features such as debugging support. The extension comes with a language server implementation that supports all before mentioned use cases. Besides the language server, that runs in a separate process, the extension also includes a client that communicates with the server via the language server protocol (LSP). The client registers the server's capabilities to the built in Visual Studio Commands, which can be called from within a workspace.

The evaluation in the form of a prototype has shown that it takes some time for the extension to find references. The known problem of a never terminating request if references for a symbol are searched and the file containing the symbol is not opened in the editor, could also be observed. This language server does not offer the possibility to find the references of an operator overload.

Pros

- Probably most widely used language server provider for C++
- Easy to install and configure
- Well documented and supported
- Supports all required features (Document Symbols, Reference, Definitions)

Cons

- Moderate performance when searching references
- No support for operator references
- Can get stuck if source files are not opened in editor

4.4.2 Clangd

The clangd extension [19] is maintained by the LLVM-Project [56] and provides language support for C++ code. The clangd language server can be easily integrated into a variety of source code editors. Clangd offers features such as cross-reference finding, that is also used for the definition and declaration discovery, navigation and symbol hover information providing and much more. Even though it is not well documented, clangd also supports Document Symbol discovery. The communication between the clangd language server and the client contained in the clangd extension is based on the language server protocol (LSP) as well. The features are also registered to the built in Visual Studio Code commands.

The evaluation using a prototype has shown that clangd is much faster in providing reference information than the CppTool language server. The discovered Document Symbols do not include macros as they are used for the CUTE test definition. Apart from that all required features are supported.

Pros

- Widely used language server provider for C++ not just in VS Code
- Easy to install and configure
- Supports all required features (Document Symbols, Reference, Definitions)
- Good performance for all operations (References, Symbols and Definitions)

Cons

- Does not list macro definitions in the document Symbols

4.4.3 Decision

Both evaluated options offer the required functionality and each of them has some advantages. For this reason, the decision has been made to support both of them in the CUTE extension. Setting the extension up to support multiple language server also supports the nonfunctional requirement regarding modifiability and extendibility, as with the required design also further language server provider could be supported in the future without having to change any non-related components. This decision was taken after a consultation with the project supervisor.

4.5 Language Server Integration

To integrate the earlier evaluated language servers there are different approaches available. It needs to be kept in mind, that the decision was made to support multiple different language server provider and that the solution should be built in a way that in the future even more language server provider could be supported. All the supported language servers should communicate using the language server protocol (LSP) [18] over JSON-RPC [57]. The LSP was designed to unify the communication between a source code editor and the different implementations. In the following sections the different options are evaluated, whereafter a decision for an approach was made.

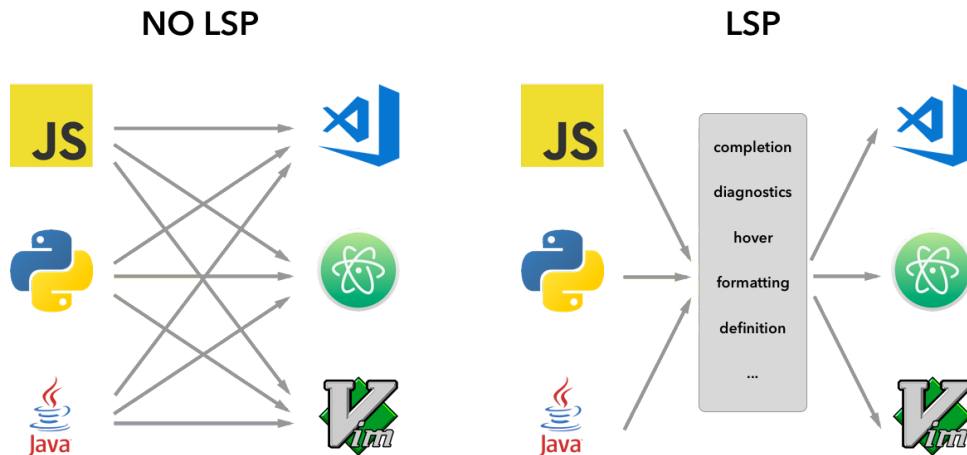


Figure 15: LSP Comparison [54]

4.5.1 Custom Language Server Client

The first approach that was evaluated in the form of a prototype was the implementation of a custom language server client implementation. There are examples and tutorials available that explain how to implement a client for Visual Studio Code that communicates with a language server using the LSP. The objective of this approach was to write a client that can be used for both supported language servers.

The implementation of the prototype revealed that such a client can't be implemented that easily, as the messages differ from provider to provider. A further difficulty that occurred was the nature of the protocol itself. As the LSP is based on JSON-RPC [57] calls adding a client to a running server turned out to be quite difficult. The tutorials, which are available, all include a custom language server implementation. For each client such a language server instance will be started and at the startup the communication channel is configured. No working client, that allowed communication with either a running clangd server or a running CppTools server, could be implemented.

Pros

- Full control over the communication
- Only has to support the required features

Cons

- Difficult configuration
- No working solution found

4.5.2 Use Built-in Visual Studio Code Commands

The second approach, for which the prototype was adapted, is based on the built-in Visual Studio Code commands [55]. VS Code offers some built-in commands that aggregate the functionality of different implementations. These built-in commands also include some commands, which are targeted towards language server providers. Using this approach most of the code used can be shared between the different language server implementations. Only cases where they return the results in slightly different formats need to be addressed individually. The evaluation showed that all required features can be addressed using these built in VS Code commands. The following listing shows the used commands that are offered by Visual Studio Code together with their corresponding parameters.

vscode.executeReferenceProvider - Execute all reference providers.

- **uri** - Uri of a text document
- **position** - A position in a text document
- *returns* - A promise that resolves to an array of Location-instances.

vscode.executeImplementationProvider - Execute all implementation providers.

- **uri** - Uri of a text document
- **position** - A position in a text document
- *returns* - A promise that resolves to an array of Location or LocationLink instances.

vscode.executeDefinitionProvider - Execute all definition providers.

- **uri** - Uri of a text document
- **position** - A position in a text document
- *returns* - A promise that resolves to an array of Location or LocationLink instances.

vscode.executeDocumentSymbolProvider - Execute document symbol provider.

- **uri** - Uri of a text document
- *returns* - A promise that resolves to an array of SymbolInformation and DocumentSymbol instances.

Pros

- Access to all required language server functionalities
- A lot of shared code between used language servers
- Easy to use and configure
- Supports the nonfunctional requirements regarding maintainability and modifiability / extensibility

Cons

- Some specific functionalities not directly available

4.5.3 Use Proprietary Language Server Provider Commands

The third approach, that was evaluated using a prototype, was using proprietary commands and the corresponding language server client implementations of each language server provider. This approach quickly delivered some promising results as it allowed it to communicate with the language servers for the first time. The prototypes were able to get the references for specific symbols and also their definition and declaration positions. The downside of this approach was that almost no logic could be shared between the different supported language server providers. Choosing this approach would have compromised the nonfunctional requirement of maintainability as this option would have led to a lot of very provider specific code and probably also some duplications.

Pros

- Access to the full functionality of each language server implementation
- Delivered the expected results

Cons

- Difficult to maintain
- A lot of very specific code for each implementation
- Code sharing between the components very difficult

4.5.4 Decision

The evaluation clearly shows that the second approach using the built-in Visual Studio Code commands is the only reasonable. The first approach is technically only very difficult or not viable at all and the third option leads to very much provider specific code which decreases the maintainability of the CUTE extension drastically. If the implementation shows that some more specific functionalities are needed, which are not offered through the built-in commands, these can be included the third approach. The main functionality can easily be covered by the built-in VS Code commands in a very elegant way.

4.6 Debugging Libraries

The decision for the supported debugging libraries is based on the analysis of the TestMate testing extension. During the analysis of that C++ testing extension, it turned out that TestMate supports the following three debugging extensions:

- CppTools - C/C++ for Visual Studio Code [20]
- Native Debug [58]
- CodeLLDB [59]

A further analysis of these three debugging extensions revealed that they all support a somewhat similar functionality with only minor differences. Therefore, the decision was made to support the same debugging extensions as well in the CUTE testing extension. This decision also supports the nonfunctional requirement regarding modifiability and extendibility as the design required to support all three debugging extensions, will automatically be extensible for further debugging extensions.

4.7 MSYS2

During the development it was noticed that the handling with the clangd extension needs the `compile_commands.json`. But the JSON file is not created by CMake in connection with MSVC [17]. Only the generators "Unix Makefile" and "Ninja" [88] are able to create this file and write it back into the file system. As the Clangd extension offers remarkable performance a way should be found that allows it to use Clangd with a compatible compiler under windows. The decision was made to set up a Windows build system with Clang [16] and Make for Windows. So far this worked well for development. However, when installing on a freshly installed Windows, it turned out that Clang does not easily work without MSVC [17]. Although a generation of the required JSON with Make is possible, the underlying MSVC components are still needed as Clang does not provide a required standard library. This last dependency should also be removed in order to allow a simple installation and usage of the CUTE plugin. The decision was based on the fact that to use CUTE in connection with Cevelp no MSVC installation is required either. In combination with the CUTE plug-in for Cevelp MinGW [89] is used. MinGW provides the includes, libraries and the runtime. MSYS2 was chosen to configure MinGW for the CUTE extension for VS Code's needs. The configuration needs to make sure that the build process works correctly and the required `compile_commands.json` is generated. With the help of MSYS2 the MinGW environment can be configured in a way that the compiler (Gcc [15]), CMake [9] and Clangd [19] are available and can be used by the extension. It is important here that the binary directory of the MinGW environment is contained in the PATH variable of the system.

4.8 User Guide

For the user guides and tutorials, the decision was made to base them in an audio-visual format. Therefore the installation process and demonstrations of features are captured on video. The videos cover the entire installation and demonstrate the usage of the entire feature set. A video brings the content in a more understandable way to the user and visually represents what needs to be done. In an instruction text many pictures would be needed to achieve the same results, which at a certain size is equivalent to a video. The videos are uploaded on the "CUTE Test" channel on YouTube [90].

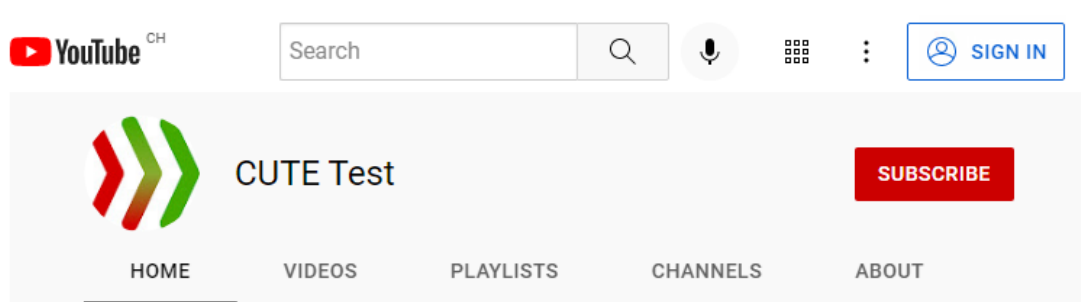


Figure 16: CUTE testing on YouTube [90]

5 Design

In this chapter the design of the CUTE extension is going to be described and explained. For this design and architecture documentation the c4model [60] will be loosely implemented. At first there will be a general context overview of the extension with its external dependencies and integration possibilities into Visual Studio Code. Afterwards there is going to be a high-level overview of the containers used for the implementation of this extension. This is followed by a more detailed description of the single components and their interactions. At the end of this chapter a detailed code overview on class level can be found. To further clarify the complex flows through the CUTE extension some sequence diagrams will be available. These sequence diagrams should improve the understanding of the more complex workflows by visualizing them.

5.1 Architecture overview

In this section of the design documentation chapter a general overview of the context in which the CUTE extension is acting will be provided.

5.1.1 CUTE Extension

The diagram below shows the external dependencies of the CUTE extension.

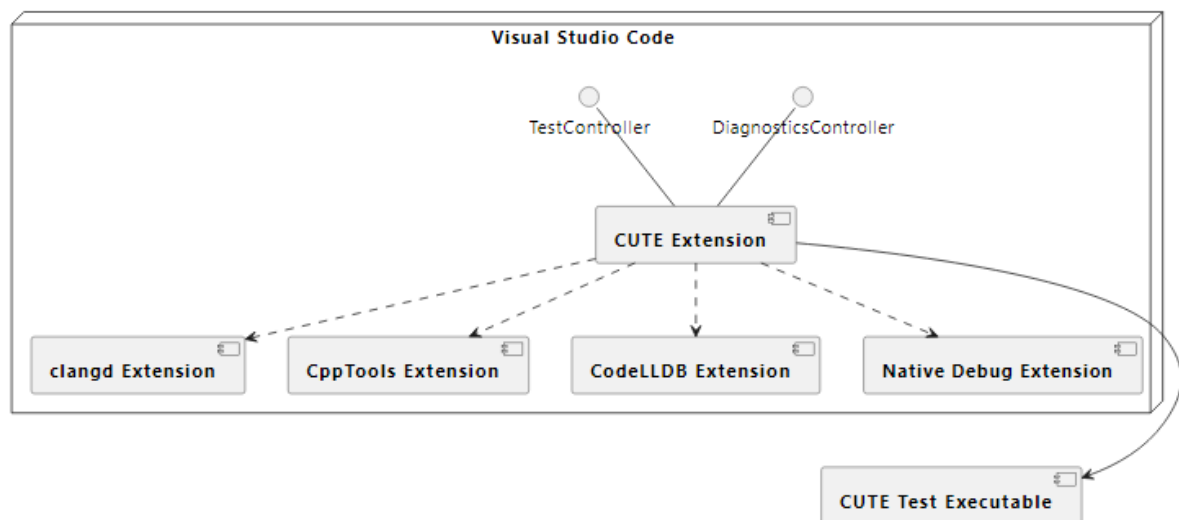


Figure 17: Extension context diagram

The CUTE extension has three sorts of dependencies.

(i) **Language Feature Provider**

The first kind of dependency includes extensions that provide language feature support for C++ code. This language feature support is used to discover test cases, test suites and test runners amongst providing further features that allow it to support users while writing CUTE tests. The first kind of dependency includes the CppTools and clangd [19] extensions which will be described later in this chapter. It is recommended that the clangd extension is used as language feature provider as it offers better performance compared to the CppTools [20] extension. If both extensions should be installed on the system, the clangd extension is preferred. In such an environment clangd shows a warning and recommends disabling the IntelliSense features provided by the CppTools extension to prevent interference.

(ii) **Debug Logic Provider**

The second kind of dependency includes extensions that provide debugging capabilities which can be used to debug CUTE test cases. The second category of dependencies includes the CppTools [20], the CodeLLDB [59] and the Native Debug [58] extensions, which will be described later in this chapter. The CUTE extension decides on which extension to use for debugging based on the system environment. If none of the above-mentioned extensions should be installed, the test debugging feature will be disabled. If multiple of these extensions should be installed, the default order of choice is:

- (1) CppTools [20] - C/C++ for Visual Studio Code - Microsoft
- (2) CodeLLDB [59] - Vadim Chugunov
- (3) Native Debug [58] - WebFreak

(iii) **CUTE Test Executables**

The third kind of dependency includes the actual CUTE test executables. These are built from the solution and, when using the newest CUTE [2] version, can be used to discover test cases and their location within the solution. These executables are started and queried by the CUTE extension to retrieve the desired information. This way of discovering test cases and test suites is more performant and reliable than using language server information. Therefore, this is the preferred way of test discovery if the right version of CUTE is used.

5.1.2 Supported Extensions

In this section of the chapter Architecture overview the earlier mentioned extensions which can be used in the context of the CUTE extension will be described. For each of the supported extensions a brief description and its use case in the context of the CUTE extension can be found.

(i) **Clangd [19]**

Clangd is a Visual Studio Code extension that, according to the extension description, helps developers to write, understand and improve C/C++ code by providing the following features via a language server implementation:

- (a) code completion
- (b) compile errors and warnings
- (c) go-to-definition and cross references
- (d) hover information and inlay hints
- (e) include management
- (f) code formatting
- (g) simple refactorings

In the context of the CUTE extension the information provided by the clangd language server is used to discover test cases, test suites and test runners. Further the language server information is used to discover potential problems in the test code like unregistered test cases, uncalled test suites or test declarations using the legacy syntax. The language server information is also used to enable quick fix refactoring options to mitigate the before mentioned test code problems.

The language server information is accessed via built-in Visual Studio Code commands which can be executed via the `vscode.commands.executeCommand` API [61]. The strategy of not directly interacting with the language server or the clangd extension over custom commands allows it to exchange the language server provider without any code changes. The only requirement for the language server provider extension is, that it registers its functionality to the built-in Visual Studio Commands.

(ii) **CppTools - C/C++ for Visual Studio Code [20]**

C/C++ for Visual Studio Code is an extension by Microsoft, that provides language support for C and C++. This support includes features such as debugging capabilities and IntelliSense. For the later there is a language server implementation available. The CppTools extension registers its language server capabilities to the built-in Visual Studio Code commands, which allows it to exchange the earlier mentioned clangd language server implementation by the CppTools language server implementation without any code changes. When both, the clangd extension and the CppTools extension are installed in parallel IntelliSense should be disabled in order to prevent problematic interferences.

It is recommended that the clangd extension with its language server implementation is used together with the CUTE extension as the usage of the CppTools extension with its language server leads to a performance penalty.

Beside its language server features, the CppTools extension offers C++ debugging capabilities, which can be used as an option to debug test cases. The CUTE extension offers the choice between three different extensions which all allow C++ debugging.

(iii) **CodeLLDB [59]**

The CodeLLDB Visual Studio Code extension is a native debugger powered by LLDB [86], that offers the functionality to debug C++ code amongst other languages. The CodeLLDB extension offers features such as:

- (a) Conditional breakpoints, function breakpoints, logpoints
- (b) Hardware data access breakpoints (watchpoints)
- (c) Launch debuggee in integrated or external terminal

CodeLLDB is the second option available to debug CUTE test cases beside the earlier mentioned CppTools extension functionalities. Depending on the extensions installed in a specific Visual Studio Code instance, the CUTE extension selects an available debug provider and configures it accordingly. If none of the supported debugging extensions is installed, the test case debugging feature gets disabled.

(iv) **Native Debug [58]**

The Native Debug Visual Studio Code extension by WebFreak is another native VSCode debugger that supports both GDB [85] and LLDB [86]. This extension offers a wide range of features somewhat similar to the features described in the CodeLLDB or CppTools extension sections. Native Debug is the third option to debug CUTE test cases beside the earlier mentioned extensions that also provide debug functionality. The CUTE extension chooses the debug functionality provider based on the installed extensions in a specific environment.

5.2 CUTE Extension Components

In this section an overview of the CUTE extension is provided on component level. Thereby each of the individual components is described and put in context with its peers within the extension. In addition to the components built specifically for this application, the major Visual Studio Code dependencies will also be mentioned.

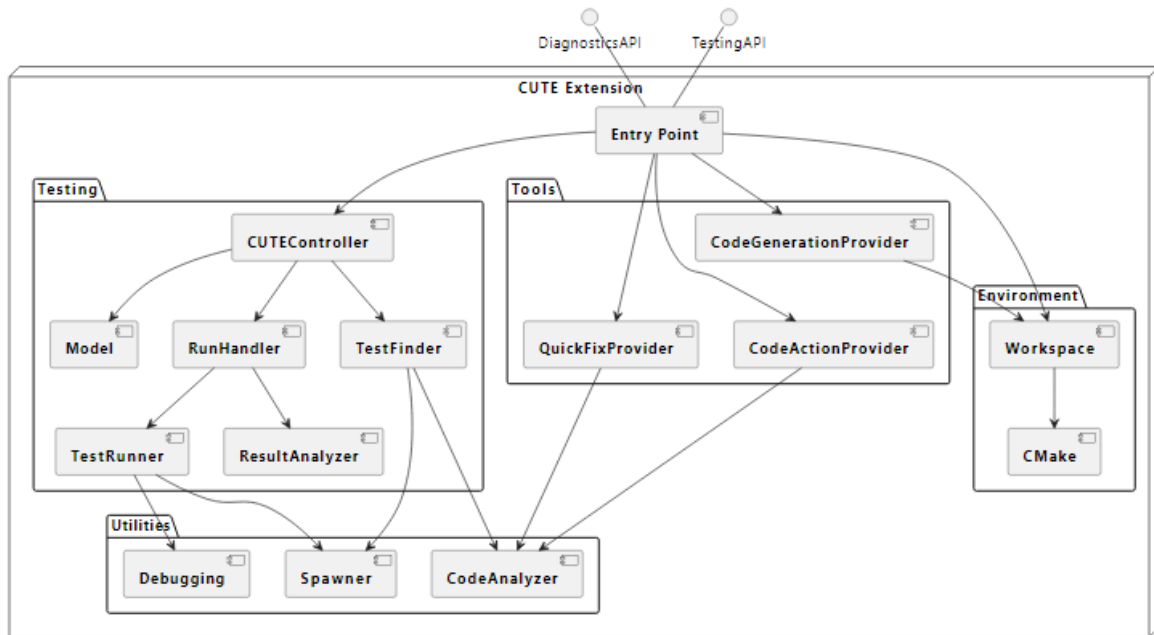


Figure 18: Component diagram

The diagram above shows the components of the CUTE extension split up into the four packages Testing, Tools, Utilities and Environment. The Testing package contains all components which contribute some logic concerning the main testing use cases. This Testing package therefore contains the business logic of the extension. The Tools package contains components which provide additional functionality such as warnings and quick fixes. Beside these convenience tools, this package also contains the logic to generate code parts such as new projects, new test suites or new test cases. The utilities package contains components that provide base functionality which is used by components in the earlier mentioned packages. This base functionality covers the code analysis, the executable spawning, and the configuration of the supported testing extension. The Environment package contains components that can be used to analyze and manage a workspaces structure and the corresponding CMake file. These components build the connection between the workspace and the business logic.

CUTEController - Business Logic Provider This component contains the test controller functionality. The responsibilities of this component include tracking the test elements, providing the run profiles, and triggering test discovery refreshes for CUTE tests. This component contains the main functionality of the business logic and combines all further testing components based on the systems set up. In this component, a `vscode.TestController` is set up and made available for the registration in Visual Studio Code. This `vscode.TestController` is part of the official testing API on which this CUTE testing extension is built.

RunHandler - Handle Test Runs The `RunHandler` component contains the functionality to setup test runs in a specific configuration. These test runs can contain a selection of test elements such as test suites or test cases. Each test run can be configured as either a normal run or a debug run. Each test run is triggered by the `CUTEController` component and its testing API implementations.

TestFinder - Discover Test Elements

The `TestFinder` component contains the functionality to discover test elements such as test suites and test cases. There are two different implementations of this functionality, for which a description is available in the chapter Implementation. One implementation bases the test discovery on the CUTE test executables themselves and therefore requires the functionality defined in the `Spawner` component. The other implementation is based on code information, which is provided by the `CodeAnalyzer` component.

Model - Domain Classes

The `Model` component contains the domain classes which were defined during the domain analysis. The business logic regarding the main testing use cases is based on the classes defined in this component. This functionality was designed in a domain driven fashion.

ResultAnalyzer - Analyze Test results

This component contains the functionality to analyze the outcome of a test. There are multiple implementations possible based on the chosen result format. The implementation for the XML based CUTE results is described in the chapter Implementation.

TestRunner - Run Tests

The `TestRunner` component contains the functionality to execute individual test cases in a specific configuration. The execution of CUTE test cases is based on the test executables, which are started during a test run using the functionality provided by the `Spawner` component. The `TestRunner` component returns an unanalyzed test result which can be passed to the `ResultAnalyzer` component afterwards to check the test outcome.

QuickFixProvider - Mitigate Code Problems

The QuickFixProvider component contains functionality to mitigate code warnings using quick fixes. To allow the implementation of more complex quick fix commands that require some code information, this component has a reference to the CodeAnalyzer component. The quick fix functionality provided by this component is registered in the extension context by the EntryPoint component.

CodeActionProvider - Display Code Problems

The CodeActionProvider component contains functionality to analyze the test code and show potential problems to the users. This component uses the functionality of the CodeAnalyzer Component to analyze the test code and provide warnings in the following cases:

- Unregistered test cases – this test will never be executed
- Unregistered TestSuite – this suite is not used and will none of its tests will be executed
- Legacy Syntax – the old CUTE macro is used to define a test and therefore the extension cannot provide the full experience.

These problems can be mitigated using the quick fixes provided by the QuickFixProvider component. The CodeActionProvider component creates and configures a `vscode.DiagnosticCollection` [64] which is registered to the extension context by the EntryPoint component in order to make the problems visible in the VS Code editor.

CodeGenerationProvider - Generate Test Code

The CodeGenerationProvider component includes functionality to create test code. The code that can be generated is either a new test case, a new test suite or a new test project. The test case generation is snippet based, which means that the editor provides this option as soon as a user starts to type `TEST(...)`. To generate a new test suite file, a command, which allows entering a suite name, can be used over the VS Code commands API. The project generation is template based and can also be triggered using a command over the Visual Studio Code commands API.

Debugging - Configure Debugging Extensions

The Debugging component includes the logic to configure the supported debugging extensions. The selection of which debugging extension should be used is made by the CUTEController component based on the available extensions in the environment. Currently there are three debugging extensions supported, for which the Debugging component contains the corresponding configuration possibilities. These configurations are based on the analysis of the TestMate testing extension.

Spawner - Start Executables

The Spawner component includes the functionality to start executables. The logic in this component allows it to pass specific arguments and execution options to an executable file. Further, there is functionality to read the standard output and standard error of an executable. This component is mainly used to start CUTE test executables in this application's context.

CodeAnalyzer - Analyze Code Features

The CodeAnalyzer component contains functionality to analyze code features using information provided by language server providers. Currently, there are two language server providers supported. The first and preferred one is the clangd extension [19] which provides language information performant and reliable. The second supported one is the C/C++ extension for VS Code by Microsoft [20]. This second option provides similar functionalities but not as performant as the first option. The main use cases of this component are the code-based test discovery and the functionalities provided by the CodeActionProvider component.

Workspace - Access Workspace Properties

The Workspace component contains logic to provide access to certain VS Code workspace and file information. This component builds the connection between the business logic and the workspace and file-based VS Code editor environment. This component further contains functionality to check the available third-party extensions and allows access to the VS Code workspace settings, which are used to configure certain components within this CUTE extension.

CMake - Analyze CMake Files

The CMake component offers parsing possibilities for CMake [9] files. This functionality provides access to certain properties of a CMake project configuration. The CMake component's functionality is used to find potential test executables based on the CMake configuration. This component's functionality primarily is accessed over the Workspace component within the other components.

5.3 Class Level Overview

In this section the architecture of each earlier described component will be analyzed on class level. Each components' classes are visualized in a class diagram followed by a brief description of the main concepts and implemented patterns used to create the components design. The diagrams are separated by the earlier defined packages Testing, Tools, Utilities and Environment. The classes of the components within each package will then be visualized in a way that allows an easy understanding of the used concepts.

5.3.1 Testing Component

This section contains the design of the Testing package classes. The class diagram below shows the individual classes split up into five independent namespaces. The diagram is followed by a description of the classes and their relations within the component but also across the whole project. All the classes, that contain some functionality, are implementations of some interface in order to achieve maximal extendibility and testability using unit tests.

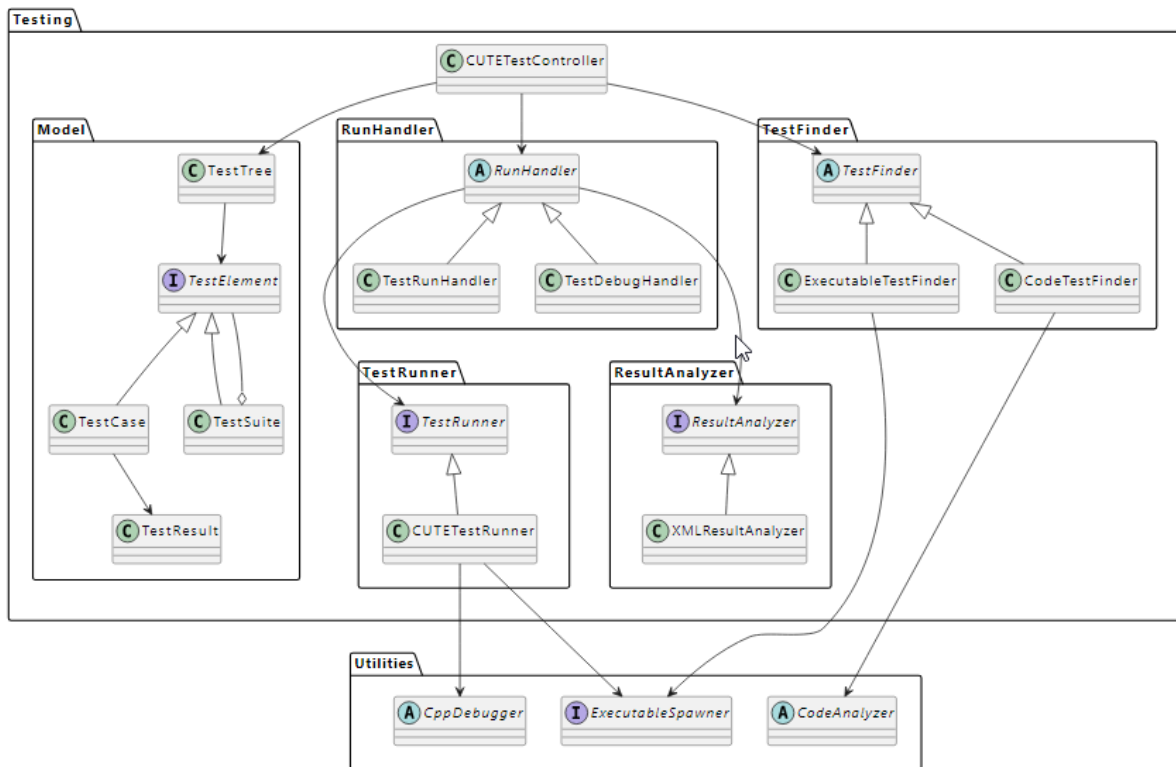


Figure 19: Testing component diagram

The **Model component** contains the domain classes which implement the domain model that can be found in chapter Domain Analysis.

The **TestFinder component** contains an abstract base class and the two implementations for code and executable based test discovery. The executable based implementation uses an implementation of the ExecutableSpawner interface, to start the test executables. The Code-TestFinder on the other hand uses an implementation of the abstract CodeAnalyzer class to get the required code information.

The **RunHandler component** contains an abstract RunHandler class and two corresponding implementations. One implementation, the TestRunHandler class, is responsible to start normal test runs, during which the tests get executed in parallel. The other implementation, the TestDebugHandler class contains the functionality to handle test runs with a debugger attached. These debug runs are executed sequentially test by test. The ResultHandler class uses an implementation of the TestRunner interface to run a test and an implementation of the ResultAnalyzer interface to analyze the test outcome.

The **TestRunner component** contains the TestRunner interface and its CUTETestRunner implementation. The CUTETestRunner uses an implementation of the ExecutableSpawner interface to start a test executable with the corresponding arguments.

The **ResultAnalyzer component** contains the ResultAnalyzer interface and its XMLResultAnalyzer implementation, which is used to analyze CUTE test results in the XML format.

5.3.2 Tools Component

This section covers the design of the Tools package classes. The diagram shown below, provides a basic overview of classes and namespaces within the components. As a diagram containing all classes of the tool package would not fit onto a single page, the decision was made to split the class diagram into the single components. The class diagrams of each component follow the below shown overview diagram.

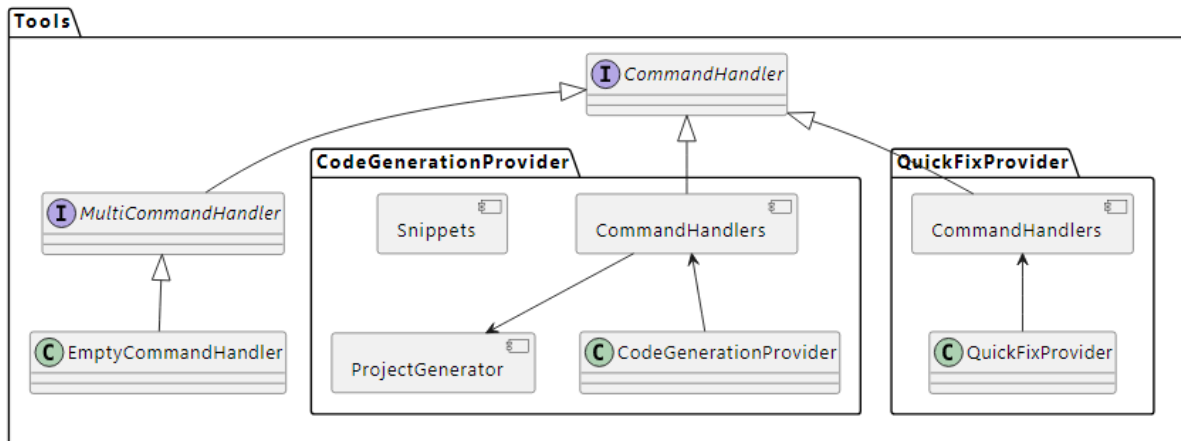


Figure 20: Tools component diagram

The tools package contains a `CommandHandler` interface which is extended by the `MultiCommandHandler` interface. This interface is implemented by the `EmptyCommandHandler` class, which represents a kind of null object [65]. The `EmptyCommandHandler` implementation is set as default command handler when the provider classes are instantiated without passing any specific command handler implementations. The `CodeGenerationProvider` component contains command handler implementations for the generate project and the generate new suite commands. The generate project command handler uses the logic from the `ProjectGenerator` namespace to set up a new project. In addition to the command handler implementations, the code snippet definitions can be found in this component in the form of JSON documents. The `QuickFixProvider` component is built in a similar way as the `CodeGenerationProvider` component. It contains command handler implementation for the quick fix commands which will be described later in this section. This overview is missing the `CodeActionProvider` component for readability reasons. The `CodeActionProvider` class diagram can be found below.

CodeActionProvider

The class diagram below reveals the classes and relations within the CodeActionProvider component. On the one hand, this component contains the TestCodeDiagnostician class, which is responsible to analyze the code for potential problems using information from an implementation of the abstract CodeAnalyzer class. The TestCodeDiagnostician class sets up a vscode.Diagnostics collection [64] that can be registered in the extension context to make the contained diagnostics available in the editor an problems view of Visual Studio Code. On the other hand, this component contains the TestCodeActionProvider class, which contains the logic to offer certain quick fixes for a problem detected by the TestCodeDiagnostician class. The TestCodeActionProvider class uses information from a CodeAnalyzer implementation to provide the potential code actions. The CodeActionProvider class implements the vscode.CodeActionProvider [87] class and can therefore be registered in the extension context to offer its functionality.

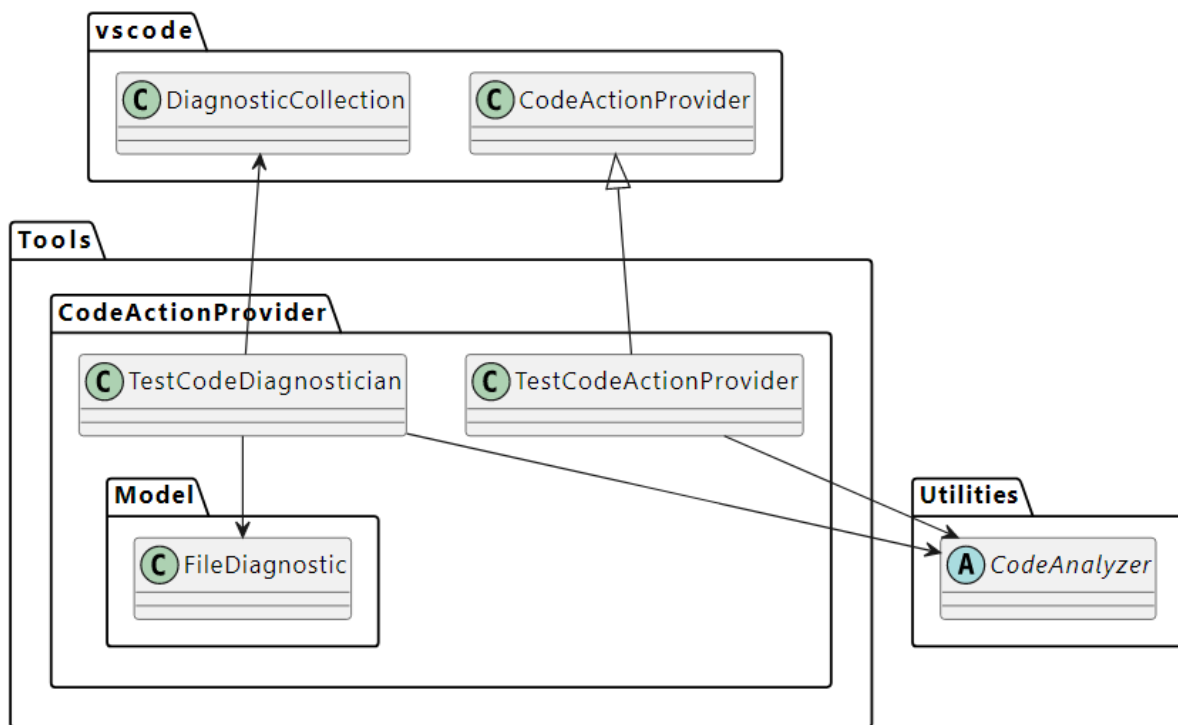


Figure 21: CodeActionProvider diagram

CodeGenerationProvider

The CodeGenerationProvider component contains two implementations of the CommandHandler interface. The first implementation is the GenerateNewSuiteFileCommandHandler class, which offers functionality to create a new test suite in a new file. The second implementation is the GenerateProjectCommandHandler class, which provides the functionality to create new test projects. The logic to set up a new test project is located in the CUTEProjectGenerator class, that implements the ProjectGenerator interface. The CodeGenerationProvider class offers the possibility to register the command handlers to the extension context and make them available thereby.

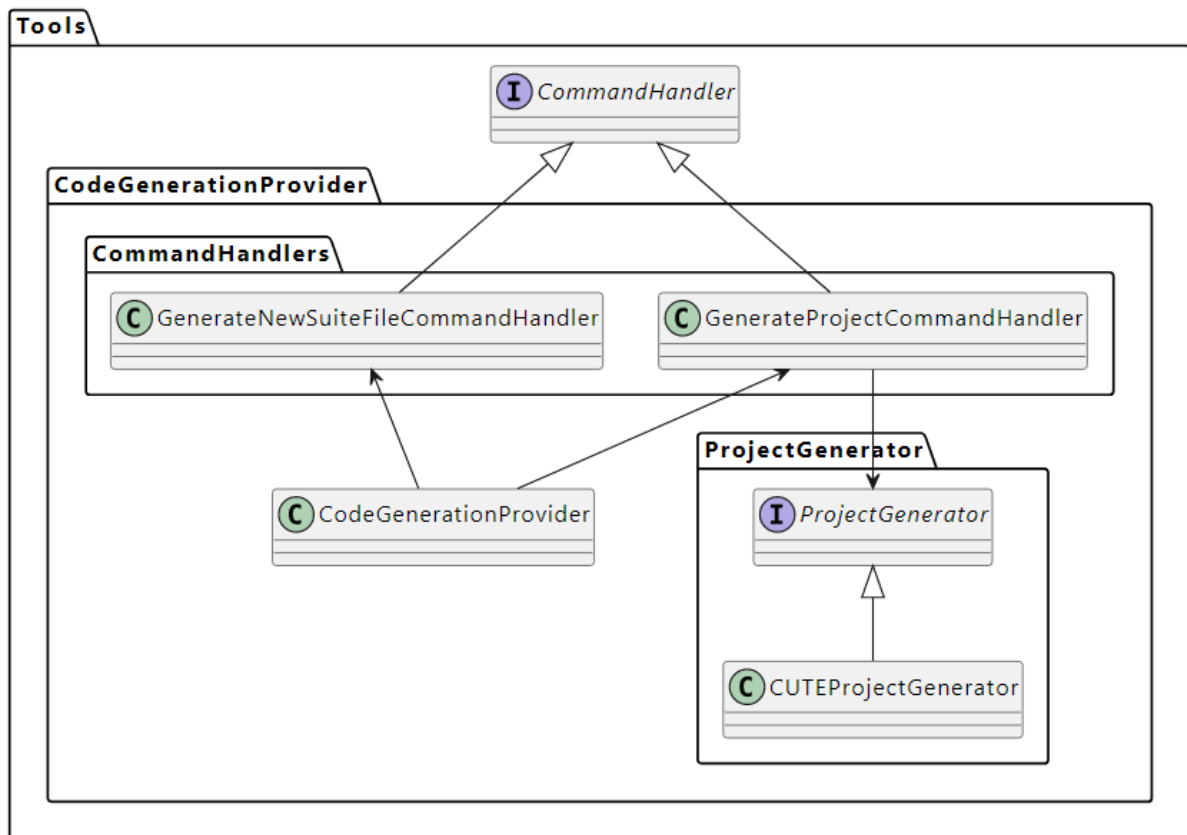


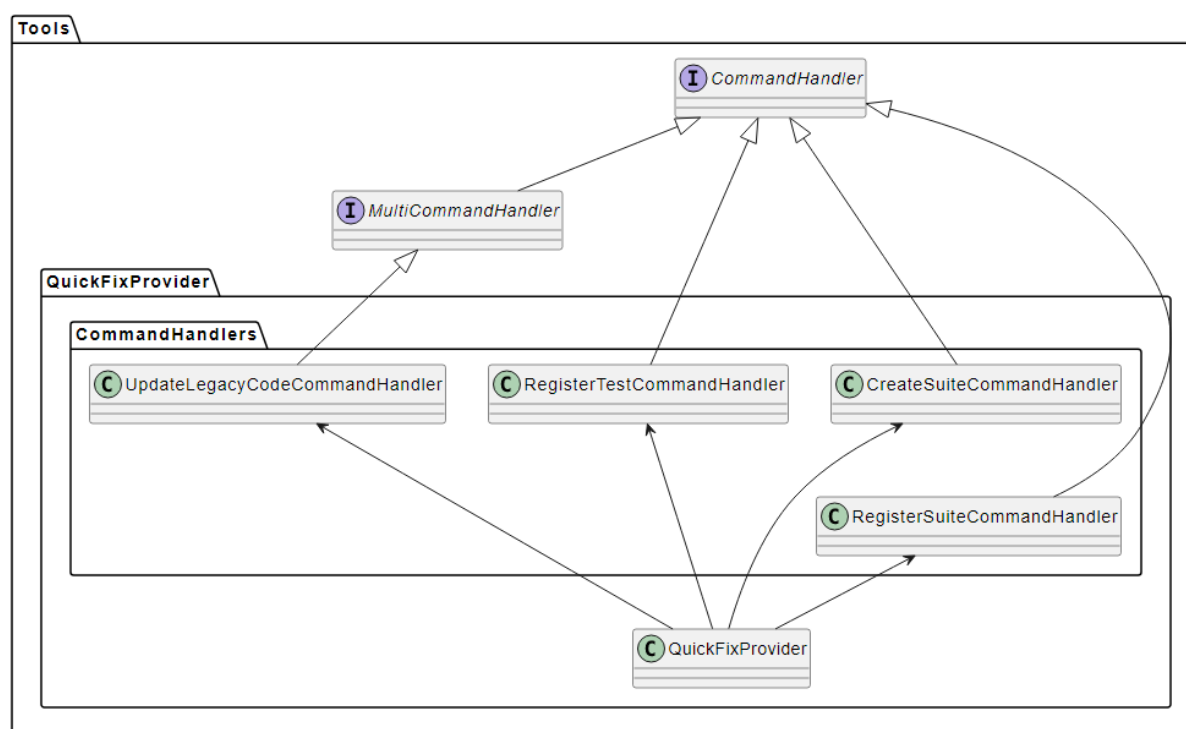
Figure 22: CodeGenerationProvider diagram

The QuickFixProvider component contains three implementations of the CommandHandler interface and one implementation of the MultiCommandHandler interface.

The *RegisterSuiteCommandHandler* contains the implementation of the logic that provides the functionality to register a test suite in the main method. This quick fix is provided by the *CodeActionProvider* if the *TestCodeDiagnostician* detected a test suite that potentially never gets called.

The *UpdateLegacyCodeCommandHandler* offers the functionality to update legacy syntax test declarations to the new syntax using the newly created TEST macro. This quick fix is provided by the CodeActionProvider if a test declaration using the old syntax should be discovered by the TestCodeDiagnostician. As this command handler implements the MultiCommandHandler interface, it is possible to update all legacy syntax test declarations at once.

The QuickFixProvider class is implemented in a similar fashion as the CodeGenerationProvider and offers the functionality to register the command handlers to the extension context to make them available.



5.3.3 Utilities Component

This section covers the design of the Utilities components classes. For reasons of readability the classes were split into two parts for the class diagram. The class diagram below, provides an overview of classes within the CodeAnalyzer component. The classes of the Spawner and the Debugging components follow in a separate class diagram below.

CodeAnalyzer

The CodeAnalyzer component contains an abstract CodeAnalyzer class, which contains the template methods for the code analysis. For each of the supported Language Server Provider extensions an implementation of that abstract CodeAnalyzer class can be found. Within these implementations the specific functionalities, which are used in the template methods, are implemented. There is an additional abstraction layer for the language server access. The code analyzer works against the LanguageFeatureClient interface for which there are two implementations. The first implementation is the CachedLanguageFeatureClient, which uses the VS Code built in commands to access the language servers and keeps track of the received information for performance improvements. The second implementation is the CppToolsLanguageFeatureClient, which contains some additional safeguards in the form of timeouts and Intellisense refresh commands to achieve the best possible results. The used implementation is injected into the CodeAnalyzer implementation by the EntryPoint Component.

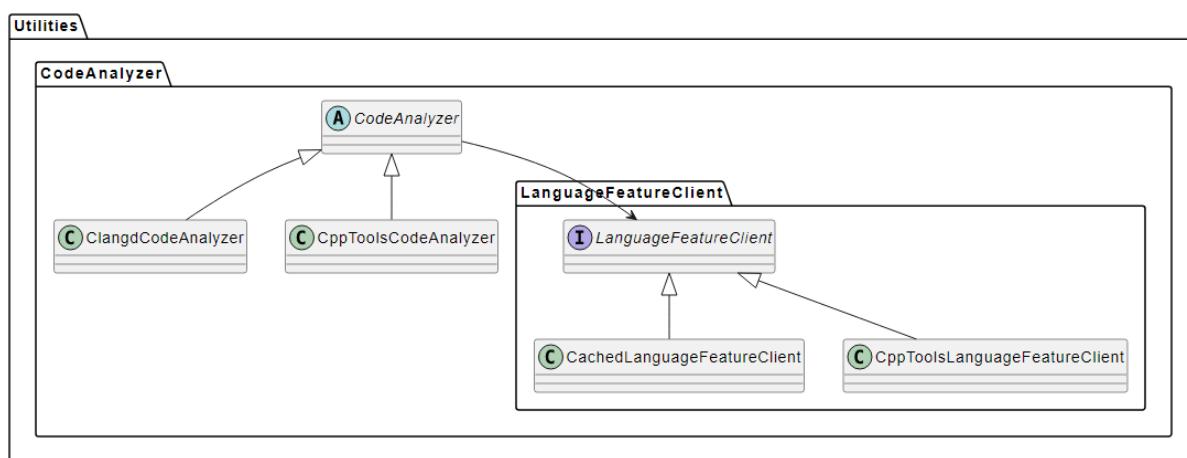


Figure 24: CodeAnalyzer diagram

Spawner & Debugging

The spawner component consists of the ExecutableSpawner interface with a single implementation in the form of the ChildProcessSpawner class. This class contains the functionality to start executables with specific arguments and execution parameters. Further it offers the possibility to read the standard output and standard error after the execution has finished.

The Debugging component consists of an abstract base class named CppDebugger. There are three implementations of this abstract class, for each supported debugging extension one. These implementations provide the possibility to configure the specific debugger in the context of CUTE test debugging. The decision about the implementation used is made by the DebugHandler based on workspace settings and installed extensions.

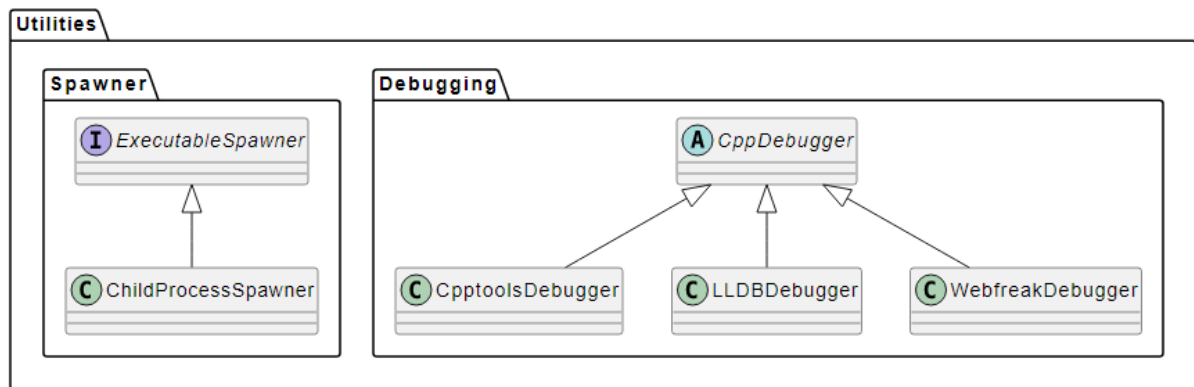


Figure 25: Debugger diagram

5.3.4 Environment Component

In this section the class design of the Environment components' classes is covered. This Environment package consists of two components, the Workspace and the CMake component.

The *IDEEnvironment* class provides functionality that allows to evaluate environment settings like installed third party extensions. This class plays a major role in the EntryPoint component, which decides which implementations to use based on information provided by the IDEEnvironment class.

The *constants* class contains constant variables such as command names. The benefit of using this constants class is that the rest of the code does not contain magic numbers or strings, what improves the maintainability of the extension.

The *Workspace* component contains three classes, that build the connection between the business logic and the workspace opened in Visual Studio Code. The *Workspace* class provides functionality to find and manage files and configurations within the workspace. Further, the *Workspace* class provides access to some CMake properties which are provided by the CMake parser. The *Document handler* provides a similar functionality on file level. It abstracts the logic of reading, writing, and updating files. The *WorkspaceWatcher* class provides the functionality to trigger events on file changes within the workspace. This logic is used to refresh the *TestTree* on which the test explorer is based.

The CMake component contains the *CMakeParser* class. This class provides the functionality to extract the CMake commands from a CMake configuration file. These commands can then be passed to the *CMakeList* class, which analyzes the commands further and splits them into the corresponding model classes. The *CMakeList* then provides the functionality to extract these discovered properties such as executables, libraries or include directories. This CMake parser functionality is used to detect potential test executables of a test project.

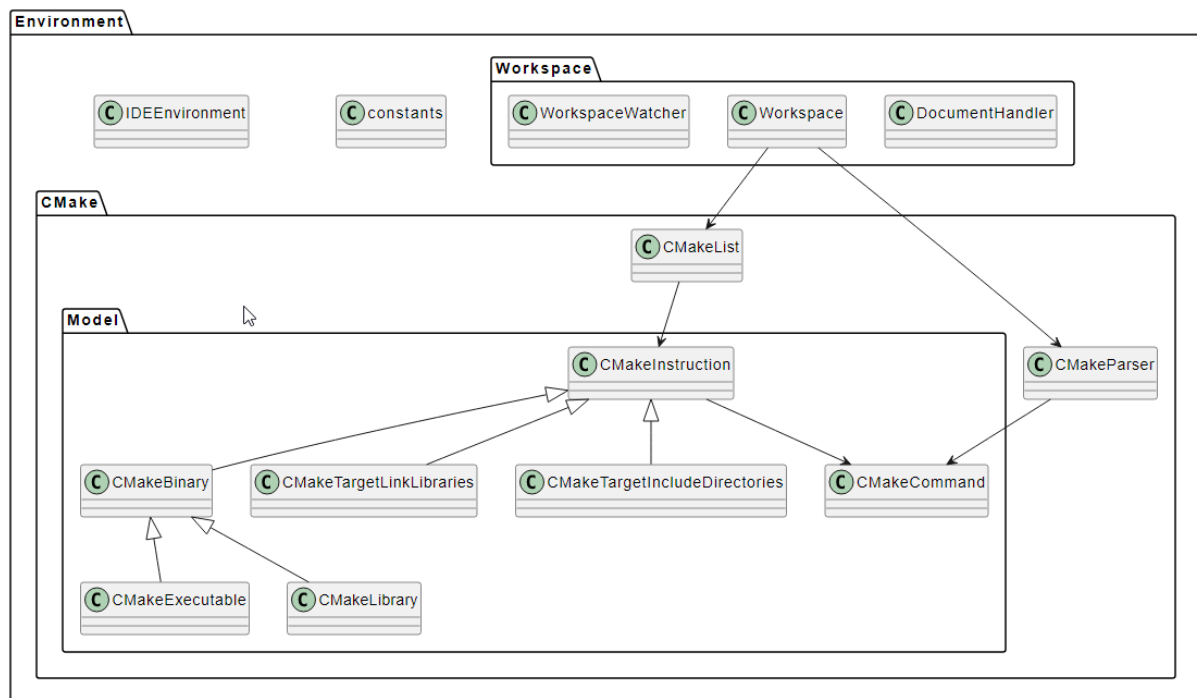


Figure 26: Environment component diagram

5.4 CUTE

In this section of the chapter Architecture overview a description of the changes made to the CUTE library y [10] itself can be found. These changes were necessary to provide a performant and reliable way of detecting test cases, test suites and their implementation locations. This section does not just include the final solution but rather all the different attempts made with a corresponding description for each attempt. The main goal was it to keep the library in backward compatible shape such that already existing test projects still work with the updated CUTE library without having to change any code within the test projects.

- (i) Extension of the already existing CUTE marco.

The problem with this attempt was, that with the extension of the already existing CUTE macro it would only be possible to get the location at which the test function is wrapped into to a cute test and not the location of the test function itself. Therefore, no solution to extend the CUTE macro in a way, that that would allow it to receive the required location of a test function, could be found using this first attempt.

- (ii) New macro used for test definition. (V1)

For this second approach global objects, which include the required information, should be created. This required information includes the test name, the suite name and the filename and line number of the location where the test function can be found. The main difficulty of this approach was, that it was not possible to create a connection between the test, which was being executed, and the global location object during a test run. Using this approach all the required information could be gathered but the test cases could not be assigned to its location and vice versa.

In the listing below the macro, which was created during this second approach, can be seen.

```
1 #define TEST(name) void name(); \
2 namespace { \
3     cute::location name ##__LINE__((&name), __FILE__, __LINE__); \
4 } \
5 void name()
```

Listing 11: TEST macro v1

(iii) New macro used for test definition. (V2)

In this third approach we tried to move the creation of a cute test to the actual test function and therefore put a new function in front of the real test function. The storing away of the location now does not take place during compile time macro replacement any longer but rather at runtime. The reason for this behavior is, that we now return the cute test from a wrapper function which is created by the new macro. The newly created wrapper function can be called, what then puts its location in a global vector object. In the end the actual cute test is the return value of this new wrapper function. The main problem with this approach is, that there is no compare operation available for `std::function`. This compare operation would be needed to connect a location to the native test function. Using already existing test projects that still use the old CUTE macro should also work using this updated CUTE library. The only downside of using the old CUTE macro is that the CUTE extension for Visual Studio Code could not offer its full functionality.

```
1 #define TEST(name) \
2 void name ## _IMPL(); \
3 cute::test const name() { \
4     globalMap.insert({name ## _IMPL, cute::location{...}}); \
5     return cute::test{...}; \
6 } \
7 void name ## _IMPL()
```

Listing 12: TEST macro v2

(iv) New macro used for test definition. (V3)

During this third approach the given functionality from the second approach was extended and modified slightly. The logic of a wrapper function introduced by the TEST macro was carried over from the second approach. This wrapper function still returns a cute test as it already was the case after the second approach. But this third approach brought a new location object along which is stored in the cute test object. With this approach there was no way around editing the cute test class as a new private member containing the location needed to be added. Even though the backward compatibility could still be kept as the location constructor parameter is optional and if not provided, as this is the case when using the old CUTE macro, the location property will simply stay empty. In the CUTE extension for Visual Studio Code this special case of having an empty location property in the cute test object will be handled by warning the user about not being able to enjoy full experience provided by the test extension. (Code navigation for example will not be available)

```
1 #define TEST(name) namespace { \
2     void name ## _IMPL(); \
3 } \
4 cute::test const name() { \
5     return cute::test( \
6         (&name), \
7         (cute::location(__FILE__, __LINE__)), \
8         (#name) \
9     ); \
10 } \
11 void ::name ## _IMPL()
```

Listing 13: TEST macro v3

6 Implementation

This chapter should provide an overview of the CUTE extension's implementation. To provide this overview, a description of the used technologies is available at the beginning of this chapter. This general description of the technologies used to implement the CUTE extension for VS Code is followed by a description of the implemented quality measures and an explanation of the testing infrastructure. Subsequent to these general topics, the logic behind the key functionalities is described and explained. To clarify the more complex workflows within the extension, sequence diagrams are available in this chapter.

6.1 Used technologies

In this section the used technologies will be covered. Thereby first of all a general overview will be provided, whereafter the technologies used in the different components will be explained in detail. There are some basic constraints for the development of Visual Studio Code extensions such as the programming language which is Typescript. Each of the used VS Code interfaces that is used within the CUTE extension will be described later in this section. Further, some used tools such as linter and prettier will be explained.

6.1.1 Extension Folder Structure

The listing below shows the standard folder structure [66] on which all Visual Studio Code extensions are based. The CUTE extension is no exception and therefore builds on the same file structure as the one shown below. The only difference is that the CUTE extension only uses the extension.ts file as entry point and to set everything up. The business logic is distributed over different files according to the chosen extension design.

```
1  .
2  +-- .vscode
3  |   +-- launch.json    // Launching and debugging config
4  |   \-- tasks.json     // Build task config to compile TypeScript
5  +-- .gitignore         // Ignore build output and node_modules
6  +-- README.md          // Extension's functionality description
7  +-- src
8  |   \-- extension.ts   // Extension source code
9  +-- package.json       // Extension manifest
10 \-- tsconfig.json      // TypeScript configuration
```

Listing 14: Extension Standard Folder Structure [66]

6.1.2 Extension Manifest

The package.json file builds the extension manifest and contains a mix of Node.js fields and extension specific properties such as activationEvents and contributes. Every VS Code extension must have such an extension manifest in the form of a package.json file. In this section the manifest of the CUTE extension is analyzed and explained.

General Extension Properties

The listing below shows the general extension focused properties defined in the package.json file. The combination of the publisher and name properties in the form publisher.name builds the unique identifier of the extension. The CUTE extension is started as soon as the startup has finished, as its only activation event is onStartupFinished [69]. The entry point of the extension is the out/extension.js file, which is built from the extension.ts file which will be explained later in this chapter. The contributes object is covered separately to improve the readability.

```
1 {  
2   "name": "cute-testing",  
3   "publisher": "CUTETest",  
4   "displayName": "CUTE Testing",  
5   "description": "VS Code support for the CUTE testing framework",  
6   "version": "0.0.1",  
7   "homepage": "https://cute-test.com/",  
8   "repository": "https://gitlab.ost.ch/cute-extension-for-vs-code/",  
9   "engines": {  
10     "vscode": "^1.67.0"  
11  },  
12   "categories": [  
13     "Other"  
14  ],  
15   "activationEvents": [  
16     "onStartupFinished"  
17  ],  
18   "main": "./out/extension.js",  
19   "contributes": {  
20  
21  },
```

Listing 15: General Extension Properties

Contributed Extension Commands

The next listing shows the configured commands [68]. This commands object is a part of the contributes property, that can be seen in the listing above. The commands object contains the configuration of the commands that the CUTE extension provides. The three commands provided can be used to generate new test projects, to add new test suite files and to update the legacy syntax in the whole workspace.

```
1  "commands": [  
2    {  
3      "command": "cute.generateProject",  
4      "title": "CUTE: Generate project"  
5    },  
6    {  
7      "command": "cute.createSuiteFile",  
8      "title": "CUTE: Create test suite file"  
9    },  
10   {  
11     "command": "cute.updateAllLegacySyntax",  
12     "title": "CUTE: Update legacy syntax"  
13   }  
14 ]
```

Listing 16: Extension Commands

Contributed Extension Snippets The listing below shows the configured snippets [67]. This snippets object is a part of the contributes property, as the already mentioned commands object. This configurable property contains the snippet definition location and the language identifier for which the snippet is usable. The CUTE extension offers a snippet to create TEST(name) tests within a cpp test file.

```
1  "snippets": [  
2    {  
3      "language": "cpp",  
4      "path":  
5        ↪ "./src/tools/CodeGenerationProvider/Snippets/snippets.json"  
6    }  
7  ]
```

Listing 17: Extension Snippets

6.1.3 Extension Entry File

This section contains an explanation of the extension entry file which for the CUTE extension is the `extension.ts` file. This entry file exports two functions, the `activate` and the `deactivate` functions. The `activate` function is called when an activation event occurred, which is defined in the `package.json` file. In the case of the CUTE extension, the `activate` function is called as soon as the Visual Studio Code startup has finished. Inside the `activate` function, a `CUTEExtension` instance is created, which is responsible to set up the CUTE testing functionality and register it to the `vscode.ExtensionContext`. The `deactivate` function is called when the extension is disabled or uninstalled and can be used to clean up the environment. The CUTE extension simply shows a notification when the extension gets deactivated.

```
1 export async function activate(context: vscode.ExtensionContext) {
2     vscode.window.showInformationMessage('CUTE testing extension is active now
   :');
3
4     try {
5         const cuteExtension = new CUTEExtension();
6         cuteExtension.setup(context);
7     } catch (error) {
8         let errorMessage = 'Something went wrong while activating the CUTE testing
   extension';
9         if (error instanceof Error) {
10             errorMessage += `: ${error.message}`;
11         }
12         vscode.window.showErrorMessage(errorMessage);
13     }
14 }
15
16 export function deactivate(): void {
17     vscode.window.showInformationMessage('CUTE testing extension got deactivated
   ');
18 }
```

Listing 18: Extension Entry Point

6.2 Quality measures

In this section the implemented quality measures are described. Thereby the different tools and measures will be described individually. All the here described measures should help to keep the code quality high and allow it to build a well maintainable extension for VS Code.

6.2.1 Automated Tests

There are automated unit-, integration- and system tests available, that cover the most important features of the CUTE extension. The automated tests are based on the mocha testing framework. VS Code offers the test-electron API [82] to simplify extension testing. This API offers functionalities such as launching VS Code with a specific workspace, downloading a different version of VS Code rather than the latest stable release or launching VS Code with additional CLI parameters. For the integration and system tests there is a sample CUTE testing project available to simulate a workspace containing a valid testing project. When running the automated tests, a VS Code extension development host [83] is started to host the CUTE extension.

6.2.2 Manual Tests

To cover the whole functionality offered by the CUTE extension for VS Code, a set of manual tests was defined. The instructions for these manual tests can be found in the chapter Testing. Using these manual tests, the usage of the CUTE extension should be simulated to test the usability and functionality of all features in a way that they will be used in production. These manual tests provide the possibility to test the flow through the entire functionality also including the UI.

6.2.3 Continuous Integration (CI)

To ensure a high quality a continuous integration (CI) pipeline was set up. This pipeline should prevent breaking changes being merged into the master branch. Failing builds need to be addressed immediately to keep the general quality of the extension high. The continuous integration pipeline contains the following steps. These steps ensure the code style and the code functionality. There is also a step to automatically publish the extension. (Continuous Deployment CD)

1. Linting
2. Building
3. Testing
4. Publishing

The linting should ensure that the style guide that was chosen gets implemented. The decision was made to use the AirBnB style guide [70] with some adjustments for prettier. Prettier [81] is an extension to format the project properly. In a final step some metrics are calculated with SonarQube [14] to get a summary overview about code smells such as bad programming style or security issues. In the end the package will be published automatically in the VS Code marketplace [8] when a tag is created.

6.3 General Extension Logic

In this section the main setup of the CUTE extension is explained. To provide an overview, the different configuration options are described for each supported language server provider. For each combination of workspace settings and installed extensions the available features provided by the CUTE extension are listed below.

6.3.1 TEST(...) macro available

The CUTE extension checks whether or not the newly defined TEST(...) macro is available in the CUTE framework [10] files in the currently opened workspace. If the new macro is available, the preferred way of test discovery is the executable based implementation. There is a configuration option available to override this preference and to choose the code-based test discovery implementation. This setting can either be edited in the Settings UI [84] under Extensions->CUTE testing or the settings.json file. If the code-based implementation should be chosen without any language server provider extension installed, the user gets notified that this configuration is not valid in the current setup.

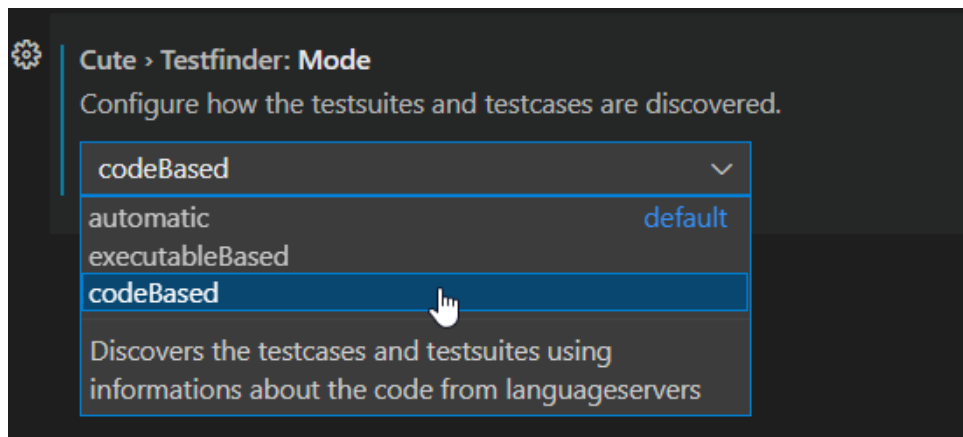


Figure 27: TestFinderMode Setting

6.3.2 TEST(...) macro not available

The CUTE extension checks whether or not the newly defined TEST(...) macro is available in the CUTE framework [10] files in the currently opened workspace. If an old version of the CUTE framework should be used in the opened workspace, the executable based implementation of the test discovery logic is not available. If the executable based implementation should be chosen in the settings anyhow, an error message is shown to the user that tells him that this configuration is not valid in the current project.

6.3.3 Clangd Installed

If the Clangd [19] extension is installed, all features are available and work with a decent performance. If an old version of the CUTE framework [10] should be used, that does not support the new TEST(...) macro, it is advised to use the code-based test discovery implementation in combination with the Clangd extension. No additional settings are required to use the full functionality including the convenience tools such as quick fixes and code diagnostics.

6.3.4 C/C++ for VS Code Installed

If the C/C++ for VS Code extension (CppTools) [20] should be installed, all features are available. If an old version of the CUTE framework should be used, that does not support the new `TEST(...)` macro, the code-based test discovery implementation can be used in combination with the CppTools extension. Using this configuration, a worse performance is expected, as the CppTools cross reference provider does not deliver results as fast as the Clangd provider. On low performance systems it occurs from time to time, that the CppTools cross reference provider gets stuck, what then requires a restart of Visual Studio Code. If this behavior can be observed frequently there is a setting available to use the explicit language client mode. This setting does only affect the CUTE extension in combination with the CppTools extension. If this configuration option is set explicit the CUTE extension opens the files from which a reference request is triggered in the explorer, before querying the language server. Using the explicit mode, the cross-reference provider gets stuck no longer. As the automatic opening of files leads to a compromised user experience, the code diagnostics are not available in this configuration. With the code diagnostics disabled, the requests to the language server get reduced to a minimum.

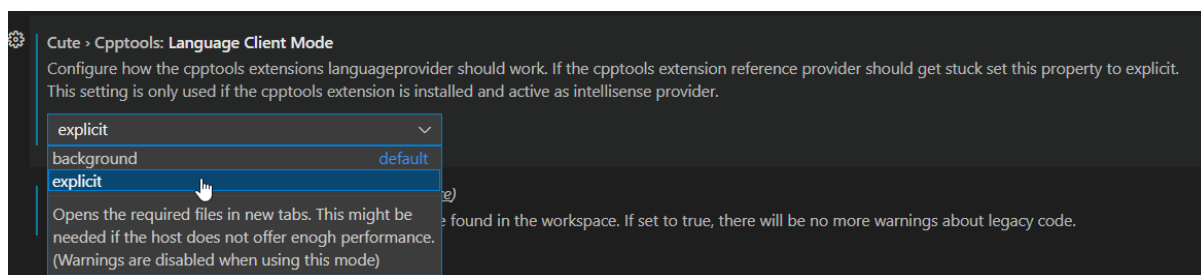


Figure 28: LanguageClientMode Setting

6.3.5 Clangd & C/C++ for VS Code Installed

If both supported language server provider extensions should be installed, the Clangd functionality is preferred for performance reasons. The Clangd [19] extension itself shows a warning if installed together with the CppTools [20] extension and requests the user to disable the CppTools IntelliSense functionality.

6.3.6 No Language Server Provider (LSP) Installed

If none of the supported language server provider extensions should be installed, the convenience tools and code-based test discovery are not available. The CUTE extension informs the users about the reduced functionality. If only an old version of the CUTE framework that does not support the new `TEST(...)` macro is available, no test discovery implementation can be used. In this case the CUTE testing extension is not usable to run or debug tests.

6.4 RunHandler Implementation

In this section an overview of the logic behind the RunHandler can be found. There are two implementations of the abstract base RunHandler class. One implementation's responsibility is to handler normal test runs and the other's is to handle test runs with a debugger attached. Both implementations are explained in the following sections using sequence diagrams and a description of all involved steps. Both RunHanlder implementations are registered as run profiles in VS Code over the Testing API. When a test run is started over the TestExplorer UI or a command, these RunHandler implementations are called based on the selected run profile.

6.4.1 TestRunHandler

When a normal test run is triggered over the Testing API, this RunHandler implementation is called. The following sequence diagram shows all involved elements and their interactions during a test run. This RunHandler implementation allows it to run tests in parallel.

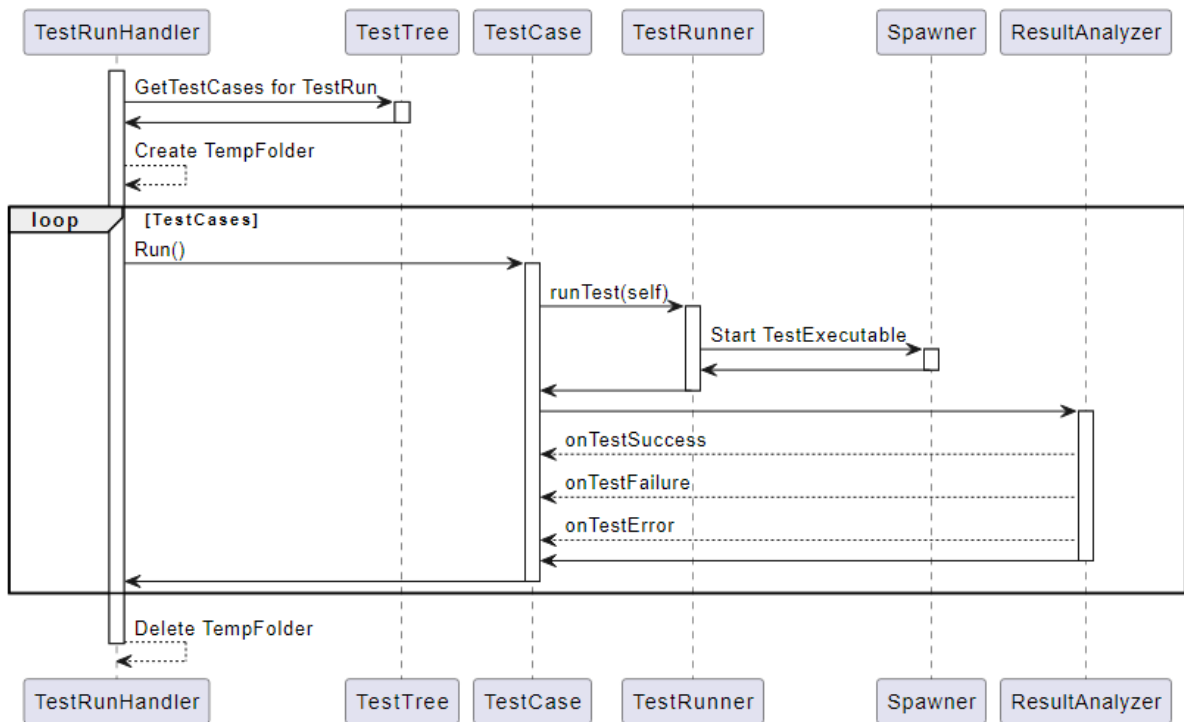


Figure 29: TestRunHandler sequence diagram

1. **Filter the TestTree** - find test cases included in the test run The first step in a test run sequence is to filter the discovered test cases based on the user's selection in the test explorer. The TestTree implementation allows it to filter test cases over multiple nested levels of test suites.
2. **Prepare temp working directory** - create a temporary directory to write the results to After the tests, that should be included in a test run, are found, a temp folder gets created. In this temp folder a subfolder for each test will be created during the test run. This temporary folder is set as current working directory when a test execution is started. The benefit of this temporary folder structure is the possibility to run tests in parallel without overwriting the result XML files of other tests in the same test run.
3. **Run Test** - execute tests in parallel and analyze their results Once the environment is set up and all tests, that should be part of the run, are found, the run method on each TestCase instance is called. Each TestCase instance will then start its registered TestRunner instance with the required parameters. The TestRunner instance itself will then use the Spawners functionality to start the test executable with the corresponding arguments for that specific test. The TestRunner instance reads the XML result file from the temporary folder as soon as the test executable process has finished. The result is then passed to the result analyzer, which is registered on the TestCase instance. The result analyzer analyzes the result and notifies the TestCase over the outcome using callbacks. Inside these callbacks the vscode.TestItem instance that is a part of the TestCase instance will be updated.

```
1 %temp%\{TestRun_GUID}
2 +---test1
3 |       Test.exe.xml
4 |
5 +---test2
6 |       Test.exe.xml
7 |
8 +---test3
9 |       Test.exe.xml
10 |
11 \---test4
12       Test.exe.xml
```

Listing 19: TestRun Folder Structure

6.4.2 TestDebugHandler

When a debug test run is triggered over the Testing API [37], this RunHandler implementation is called. The following sequence diagram shows all involved elements and their interactions during a debug test run. This RunHandler implementation does not support parallel test executions. This RunHandler is only registered as run profile if any of the supported debugging extensions is installed.

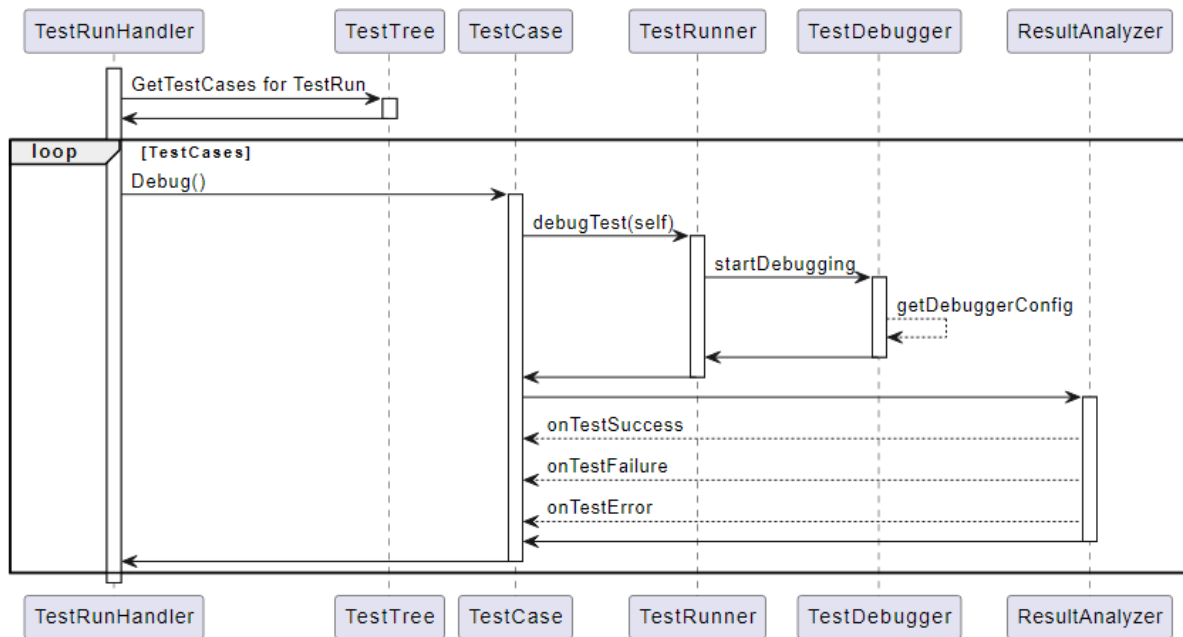


Figure 30: TestDebugHandler sequence diagram

1. **Filter the TestTree** - find test cases included in the test run The first step in a test debug sequence is similar to the one described under the normal test run sequence. It is about filtering the discovered test cases based on the user's selection in the test explorer. The TestTree implementation allows it to filter test cases over multiple nested levels of test suites.
2. **Start Test Debugging** - start a debugging process for each test The second step in the test debug process starts similar to the normal test run sequence. On each TestCase instance the debug method is called to start the debugging of that test. This time the call to the debug method is made test by test and not in parallel. Therefore, no temporary file structure is required as just one test is executed at a time.
3. **Configure Debugger** - set up the debugger configuration based on the debugger implementation The TestCase instance's debug method triggers the debugTest method on the TestRunner instance. This time the TestRunner instance does not use the Spawner directly to start the test executable. The TestRunner instance uses a TestDebugger instance, to set up the VS Code debugger. The class of the TestDebugger instance depends on the installed debugging extensions.
4. **Analyze Result** - check the tests outcome The result analysis step is equal to the one described in the normal test run sequence. The ResultAnalyzer analyzes the result, which is passed as string and uses the callback functions to update the vscode.TestItem [47] on the TestCase instances.

6.5 TestFinder Implementation

In this section the logic behind the TestFinder functionality is explained. This functionality is used to set up the TestTree instance, which builds the base of the testing user experience. In order to make tests available in the test explorer they first need to be discovered and then registered using the VS Code Testing API [37]. The sequence behind the test discovery and registration will be described in this section using diagrams and explanations of the single steps.

6.5.1 ExecutableTestFinder

This section contains a description of the sequence, that is used to discover tests using the executable based TestFinder implementation. This is the most reliable and performant implementation of the TestFinder functionality and therefore the preferred one.

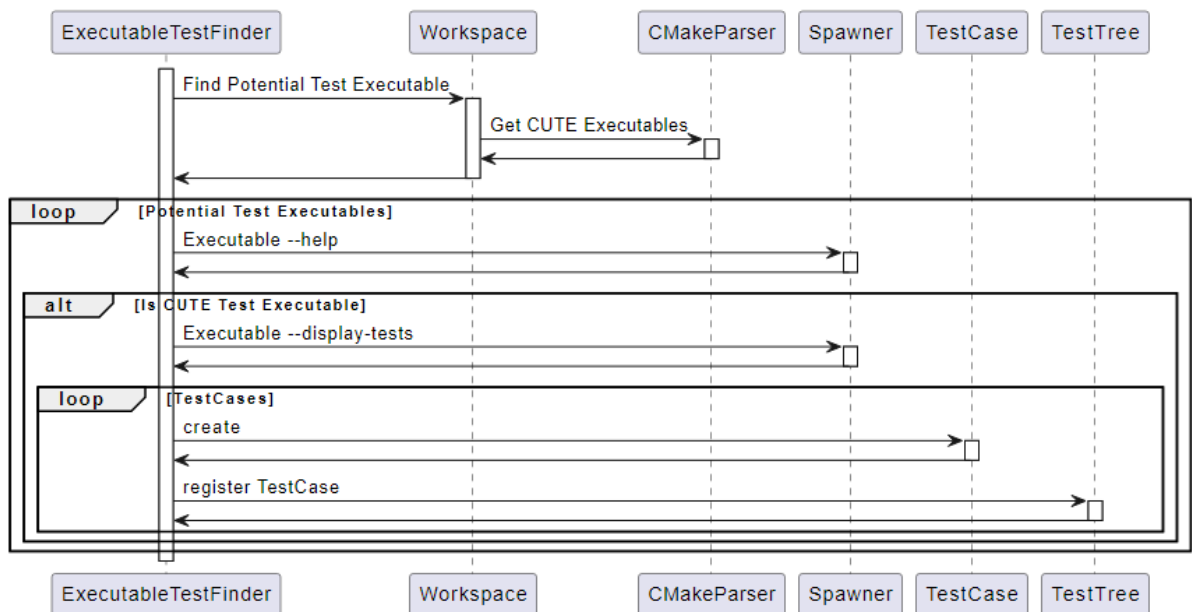


Figure 31: ExecutableTestFinder sequence diagram

1. **Find potential Test Executables** - find executables which include the CUTE framework The first step of the executable based test discovery is to find the potential test executables in the opened workspace. Therefore, the CMake file is parsed for executables that link the CUTE framework. All executables declared in the CMake file, which have a link to a CUTE framework file are returned by this step.
2. **Check Executables** - check if it is a CUTE executable To be completely sure that a discovered executable is a CUTE test executable, it is started using the spawner functionality with the `-help` argument. The standard output of the executable process is checked against the expected CUTE executable help text. Only executables that match the expected output are analyzed further. The `-help` flag was added to the CUTE framework during this project. Therefore, the executable based test discovery cannot be used with older versions of the CUTE framework [10].
3. **Query Tests** - find tests and test suites in executable For each test executable that was discovered using the previous two steps, the contained tests and their corresponding test suites are evaluated. The executables are started again but this time with the also newly added `-display-tests` argument. After the executable process has finished, the standard output contains the test elements of a specific test executable. If a test should belong to a test suite the format is `"TestSuite#TestCase \t TestFile \t LineNumber"`. If the test should be registered directly to a runner, the format is `"TestCase \t TestFile \t LineNumber"`.
4. **Create TestElements** - Create TestElement instances and register them In this next step, for each of the discovered test cases a TestCase instance is created together with a TestSuite instance if it did not exist before. When creating a new instance of a TestCase or TestSuite, a corresponding vscode.TestItem instance is created and added as property. These vscode.TestItem instances will later be registered to the test explorer using the VS Code testing API [37] when their containing TestElement instances are added to the TestTree. After this step, the tests and test suites can be started and included into test runs using the test explorer.

6.5.2 CodeTestFinder

This section contains a description of the sequence, that is used to discover tests using the code based TestFinder implementation. The performance and reliability of this implementation depends heavily on the language server provider extension used and the underlying system performance. Using the Clangd [19] extension, this TestFinder implementation is almost as performant and reliable as the executable based implementation.

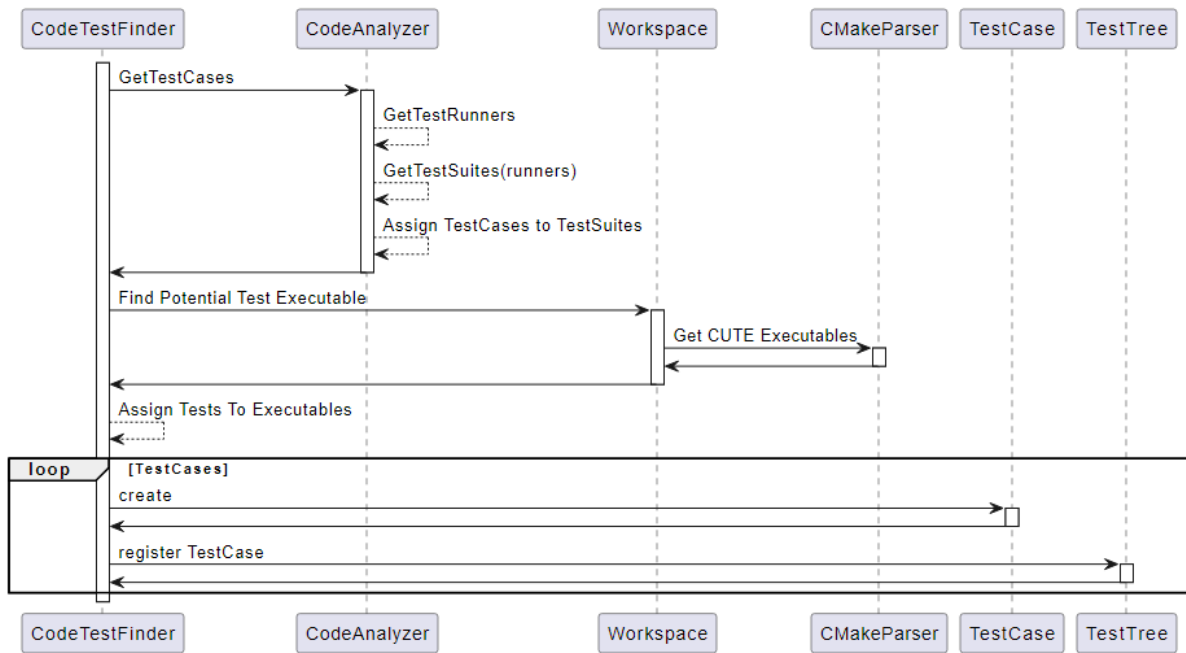


Figure 32: CodeTestFinder sequence diagram

1. **Get Tests** - find tests using the CodeAnalyzer The first step in this sequence is to find the test cases using the CodeAnalyzers functionality. The CodeAnalyzer offers functionalities to discover test runners, test suites and test cases based on information provided by the language server provider extensions. All information is combined in order to get a list of tests with their corresponding test suite and their implementation location. The exact logic behind these discovery steps is described in the next section under CodeAnalyzer implementation.
2. **Get Potential Executables** - analyze CMake configuration to get the executables The next step of the code-based test discovery is to find the potential test executables in the opened workspace. Therefore, the same logic is used as in the executable based TestFinder implementation. The CMake file is parsed for executables that link the CUTE framework. All executables declared in the CMake file, which have a link to a CUTE framework file are returned by this step.
3. **Assign Executables To Tests** - find the executable behind each test case After the test cases and the potential test executables were detected, the information needs to be connected. Based on includes in the CMake configuration of the potential executable, the executables are assigned to the tests. Primarily this assignment is based on the runner call location or the runSuite method location of a test case. If no executable can be found that includes any of these files, the test implementation location is taken into consideration as well. Only if an executable can be found for a test it will be registered to the TestTree and therefore made visible in the test explorer.
4. **Create TestElements** - Create TestElement instances and register them In this last step, for each of the discovered test cases a TestCase instance is created together with a TestSuite instance that it did not exist before. When creating a new instance of a TestCase or TestSuite, a corresponding vscode.TestItem [47] instance is created and added as property. These vscode.TestItem instances will later be registered to the test explorer using the VS Code testing API [37] when their containing TestElement instances are added to the TestTree instance of the CUTEController [46]. After this step, the tests and test suites can be included into test runs using the test explorer UI.

6.6 CodeAnalyzer Implementation

This section contains an overview of the most important CodeAnalyzer features. For each of these key features an explanation of the logic it is based on, can be found. The CodeAnalyzer functionality is used for the code-based test discovery and the convenience tools such as code diagnostics and quick fixes [63].

6.6.1 Find Test Runners

The CodeAnalyzer provides functionality to find CUTE test runner instances throughout the project. The logic behind this functionality is based on information received from language server providers and text-based analysis of certain code files or passages within the testing project. The figure and descriptions below explain that logic in a step-by-step manner.

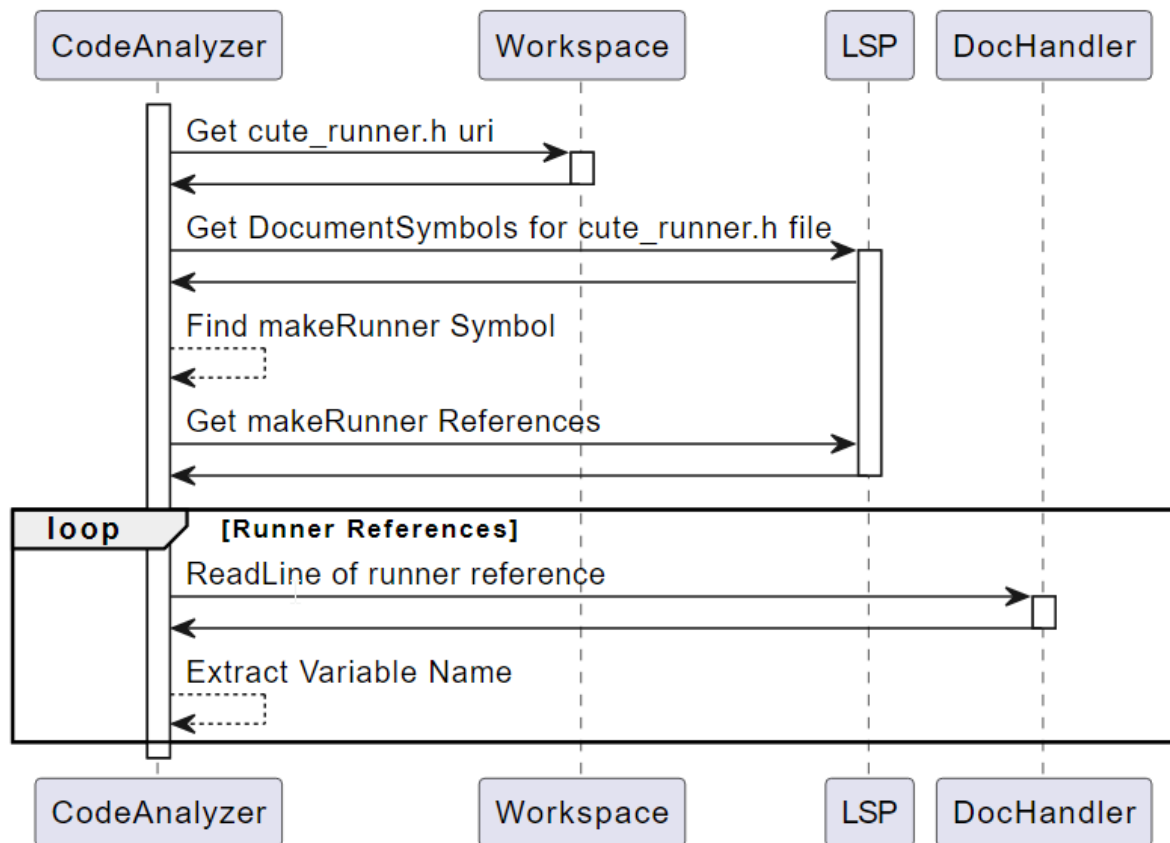


Figure 33: Find Test Runners sequence diagram

1. **Get TestRunner URI** - find the test runner file

The first step in the test runner discovery sequence is to find the `cute_runner.h` file within the workspace. The `cute_runner.h` file is a part of the CUTE testing framework and contains the implementation of the CUTE test runner and amongst others a `makeRunner` function to set up and instantiate a new test runner. The file is searched using the `vscode.workspace` functionality with the global pattern `'**/cute_runner.h'`.

2. **Get makeRunner Function** - find the location of the `makeRunner` function

The second step in the sequence to discover the CUTE test runners is to find the location of the `makeRunner` function within the before found `cute_runner.h` file. This logic is based on the document symbols of that file provided by the language server providers. As there is a difference between document symbols returned by the `clangd` [19] extension and the `CppTools` [20] extension, the specific logic is implemented in the corresponding specializations of the `CodeAnalyzer` base class.

- **CppTools**

If the `CppTools` extension is used as language server provider, a document symbol called `"makeRunner<Listener>(Listener &, int, const char * const *)"` is searched. The parent of this document must be called `"cute"`.

- **Clangd**

If the `clangd` extension is used as language server provider, a document symbol simply called `"makeRunner"` is sufficient if its parent symbol is called `"cute"`.

```
1  template <typename Listener>
2  runner<Listener> makeRunner(Listener &s, int argc = 0, const char *const *argv = 0){
3      return runner<Listener>(s, argc, argv);
4  }
```

Listing 20: `cute_runner.h` `makeRunner` implementation

3. **Get makeRunner References** - find all calls of the `makeRunner` function After the location of the `makeRunner` function is found, all references within the project need to be found, as each call of this function creates a new CUTE test runner. These references are discovered using the language server providers cross reference detection functionality. The language server providers return a list of locations at which the `makeRunner` function is called.

4. **Get the runners name** - extract the name of the test runner For each earlier detected reference to the `makeRunner` function, the corresponding code line is read and evaluated. The line is split at the `"="` sign and the right-hand side needs to match `"makeRunner"`. If this is the case the last nonempty string on the left-hand side is taken as the name of the runner. In the following example, name detected is `runner1`. This name is returned together with the location of the `makeRunner` call.

```
1  auto runner1 = cute::makeRunner(listener, argc, argv);
```

Listing 21: CUTE Runner Instantiation

6.6.2 Find Test Suites

The CodeAnalyzer provides functionality to find CUTE test suites for a set of CUTE runners, that needs to be provided as parameter. The output of the GetTestRunners functionality, which is described above, can be used as parameter input. The logic behind this functionality is based on information received from language server providers over the language server protocol (LSP) [18] and text-based analysis of certain code files or passages within the testing project. The diagram below provides an overview of the logic. Subsequent to the diagram, a description of each step can be found.

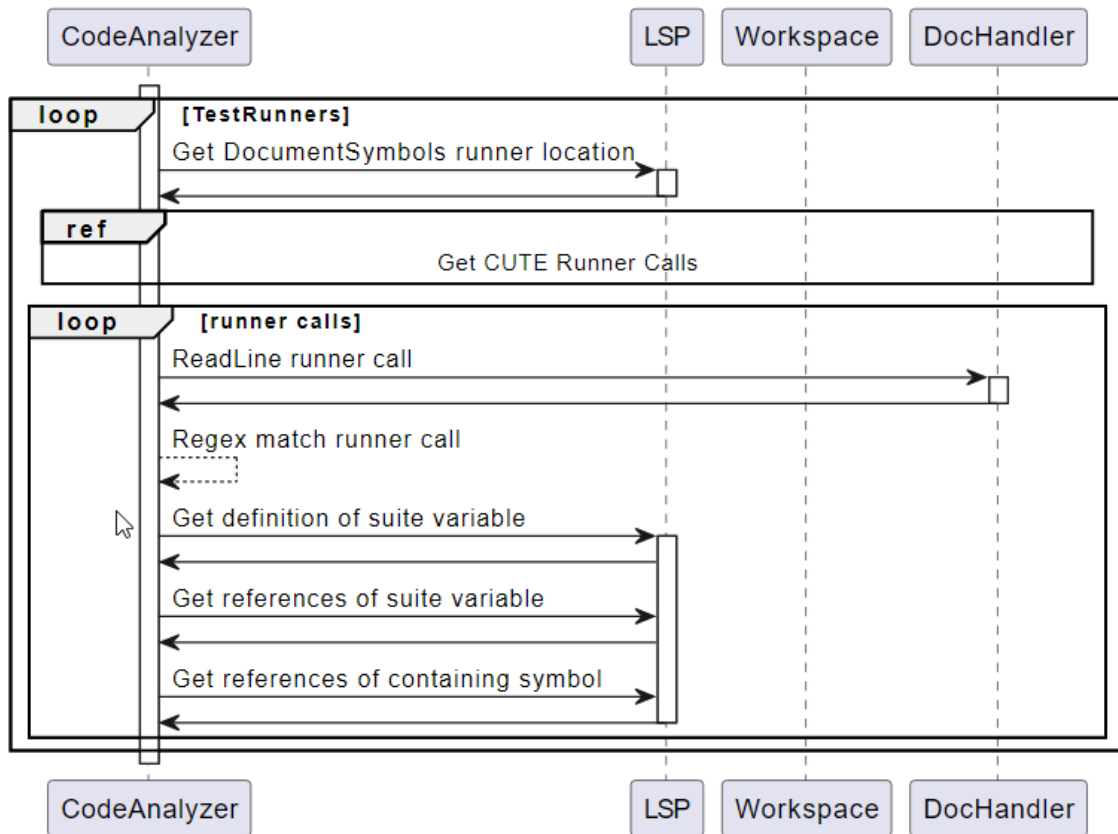


Figure 34: Find Test Suite sequence diagram

1. **Get Document Symbols** - get symbols of test files For each test runner instance, that is passed into this method, the symbols of the document containing the runner are queried by the language server provider using the language server protocol (LSP) [18]. These document symbols build the base to discover the test suites that are passed into the runner instances.
2. **Get Runner Calls** - get call locations of the test runners The CodeAnalyzer discovers all locations within a test file where a runner instance is called. The logic behind this discovery step is different for the CppTools [20] and Clangd [19] implementation, as the CppTools does not offer the possibility to discover references of operators. The detailed sequence of this step can be found in the diagram below. The listing below shows the two different runner call operators for which in this step the references are discovered.

```

1 bool operator()(const test & t) const
2 bool operator()(suite const &s, const char *info = "") const
  
```

Listing 22: CUTE Runner Call Operators

3. **Get Suites** - get test suites passed to runner calls Each previously discovered runner call is analyzed using regular expressions to split the command into the actual test suite and the additional parameters. The thereby gained information is used to find the suite definition and all suite variable references. The listing below shows an example runSuite function. In this example s is the runner variable for which the definition and references are discovered.

```

1 bool runSuite(cute::runner<cute::xml_listener<cute::ide_listener<>>> runner)
2 {
3     cute::suite s{};
4     s += testEnterHouse();
5     s += testHouseRooms();
6     s += testFail();
7
8     return runner(s, "TestSuite");
9 }

```

Listing 23: RunSuite Function

Get Runner Call Sequence - Find all runner calls

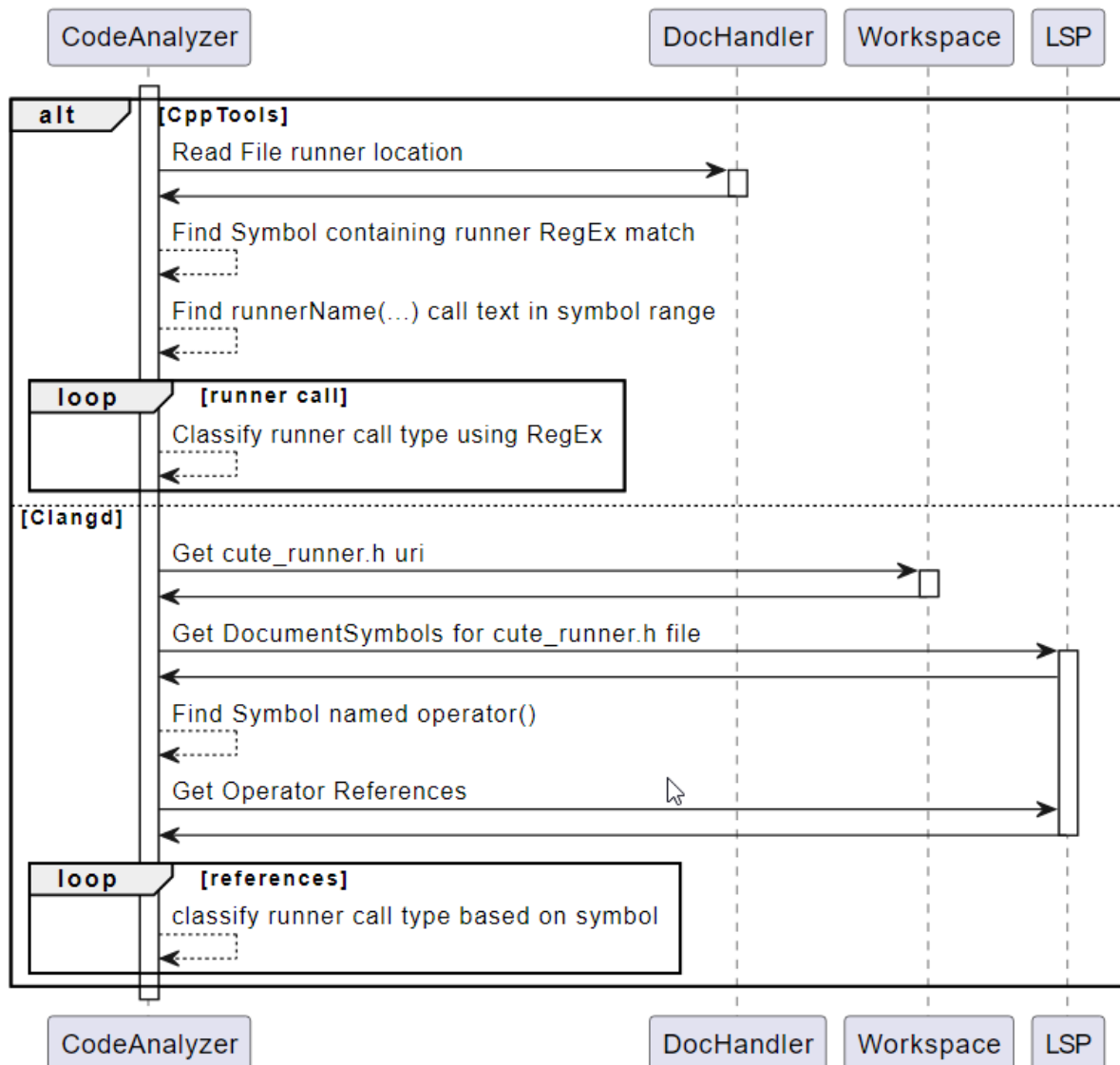


Figure 35: Get Runner Call sequence diagram

6.6.3 Find New Tests

The CodeAnalyzer provides functionality to find new CUTE tests that are defined using the new TEST(...) macro. The logic behind this functionality is based on information received from language server providers using the language server protocol (LSP) [18] and text-based analysis of certain code files or passages within the testing project. The diagram below provides an overview of the logic. Subsequent to the diagram, a description of each step can be found.

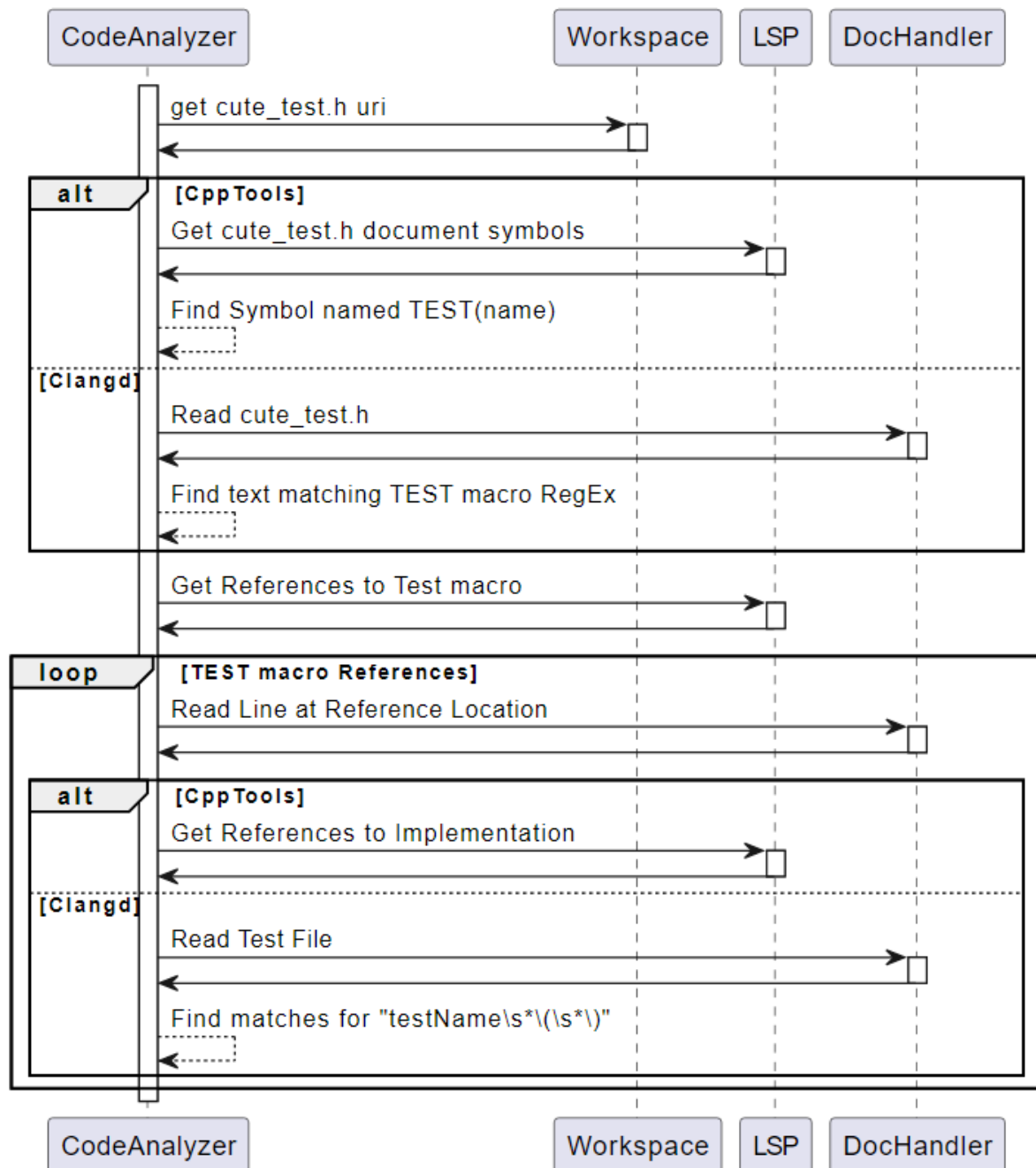


Figure 36: Find New Test sequence diagram

1. **Get cute_test.h uri** - find the cute test definition file The first step in the test discovering process is to find the cute_test.h file within the open workspace. This file is part of the CUTE testing framework and contains the implementation of a tests logic and amongst others, the newly defined TEST(...) macro, which can be used to define tests.
2. **Get TEST macro definition** - find the definition of the TEST macro The next step in this sequence is to find the location of the TEST(...) macro definition within the cute_test.h file. As the implementations of the supported language server providers show differences in the handling of macro symbols, this logic had to be implemented individually.
 - CppTools
If the CppTools extension is used as language server provider, a document symbol called "TEST(name)" is searched. CppTools includes macro definitions in the document symbol list.
 - Clangd
If the Clangd extension is used as language server provider, the document symbols do not contain macro definitions. A regular expression (RegEx [71]) based approach is chosen to find the definition of the TEST() macro.
3. **Get TEST macro references** - find all usage locations of the TEST macro The next step is to find the references of the previously discovered TEST macro. This functionality is provided by the language server provider over the language server protocol (LSP). The listing below shows an example test case that should be discovered during this step.

```
1 TEST(testHouseRooms)
2 {
3     House house{4};
4     ASSERT_EQUAL(4, house.getRooms());
5 }
```

Listing 24: Sample New CUTE TEST

4. **Get test usages** - find all locations where a test is called In order to be able to assign the test definitions to test suites, all references of a test definition need to be discovered. As the Clangd language server provider does not find the references of an element within a macro, the logic of this step had to be implemented individually.
 - CppTools
If the CppTools extension is used as language server provider, the references can easily be discovered using the therefore designed language server feature.
 - Clangd
If the Clangd extension is used as language server provider, the a regular expression (RegEx) based approach is chosen to find testName() calls within a test file.

6.6.4 Find Legacy Tests

The CodeAnalyzer provides functionality to find legacy CUTE tests that are defined using either the old CUTE(...) macro or constructor calls directly. The logic behind this functionality is based on information received from language server providers (LSP) and text-based analysis of certain code files or passages within the testing project. The diagram below provides an overview of the logic. Subsequent to the diagram, a description of each step can be found.

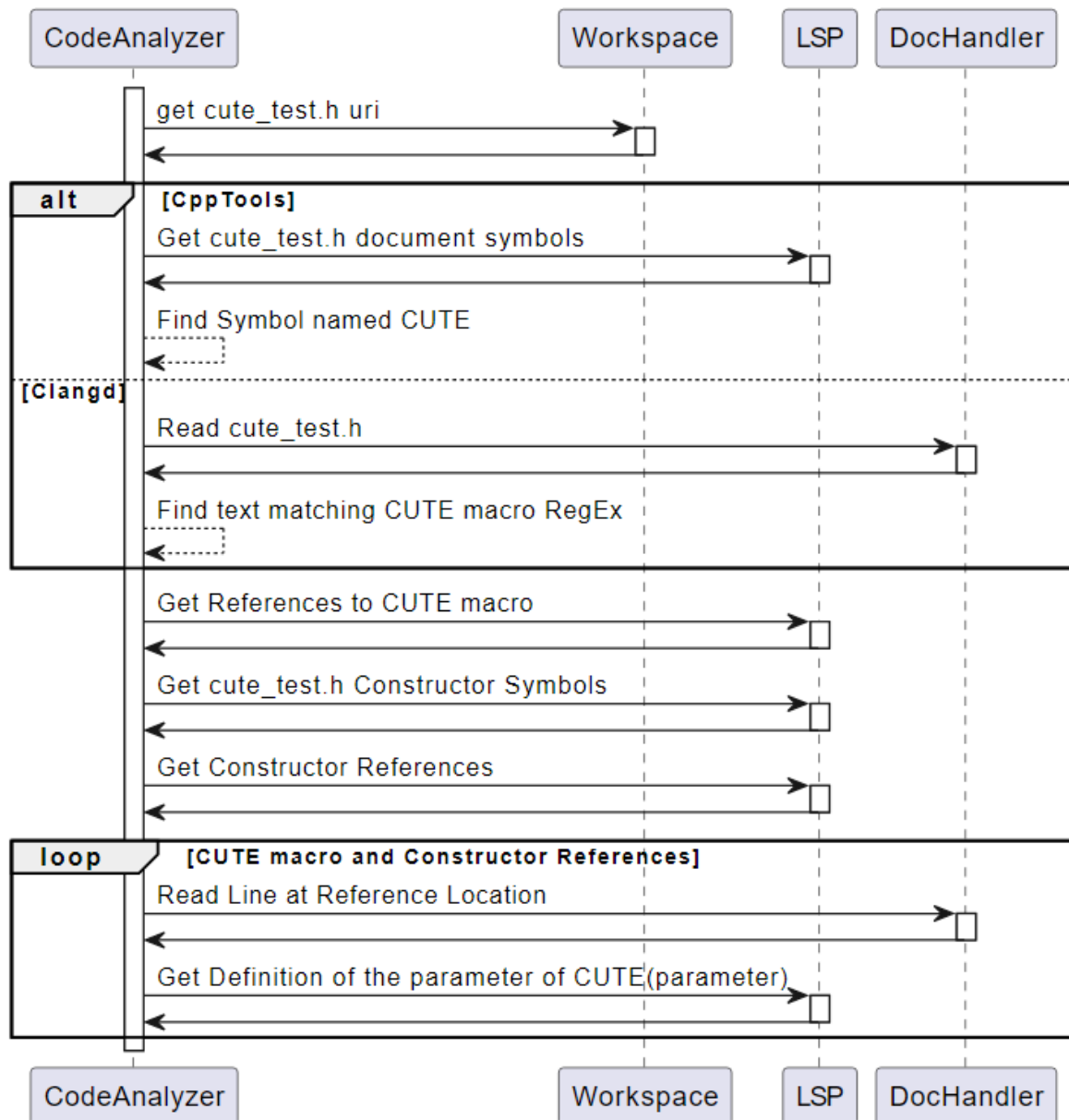


Figure 37: Find Legacy Test sequence diagram

1. **Get cute_test.h uri** - find the cute test definition file The first step in the legacy test discovering process is again to find the cute_test.h file [50] within the open workspace. This file is part of the CUTE testing framework and contains the implementation of a tests logic and amongst others, the CUTE(...) macro and the cute test constructors, which can be used to define tests.
2. **Get CUTE macro definition** - find the definition of the CUTE macro The next step in this sequence is to find the location of the CUTE(...) macro definition within the cute_test.h [50] file. As the implementations of the supported language server providers show differences in the handling of macro symbols, this logic had to be implemented individually.
 - CppTools
If the CppTools extension is used as language server provider, a document symbol called "CUTE(name)" is searched. CppTools includes macro definitions in the document symbol list.
 - Clangd
If the Clangd extension is used as language server provider, the document symbols do not contain macro definitions. A regular expression (Regex) based approach is chosen to find the definition of the CUTE() macro.
3. **GET CUTE test constructors and references** - find test constructor definitions and references As CUTE tests can also be created using constructor calls directly, the references to these constructors need to be found as well. To find the definition of the different constructors, the document symbols for the cute_test.h file are used. Once all constructor definition locations are found, the language server provider's cross reference functionality is used to discover all references of each constructor. The listing below shows the different cute test constructors available.

```

1  template<typename VoidFunctor>
2  test(VoidFunctor const & t, std::string sname = demangle(typeid(VoidFunctor).name()))
3      : name_(sname)
4        , theTest(t)
5  { }
6
7  template<typename VoidFunctor>
8  test(VoidFunctor const & t, location const & testLocation, std::string sname = ↵
9        demangle(typeid(VoidFunctor).name()))
10     : name_(sname)
11       , theTest(t)
12       , testLocation_(testLocation)
13  { }
14
15 template<typename VoidFunctor>
16 test(std::string sname, VoidFunctor const & t)
17     : name_(sname)
18       , theTest(t)
19 { }

```

Listing 25: CUTE Test Constructors [50]

4. **Get Test Implementation** - find the test implementation location To find a tests implementation, the line, on which the reference was found, gets read and analyzed using regular expressions. These RegEx patterns provide the location of the parameter which references the test implementation method. The language server provider's definition finding functionality is used to get the implementation location of a test, that got defined using the legacy CUTE(TestImplMethod) macro or a direct constructor call. The listing below shows a legacy test declaration using the CUTE macro. In this step the definition of the LegacyTest method is discovered.

```
1 void LegacyTest(){
2     House house{4};
3     ASSERT_EQUAL(4, house.getRooms());
4 }
5
6
7 bool runSuite(cute::runner<cute::xml_listener<cute::ide_listener<>>> runner)
8 {
9     cute::suite s{};
10    S += CUTE(LegacyTest);
11
12    return runner(s, "TestSuite");
13 }
```

Listing 26: Sample Legacy Test

6.7 CMakeParser Implementation

The CMakeParser class is not a real parser in every sense. This parser is a one character lookahead lexer that relates the command and the arguments. No token stream is returned and no syntax check is done here, only the commands with the respective arguments will be returned. The return value should then be passed to the CMakeLists class. Since this is not possible with RegEx [71] and for the bracketing a stack machine would be necessary, the decision was made to implement the CMakeParser in this form of implementation. The CMakeLists class takes the values returned by the CMakeParser class and allows querying for executables and makes assumptions about the test executable. This is possible via the relationships of the include paths.

7 Results

This chapter should provide an overview of the features, implemented in the CUTE extension for VS Code. Thereby each feature will be described individually regarding its general purpose, its benefits, and the specific use cases it supports. Subsequent to the description of all supported features, a comparison between the new extension's functionality (VS Code [4]) and the existing extension's functionality (Cevelop[3]) can be found.

7.1 Feature set

This section contains a description of all implemented features that are supported by the CUTE extension for Visual Studio Code. This overview should help to understand the possibilities, that are provided by this new testing extension.

7.1.1 Test Explorer

Every CUTE test that gets discovered by the therefore implemented logic within the extension, is shown in the Visual Code Test Explorer. This Test Explorer user interface was added to VS Code together with the official Testing API in version 1.59 [36]. It is available in the sidebar of Visual Studio Code and becomes visible as soon as a testing extension such as the CUTE extension is installed. The Test Explorer provides an easy to read and somewhat familiar overview about a project's tests. It further provides some useful features to the developers, such as the starting of test runs, containing all or a selection of tests. The discovery mechanism is triggered by changes of the executables, as they happen when compiling the project. The Test Explorer UI can also be refreshed manually. There are multiple different implementations of the test discovery logic, which will be discussed later in this chapter. Each thereby discovered test case has some properties such as its location within the project or the suite it belongs to. Using this information, the Test Explorer is able to offer functionalities such as the navigation to an individual test.

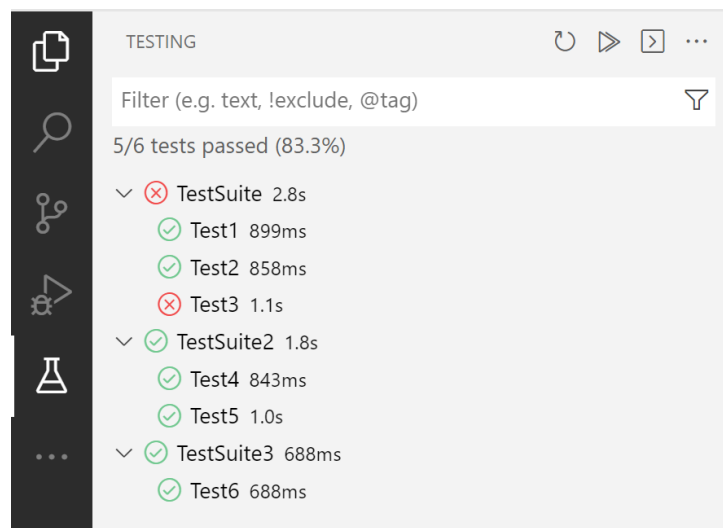


Figure 38: Test Explorer

7.1.2 Code Editor Integration

All discovered test cases are also made visible within the code. On the left side of each test implementation within a cpp file, a green arrow is shown. This arrow first of all helps a developer to spot test implementations easier, what massively improves the user experience. On the other hand, this icon in the code editor allows developers to start test runs directly from within the code. The user can select whether he would like to start the test normally or with a debugger attached if a supported debugging extension is available. The user also has the possibility to show the test within the Test Explorer. All the earlier described functionality of starting. The arrow icon changes to a symbol that describes the tests' last outcome. (Green if succeeded, red if failure etc.) A further feature is the visualization of a tests' outcomes in earlier runs.

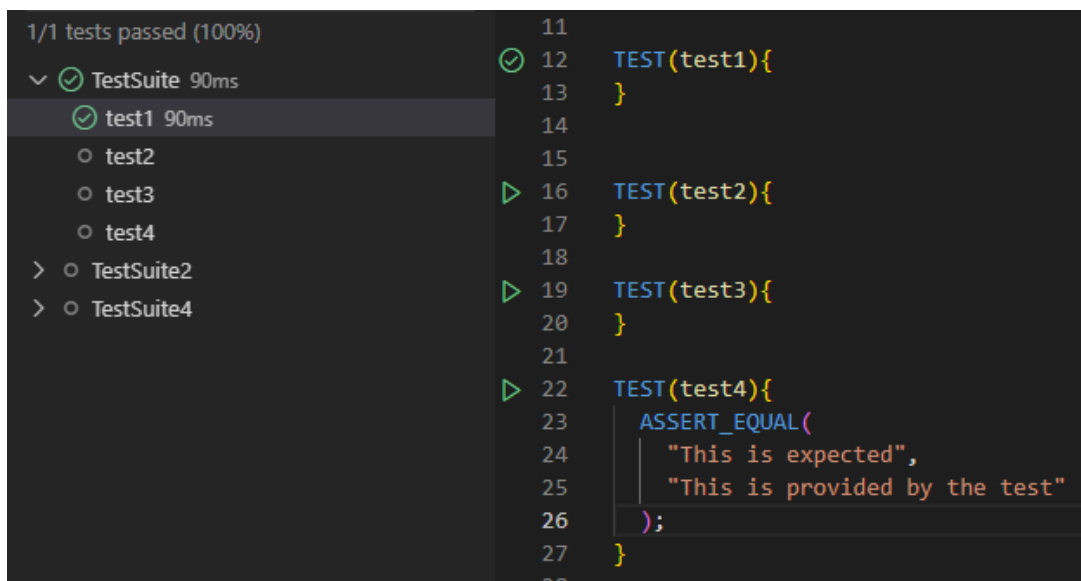


Figure 39: VS Code CUTE inline integration

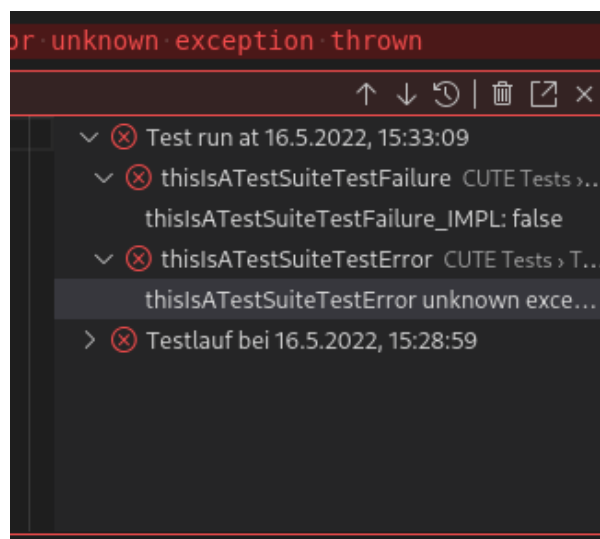


Figure 40: VS Code CUTE run history

7.1.3 Project Generation

The CUTE extension for VS Code offers the functionality to generate empty CMake projects, which can be used as a starting point for new C++ projects. When a developer chose to generate a new project, the CUTE extension provides the basic structure, sets up the empty project and configures it. The only constraint to use this feature is, that a single empty workspace needs to be opened in VS Code. This feature allows it developers without any knowledge about CMake [9], to set up a project containing an executable, a library, and a test executable. There is no need for the developer to configure the project first and therefore he can start developing C++ code right away. The basic structure of the project contains an executable, a library, that contains the application logic and the test-executable, which contains the CUTE headers, which are required to write CUTE framework-based unit tests. The headers themselves are delivered together with the CUTE extension and therefore no internet connection is required while generating a new project.

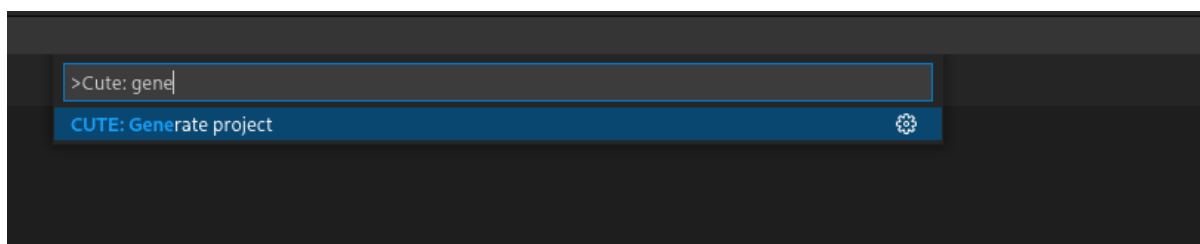


Figure 41: The VS Code CUTE project generator

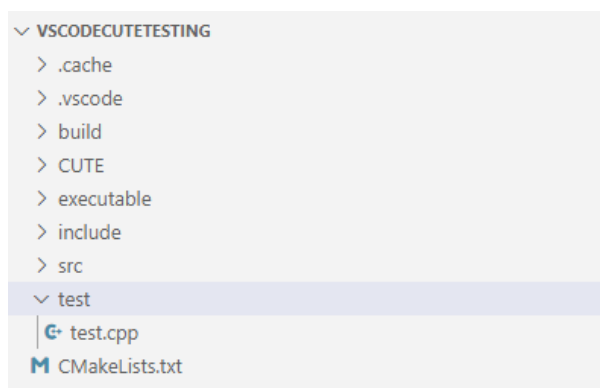


Figure 42: Structure of a newly generated project

7.1.4 Legacy Syntax Converter

The CUTE extension offers the functionality to convert existing test projects, that use the old CUTE macro for the test definition, to updated projects, which use the new TEST macro syntax for the test definition. This functionality removes the old CUTE macro from a specific test definition and wraps that tests' function header into the new TEST macro. Using the new TEST macro allows it to get the required location information for a test directly from the test executable without having to use any language server providers. The CUTE extension offers multiple ways of dealing with legacy code definitions. The user can choose if all discovered legacy tests should be converted at once or whether he would like to convert one by one. There is also an option to ignore these legacy test declaration warnings. If a user chose this option, he accepts that not all functionality is available (Test navigation and editor integration won't be available). The ignoring choice can be undone in the CUTE extension preferences.

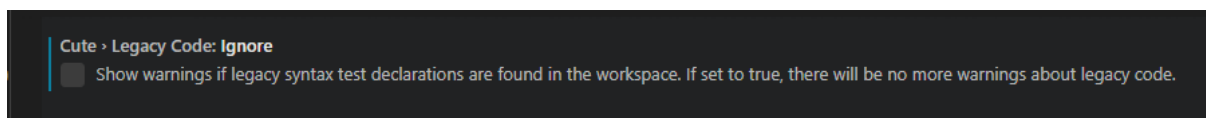


Figure 43: Legacy Syntax Settings

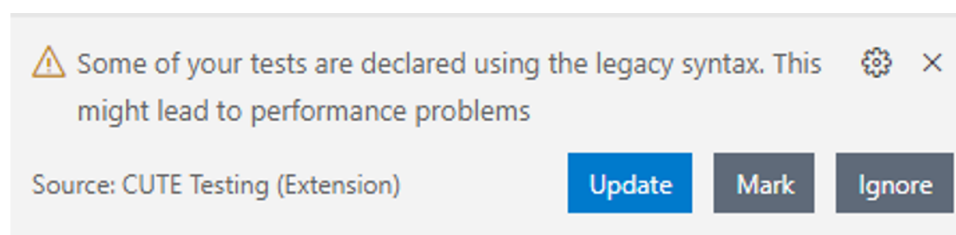


Figure 44: Legacy Syntax Update pop-up

7.1.5 Assert Failure Analysis

The CUTE extension offers a visual option to compare failed assertions. Thereby the expected and the actual values are displayed side by side. The differences between these two values are then highlighted, what draws a user's attention directly to the problematic spot within the values. This simplifies the understanding of why an assertion failed. If a test should have failed for any reason that is not related to equality comparisons, the failure reason is simply provided in textual form.

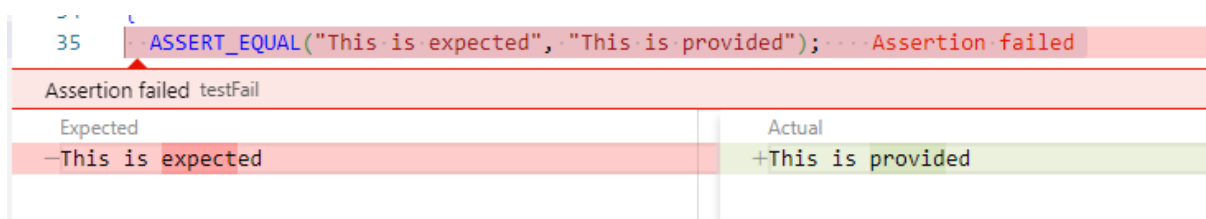


Figure 45: Assertion Failure

7.1.6 Quick Fixes

The CUTE extension for VS Code offers some additional quick fixes, which support the users with writing test projects that work as intended. Each of these provided quick fixes can be used to mitigate a code problem, that is marked within the code and in the problems view. Below the different warnings with their corresponding quick fixes are described.

Unregistered Test Case

The CUTE extension for Visual Studio Code analyzes the test project's code for tests that are not registered to any test suite and are not called directly. If such an unregistered test case should be found, its location is marked within the code and entry in the Problems view is created. When hovering the marked test declaration in the editor, a message shows up and tells the user that this test will never be called. The CUTE extension provides quick fixes that are based on the discovered test suites and offers to register this test case in one of these suites. Further, the extension provides an additional quick fix that allows the creation of a new test suite within the current file. This quick fix automatically registers the test in the newly created suite. The figure below shows an unregistered test warning in the editor and the Problems view with the offered quick fixes.

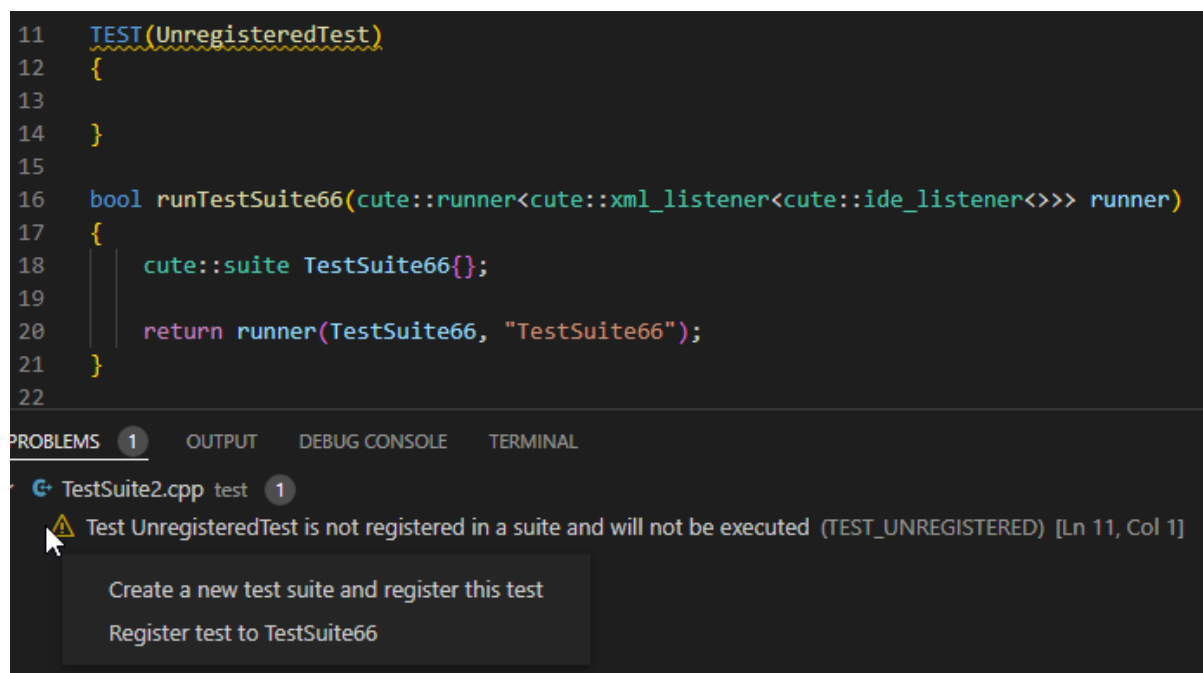


Figure 46: Unregistered Test Editor Warning

Register Test Case to existing suite

The first option to mitigate the unregistered test problem using a quick fix is to register the test case to an existing test suite. For each discovered test suite in the tests' file a quick fix option is provided. If one of those quick fix options is chosen, the extension adds the registration command of the test to the chosen test suite. The figure below shows the fixed code with the newly added registration command highlighted in blue. After the file is saved, the unregistered test warning will disappear.

```
12  TEST(UnregisteredTest)
13  {
14
15  }
16
17  bool runTestSuite66(cute::runner<cute::xml_listener<cute::ide_listener<>>> runner)
18  {
19      cute::suite TestSuite66{};
20      TestSuite66 += UnregisteredTest();
21
22      return runner(TestSuite66, "TestSuite66");
23  }
24
```

Figure 47: Unregistered Test Existing Suite Fix

Register Test Case to new suite

The second option to mitigate the unregistered test problem using a quick fix is to register the test case to a new test suite. If this quick fix option is chosen, the extension creates a new runSuite function containing the declaration of a new test suite. This function is created using a code snippet, that allows the users to set a custom suite name. The test case is registered automatically to this newly created test suite by the CUTE extension. The figure below shows the newly created runSuite function containing the test suite and the registration command. All the highlighted sections are updated when the user sets the suite name.

```
24  bool runNewSuite(cute::runner<cute::xml_listener<cute::ide_listener<>>> runner)
25  {
26      cute::suite NewSuite{};
27
28      NewSuite += UnregisteredTest();
29
30      return runner(NewSuite, "NewSuite");
31  }
```

Figure 48: Unregistered Test New Suite Fix

Unregistered TestSuite

The CUTE extension for Visual Studio Code analyzes the test project's code for test suites that are never called. Thereby the test suites need to be within a method that has a signature similar to the `runTestSuite` method in the figure below, in order to be analyzed. If the CUTE extension discovers such an unregistered test suite, it marks the location of the containing method within the code and adds an entry to the Problems view. When hovering the marked method, a message is shown, which tells the users that this test suite is never called and therefore none of its contained tests will be executed. The CUTE extension provides a quick fix to add a test suite call in the main function with an existing test runner, which might be defined in that function. The figure below shows an unregistered suite warning in the editor and the Problems view together with the available quick fixes.

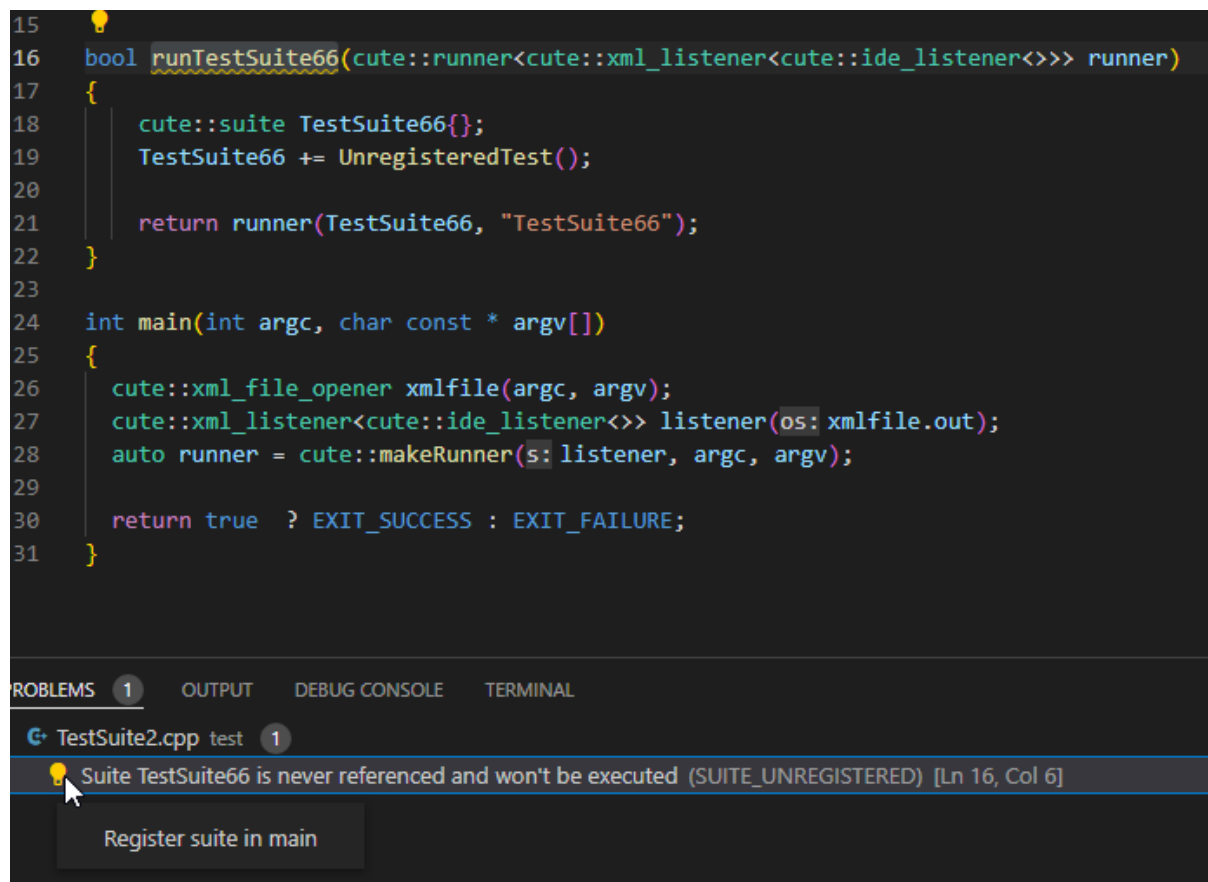


Figure 49: Unregistered Suite Editor Warning

Register Suite in main

The CUTE extension provides a quick fix to mitigate the unregistered suite problem. If this quick fix is chosen, the extension adds a call of the `runSuite` function to the main function. The figure below shows the fixed code after the quick fix was applied. The highlighted code is created or updated by the extension. Line 31 is newly created and contains a call to the `runSuite` function with the existing runner as parameter. If no test runner declaration should be available in the main function, the parameter of the `runSuite` function call is undefined and needs to be set manually by the user. Line 32 contains the return statement of the main function. The newly added test suite call gets integrated into this return statement to ensure the correct return value of the test executable. If there should be existing test suite calls in the main function the newly added call, simply gets added to the return statement with a logical and `&&` operator in front of the ternary `?` operator.

```
17 bool runTestSuite66(cute::runner<cute::xml_listener<cute::ide_listener<>>> runner)
18 {
19     cute::suite TestSuite66{};
20     TestSuite66 += UnregisteredTest();
21
22     return runner(TestSuite66, "TestSuite66");
23 }
24
25 int main(int argc, char const * argv[])
26 {
27     cute::xml_file_opener xmlfile(argc, argv);
28     cute::xml_listener<cute::ide_listener<>> listener(os: xmlfile.out);
29     auto runner = cute::makeRunner(s: listener, argc, argv);
30
31     bool runTestSuite66_result = runTestSuite66(runner);
32     return runTestSuite66_result ? EXIT_SUCCESS : EXIT_FAILURE;
33 }
```

Figure 50: Unregistered Suite Fix

CUTE Legacy Syntax

The CUTE extension for Visual Studio Code analyzes the test project's code for test declarations that are based on the old CUTE macro. If the extension should find such a test declaration using the legacy syntax, it marks the test implementation function within the code and adds and entry to the Problems view. In addition, a warning is shown in the form of a pop-up. This warning pop-up offers the option to update all legacy syntax test declarations at once or ignore the legacy syntax and show no more warnings. When hovering the marked test implementation, a message is shown to tell the users that the `TEST(...)` macro should be used to declare tests. The CUTE extension provides a quick fix to update the syntax from the CUTE macro to the new `TEST` macro.

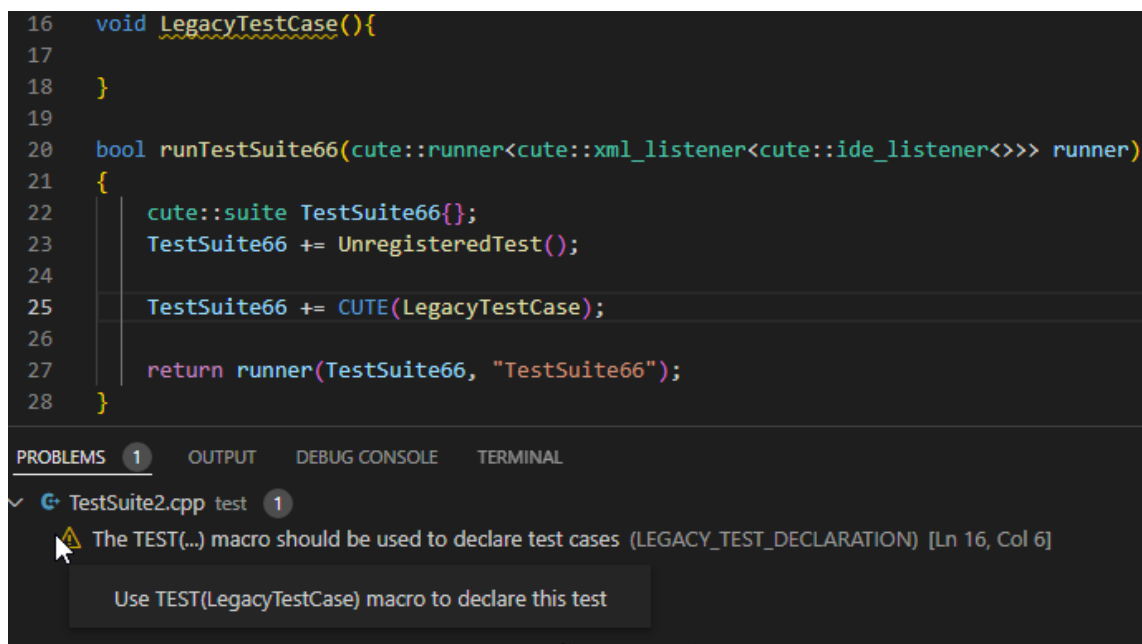


Figure 51: Legacy Syntax Editor Warning

Update Legacy Syntax

The CUTE extension provides a quick fix to mitigate the legacy syntax warnings. If the user chooses to update the CUTE macro test declarations, either all at once or test by test, the cute extension discovers the test implementation and wraps it in the new `TEST` macro. The extension also replaces the old CUTE macro test syntax with a simple function call to the newly wrapped test function. The figure below shows the updated code after the legacy syntax update quick fix was applied. The highlighted line shows the newly wrapped test function and on line 20, the updated test call can be found.



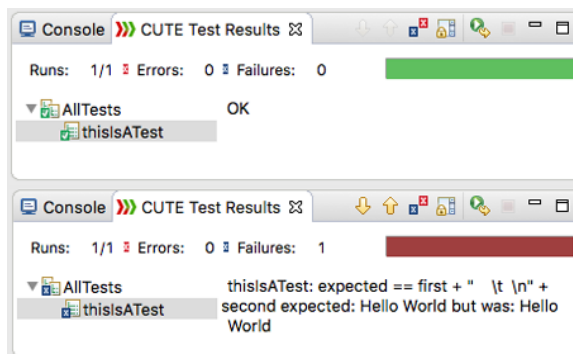
Figure 52: Legacy Syntax Fix

7.2 Feature comparison

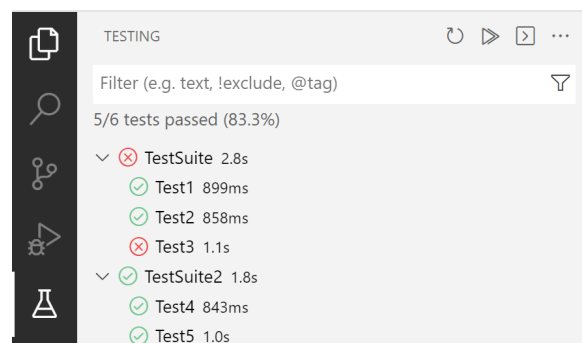
In this section the feature sets of the VS Code extension and the existing Clevelop extension [5] are compared. The table below shows the full set of features, that are available either in the CUTE extension for Clevelop or in the new CUTE extension for VS Code. This feature comparison makes it obvious that the new CUTE extension for VS Code offers additional features to make the usage of CUTE as comfortable as possible. After the following table, there will be a more detailed comparison of certain key features offered by both extensions.

	CUTE Extension for Visual Code	CUTE Extension for Clevelop
Functional		
View tests in test explorer	✓	✓
Interactive integration in code editor	✓	✗
Generation of a template CMake project	✓	✗
Single and parallel test execution	✓	✓
Diff-view of different strings	✓	✓
Analytics		
Check for legacy tests	✓	Not necessary
Check suite call	✓	✗
Check test assignment to suite	✓	✓
Quickfixes		
Convert legacy syntax to new syntax	✓	Not necessary
Create new testsuite and register test	✓	✗
Register test to known testsuite	✓	✓

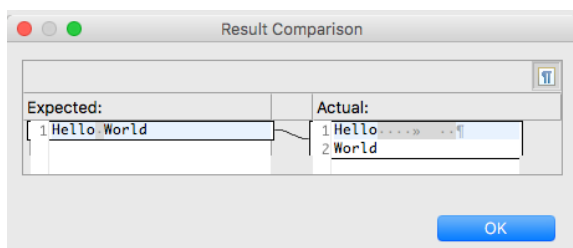
The images below show a side by side comparison of the comparable features of the Clevelop plug-in and the new VS Code extension. The comparable features are the Tests Explorer with included Red/Green bar experience and the difference viewer for failed assertions.



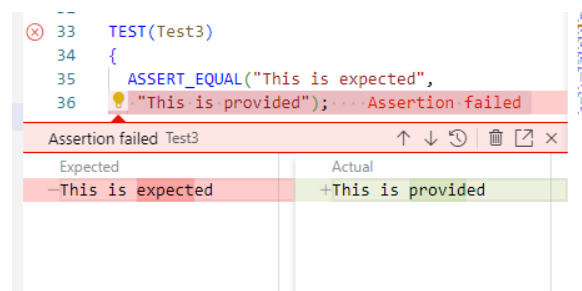
(a) Red/Green Bar Clevelop



(b) Red/Green Bar VS Code



(c) Difference viewer Clevelop



(d) Difference viewer VS Code

Figure 53: Feature Comparison

7.3 Metrics

This section provides an overview of the CUTE extension's metrics. The code which is used to implement the functionalities of the CUTE extension for Visual Studio Code is approximately 3000 lines long, contains 47 classes distributed over 57 files. The test coverage that was achieved lays with 81.6% over the aspired 80% mark.

The figure below provides an overview of the project's size. All the shown metrics within this chapter are provided by SonarQube [14] and were captured after the master build that lead to version 1.0.0 of the CUTE extension for VS Code.

Lines of Code	3,143
Lines	3,735
Statements	1,046
Functions	372
Classes	47
Files	57
Comment Lines	2
Comments (%)	0.1%

Figure 54: SonarQube Project Size

The figure below provides an overview of the test coverage on component level provided by SonarQube [14]. The coverage result deviates slightly from the actual coverage value as not all test can be run in the pipeline due to the fact that not all configurations are possible there. E.g. the windows focused logic will not be tested in the pipeline even though tests exist for these code passages.

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage
src	3,143	0	0	10	0	81.6%
environment	418	0	0	0	0	83.0%
test	—	0	0	0	0	—
testing	846	0	0	3	0	85.8%
tools	662	0	0	3	0	85.9%
utilities	1,044	0	0	3	0	82.4%
extension.ts	173	0	0	1	0	51.4%

Figure 55: Component Coverage Result

The figure below provides an overview of the general code metrics provided by SonarQube [14]. Thereby it can be seen, that no bugs have been found which leads to a reliability rating A. Further no vulnerabilities have been detected, therefore the security metric is also rated A. SonarQube detected a few code smells, all of them have to do with the chosen code style guidelines and the linter used in the pipeline. All of the detected code smells were analyzed and afterwards marked to ignore. The maintainability of the project is rated A as well.

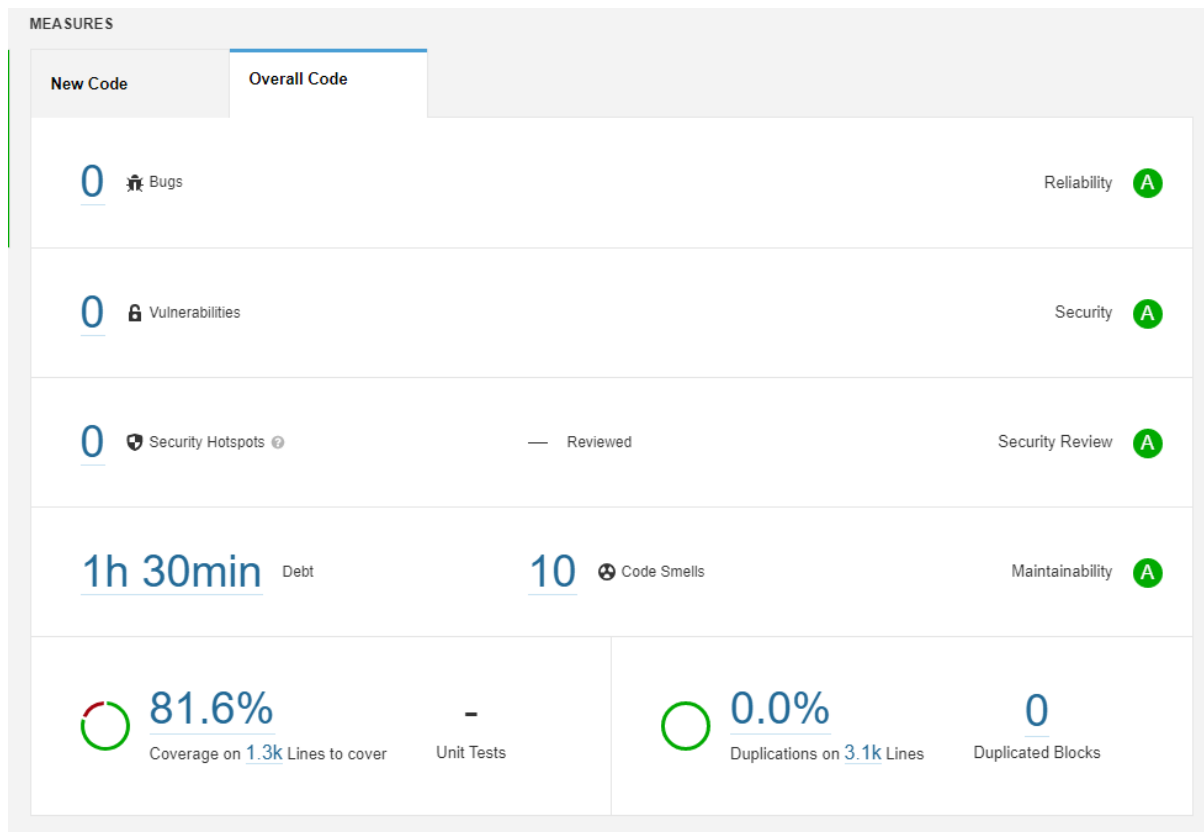


Figure 56: SonarQube Project Metrics

8 Testing

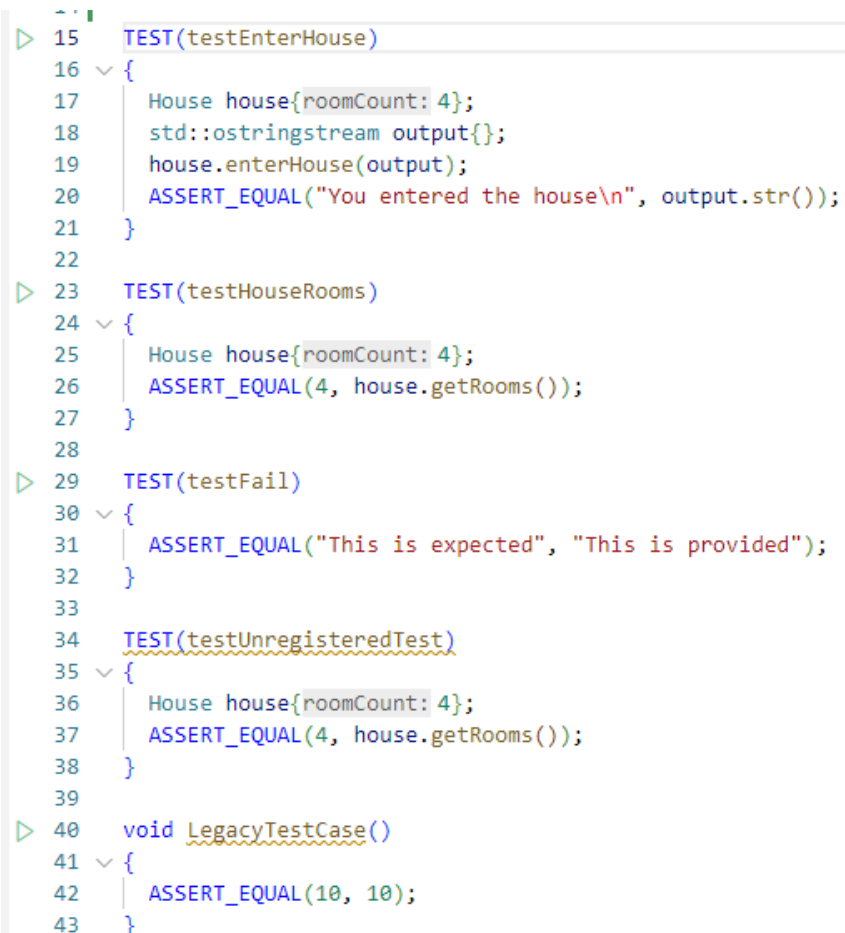
This chapter contains a description of the manual system tests of the CUTE extension for VS Code. The here defined manual tests are split into tests regarding the functional requirements and tests regarding the nonfunctional requirements (NFR). These tests should further help to ensure the quality of the CUTE extension. The tests should always be executed before merging changes back into the master branch to ensure that the existing functionality was not broken by the changes.

8.1 Functionality Testing

The first section of this chapter contains the functional test case definitions. Each test definition consists of a general description, a set of preconditions that need to be fulfilled before starting the test, instructions about all steps that are part of the test and the expected output or behavior of the extension. Subsequent to the test descriptions, a test protocol can be found, that covers all defined tests and provides information about each tests' outcome.

Test-1: code-based test discovery Clangd

Description:	This test should ensure that the code-based test discovery works in combination with the Clangd [19] extension as language server provider. All tests of the test project should be discovered and registered to the test explorer. The discovered tests should be marked in the code by the extension.
Preconditions:	<ul style="list-style-type: none">• Clangd extension installed and activated• CUTE extension installed and activated• TestFinderMode configuration is set to codeBased• cuteExtensionTestProj folder opened as workspace
Input / Interaction:	<ol style="list-style-type: none">1. Build the CMake project in the workspace2. Open the test explorer on the left3. Test discovery is started automatically4. Wait until some tests are displayed in the test explorer
Expected Output:	The code-based implementation of the test finder logic discovers four tests and registers them in the test explorer. The tests need to be marked as in the image below. A green arrow is expected on line 15, 23, 29, and 40. Make sure that Line 34 is not marked.



```
15  TEST(testEnterHouse)
16  {
17      House house{roomCount: 4};
18      std::ostringstream output{};
19      house.enterHouse(output);
20      ASSERT_EQUAL("You entered the house\n", output.str());
21  }
22
23  TEST(testHouseRooms)
24  {
25      House house{roomCount: 4};
26      ASSERT_EQUAL(4, house.getRooms());
27  }
28
29  TEST(testFail)
30  {
31      ASSERT_EQUAL("This is expected", "This is provided");
32  }
33
34  TEST(testUnregisteredTest)
35  {
36      House house{roomCount: 4};
37      ASSERT_EQUAL(4, house.getRooms());
38  }
39
40  void LegacyTestCase()
41  {
42      ASSERT_EQUAL(10, 10);
43  }
```

Figure 57: Test Explorer Clangd Code-Based Discovery Editor

Test-2: code-based test discovery CppTools

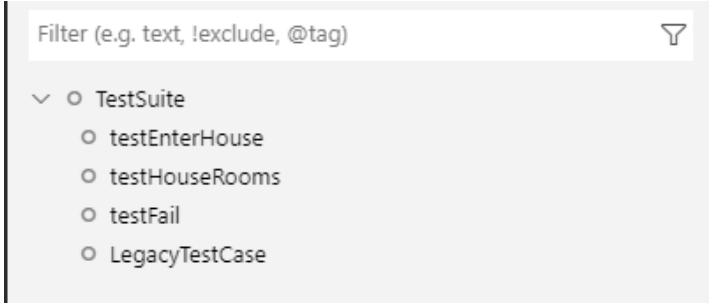
Description:	This test should ensure that the code-based test discovery works in combination with the CppTools [20] extension as language server provider. All tests of the test project should be discovered and registered to the text explorer. The discovered tests should be marked in the code by the extension.
Preconditions:	<ul style="list-style-type: none">• C/C++ extension for VS Code (CppTools) installed and activated• CUTE extension installed and activated• TestFinderMode configuration is set to codeBased• Language Client Mode is set to Explicit• cuteExtensionTestProj folder opened as workspace
Input / Interaction:	<ol style="list-style-type: none">1. Build the CMake project in the workspace2. Open the test explorer on the left3. Test discovery is started automatically4. Wait until some tests are displayed in the test explorer
Expected Output:	<p>The code-based implementation of the test finder logic discovers four tests and registers them in the test explorer. The test explorer is expected to look similar to the one shown in the image below. The tests are expected to be marked in the same way as described in Test-1.</p>  <p>The screenshot shows the VS Code Test Explorer interface. At the top, there is a search bar labeled 'Filter (e.g. text, !exclude, @tag)' with a funnel icon on the right. Below the search bar, a tree view is displayed. It starts with a collapsed 'TestSuite' (indicated by a downward arrow and a circle). Expanding it shows four test items, each with a circle icon: 'testEnterHouse', 'testHouseRooms', 'testFail', and 'LegacyTestCase'.</p>

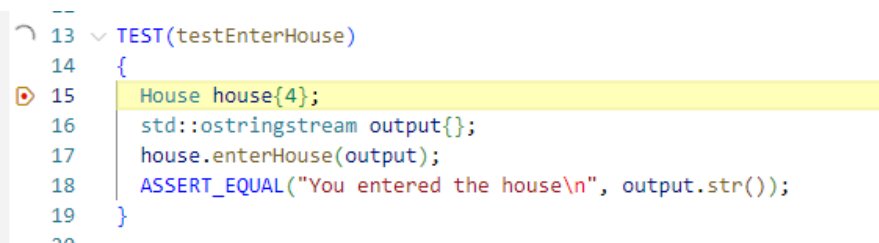
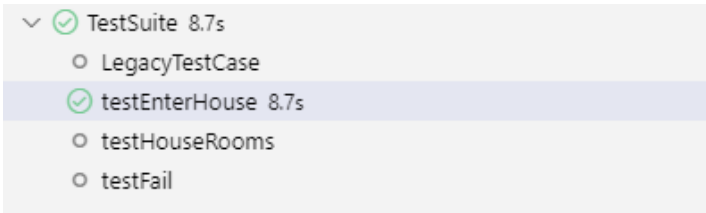
Figure 58: Test Explorer CppTools Code-Based Discovery Explorer

Test-3: executable based test discovery

Description:	This test should ensure that the executable based implementation of the test discovery logic works. All tests of the project should be discovered and registered to the text explorer. The discovered tests should be marked in the code by the extension.
Preconditions:	<ul style="list-style-type: none">• CUTE extension installed and activated• TestFinderMode configuration is set to executableBased• cuteExtensionTestProj folder opened as workspace
Input / Interaction:	<ol style="list-style-type: none">1. Build the CMake project in the workspace2. Open the test explorer on the left3. Test discovery is started automatically4. Wait until some tests are displayed in the test explorer
Expected Output:	<p>The executable-based implementation of the test finder logic discovers the same four tests as the code-based variants and registers them in the test explorer. The test explorer is expected to look similar to the one shown in the expected output of Test-2. The tests are expected to be marked as shown in the image below. Important to note is that neither line 32 is marked, as this test is not registered, nor line 38 is marked as there is no location information for legacy tests.</p> <pre>13 ▸ TEST(testEnterHouse) 14 { 15 House house{4}; 16 std::ostringstream output{}; 17 house.enterHouse(output); 18 ASSERT_EQUAL("You entered the house\n", output.str()); 19 } 20 21 ▸ TEST(testHouseRooms) 22 { 23 House house{4}; 24 ASSERT_EQUAL(4, house.getRooms()); 25 } 26 27 ▸ TEST(testFail) 28 { 29 ASSERT_EQUAL("This is expected", "This is provided"); 30 } 31 32 ▸ TEST(testUnregisteredTest) 33 { 34 House house{4}; 35 ASSERT_EQUAL(4, house.getRooms()); 36 } 37 38 ▸ void LegacyTestCase() 39 { 40 ASSERT_EQUAL(10, 10); 41 } 42</pre>

Figure 59: Test Explorer Executable-Based Discovery Editor

Test-4: CUTE test debugging test

Description:	This test should ensure that the CUTE test debugger functionality is available through the CUTE extension. The test explorer should provide an option to start a test run in the debug configuration. Starting a debug run should also be possible from within the code editor. The execution of the code should break at breakpoints.
Preconditions:	<ul style="list-style-type: none">• C/C++ extension for VS Code (CppTools) installed and activated• CUTE extension installed and activated• cuteExtensionTestProj folder opened as workspace• CMake project is built• Tests are discovered and loaded into the test explorer
Input / Interaction:	<ol style="list-style-type: none">1. Set a breakpoint in the test.cpp file on line 152. Select the testEnterHouse test in the test explorer and select start debugging3. Wait until the test run started
Expected Output:	<p>The CUTE extension starts a debugging process using the installed debugging extension. After the view in VS Code is changed to the debugging view, the test debugging starts. After a while the earlier set breakpoint is hit and the program waits there. The test duration shown in the test explorer is as long as the debugging of the test takes.</p>  <p>Figure 60: Test Debugging Breakpoint</p>  <p>Figure 61: Test Debugging Result</p>

Test-5: CUTE test run test

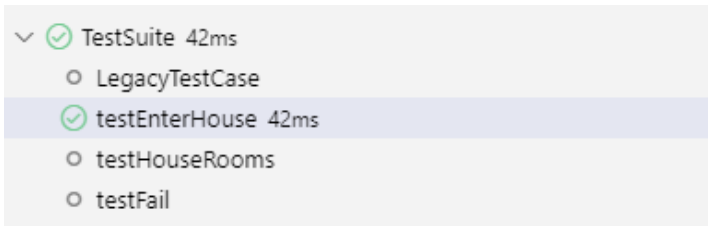
Description:	This test should ensure that single tests can be started using the test explorer. After the test run has finished, the result, for the test included in the run, should be displayed in the test explorer and in the code editor.
Preconditions:	<ul style="list-style-type: none">• CUTE extension installed and activated• cuteExtensionTestProj folder opened as workspace• CMake project is built• Tests are discovered and loaded into the test explorer
Input / Interaction:	<ol style="list-style-type: none">1. Set a breakpoint in the test.cpp file on line 152. Select the testEnterHouse test in the test explorer and select run test3. Wait until the test run finished
Expected Output:	<p>The CUTE extension starts a test run containing a single test. The test run starts quickly and this time, the program does not wait at the earlier set breakpoint. The result is visualized in the test explorer and in the editor window next to the test definition. (Location where the green arrow was)</p>  <p>The screenshot shows a test explorer window with a tree view. The root node is 'TestSuite 42ms' with a green checkmark icon. It has four child nodes: 'LegacyTestCase', 'testEnterHouse 42ms', 'testHouseRooms', and 'testFail'. The 'testEnterHouse 42ms' node is selected and highlighted with a blue background, and it also has a green checkmark icon.</p>

Figure 62: Test Run Single Test

Test-6: CUTE suite run test

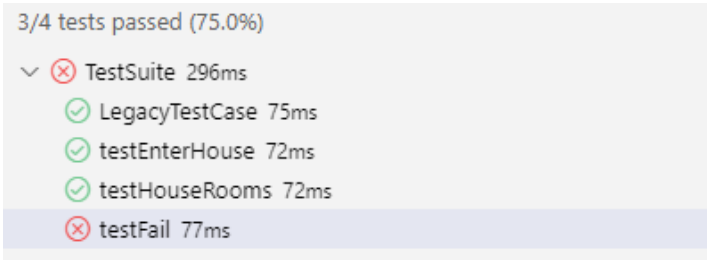
Description:	This test should ensure that all tests that belong to a test suite can be started using the test explorer. All tests should start running in parallel. After the test run has finished, for all the tests in the test suite a result should be displayed.
Preconditions:	<ul style="list-style-type: none">• CUTE extension installed and activated• cuteExtensionTestProj folder opened as workspace• CMake project is built• Tests are discovered and loaded into the test explorer
Input / Interaction:	<ol style="list-style-type: none">1. Select the TestSuite test suite in the test explorer and select run tests2. Wait until the test run finished
Expected Output:	<p>The CUTE extension starts a test run containing all four tests belonging to TestSuite1. The test run starts quickly and all tests start being executed at the same time. The result is visualized in the test explorer and in the editor window next to the test definition. (Location where the green arrow was)</p>  <p>3/4 tests passed (75.0%)</p> <ul style="list-style-type: none">✓ LegacyTestCase 75ms✓ testEnterHouse 72ms✓ testHouseRooms 72ms✗ testFail 77ms

Figure 63: Test Run Test Suite Tests

Test-7: unregistered test existing suite quick fix

Description:	This test should ensure that the quick fix to mitigate the unregistered test problem works correctly. In this test, an unregistered test case should be registered to an existing test suite.
Preconditions:	<ul style="list-style-type: none">• Clangd extension installed and activated• CUTE extension installed and activated• cuteExtensionTestProj folder opened as workspace• CMake project is built
Input / Interaction:	<ol style="list-style-type: none">1. Open test.cpp file in the editor2. Save the document to immediately trigger the code analysis for the file3. Hover the marked TEST(testUnregisteredTest) line and select quick fixes -> add test to TestSuite4. Save the document again
Expected Output:	<p>The unregistered test is detected and marked correctly as shown in the image below. The quick fix to register the unregistered test case to TestSuite2 is available. After the quick fix was applied and the document saved, the test is no longer marked as unregistered. A registration call was made in the runSuite2 function.</p> <pre>32 TEST(testUnregisteredTest) 33 { 34 House house{roomCount: 4}; 35 ASSERT_EQUAL(4, house.getRooms()); 36 } --</pre> <p>Figure 64: Unregistered Test Warning</p> <pre>55 bool runSuite2(cute::runner<cute::xml_listener<cute::ide_listener<>>> runner) 56 { 57 cute::suite s2{}; 58 s2 += testUnregisteredTest(); 59 60 return runner(s2, "TestSuite2"); 61 }</pre> <p>Figure 65: Unregistered Test QuickFix 1</p>

Test-8: unregistered test new suite quick fix

Description:	This test should ensure that the quick fix to mitigate the unregistered test problem works correctly. In this test, an unregistered test case should be registered to a newly created test suite within the tests' file.
Preconditions:	<ul style="list-style-type: none">• Clangd extension installed and activated• CUTE extension installed and activated• cuteExtensionTestProj folder opened as workspace• CMake project is built
Input / Interaction:	<ol style="list-style-type: none">1. Open test.cpp file in the editor2. Save the document to immediately trigger the code analysis for the file3. Hover the marked TEST(testUnregisteredTest) line and select quick fixes -> create new suite and add test4. Enter suite name5. Save the document again
Expected Output:	<p>The unregistered test is detected and marked correctly as shown in the image in Test-7. The quick fix to register the unregistered test case to a new test suite is available. After the quick fix was applied and the document saved, the test is no longer marked as unregistered. The runSuite function for the new suite was created and looks as shown in the image below.</p> <pre>62 bool runNewSui(cute::runner<cute::xml_listener<cute::ide_listener<>>> runner) 63 { 64 cute::suite NewSui{}; 65 66 NewSui += testUnregisteredTest(); 67 68 return runner(NewSui, "NewSui"); 69 }</pre>

Figure 66: Unregistered Test Quick Fix 2

Test-9: unregistered suite quick fix

Description:	This test should ensure that the quick fix to mitigate the unregistered suite problem works as intended. In this test, an unregistered test suite should be registered to a test runner in the main function of a test project.
Preconditions:	<ul style="list-style-type: none">• Clangd extension installed and activated• CUTE extension installed and activated• cuteExtensionTestProj folder opened as workspace• CMake project is built
Input / Interaction:	<ol style="list-style-type: none">1. Open test.cpp file in the editor2. Save the document to immediately trigger the code analysis for the file3. Hover the marked runSuite2 text on line 55 and select quick fixes -> register suite in main4. Save the document again
Expected Output:	<p>The unregistered test suite is detected and marked correctly as shown in the image below. The quick fix to register the unregistered test suite in the main function is available. After the quick fix was applied and the document was saved, the suite is no longer marked as unregistered. The registration call looks as shown in the second figure below.</p> <pre>55 bool runSuite2(cute::runner<cute::xml_listener<cute::ide_listener<>>> runner) 56 { 57 cute::suite s2{}; 58 59 return runner(s2, "TestSuite2"); 60 } 61 ~~~~~</pre> <p>Figure 67: Unregistered Suite Warning</p> <pre>55 bool runSuite2(cute::runner<cute::xml_listener<cute::ide_listener<>>> runner) 56 { 57 cute::suite s2{}; 58 59 return runner(s2, "TestSuite2"); 60 } 61 62 int main(int argc, char const * argv[]) 63 { 64 cute::xml_file_opener xmlfile(argc, argv); 65 cute::xml_listener<cute::ide_listener<>> listener(os::xmlfile.out); 66 auto runner = cute::makeRunner(s; listener, argc, argv); 67 68 bool suiteResult = runSuite(runner); 69 bool runSuite2_result = runSuite2(runner); 70 return suiteResult && runSuite2_result ? EXIT_SUCCESS : EXIT_FAILURE; 71 }</pre> <p>Figure 68: Unregistered Suite Quick Fix</p>

Test-10: single legacy test quick fix

Description:	This test should ensure that the quick fix to update a single test, that was defined using the old CUTE macro, works as intended. This quick fix should replace the old syntax for the specific test by the new TEST macro syntax.
Preconditions:	<ul style="list-style-type: none">• Clangd extension installed and activated• CUTE extension installed and activated• cuteExtensionTestProj folder opened as workspace• CMake project is built
Input / Interaction:	<ol style="list-style-type: none">1. Open test.cpp file in the editor2. Save the document to immediately trigger the code analysis for the file3. Hover the marked LegacyTestCase() command on line 38 and select quick fixes -> update syntax4. Save the document again
Expected Output:	<p>The legacy test declaration is detected and marked correctly as shown in the image below. The quick fix to update the legacy syntax to the new TEST(...) syntax is available. After the quick fix is applied and the document was saved, the test is no longer marked. The test definition was wrapped into the TEST(...) macro and the CUTE(...) macro was removed from the call command.</p> <pre>38 void LegacyTestCase() 39 { 40 ASSERT_EQUAL(10, 10); 41 } 42 43 bool runSuite(cute::runner<cute::xml_listener<cute::ide_listener<>>> runner) 44 { 45 cute::suite s{}; 46 s += testEnterHouse(); 47 s += testHouseRooms(); 48 s += testFail(); 49 50 s += CUTE(LegacyTestCase); 51 52 return runner(s, "TestSuite"); 53 }</pre> <p>Figure 69: Legacy Syntax Warning</p> <pre>38 TEST(LegacyTestCase) 39 { 40 ASSERT_EQUAL(10, 10); 41 } 42 43 bool runSuite(cute::runner<cute::xml_listener<cute::ide_listener<>>> runner) 44 { 45 cute::suite s{}; 46 s += testEnterHouse(); 47 s += testHouseRooms(); 48 s += testFail(); 49 50 s += LegacyTestCase(); 51 52 return runner(s, "TestSuite"); 53 }</pre> <p>Figure 70: Legacy Syntax Quick Fix</p>

Test-11: multiple legacy test quick fix

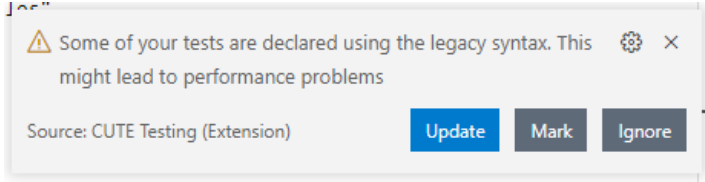
Description:	This test should ensure that the quick fix to update all detected test, that were defined using the old CUTE macro, works. This quick fix should replace the old syntax for all legacy test declarations by the new TEST macro syntax.
Preconditions:	<ul style="list-style-type: none">• Clangd extension installed and activated• CUTE extension installed and activated• cuteExtensionTestProj folder opened as workspace• CMake project is built
Input / Interaction:	<ol style="list-style-type: none">1. Open test.cpp file in the editor2. Save the document to immediately trigger the code analysis for the file3. Select the Update button on the warning that pops up in the lower right hand corner4. Wait until the update process is completed
Expected Output:	<p>The legacy test declaration is detected and marked correctly as shown in the image in the expected output of Test-11. The quick fix to update all discovered legacy syntax usages to the new TEST(...) syntax is available via the warning pop-up in the lower right hand corner. After the quick fix is applied, the test is no longer marked. The test definition was wrapped into the TEST(...) macro and the CUTE(...) macro was removed from the call command. This multi quick fix saves the documents that were updated automatically.</p> 

Figure 71: Legacy Syntax Warning Pop-Up

Test-12: generate project test

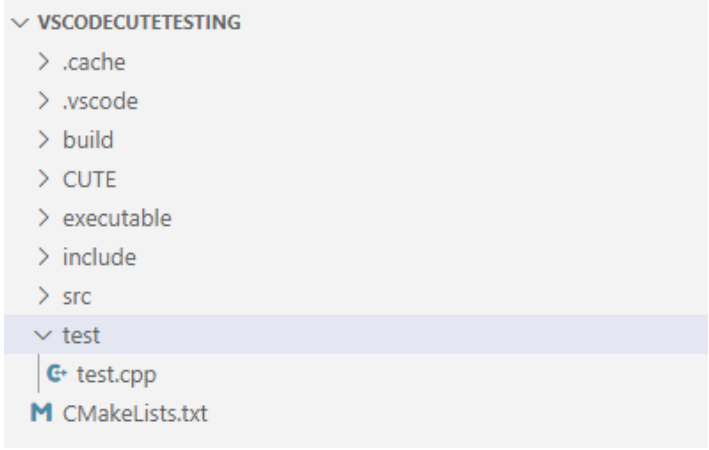
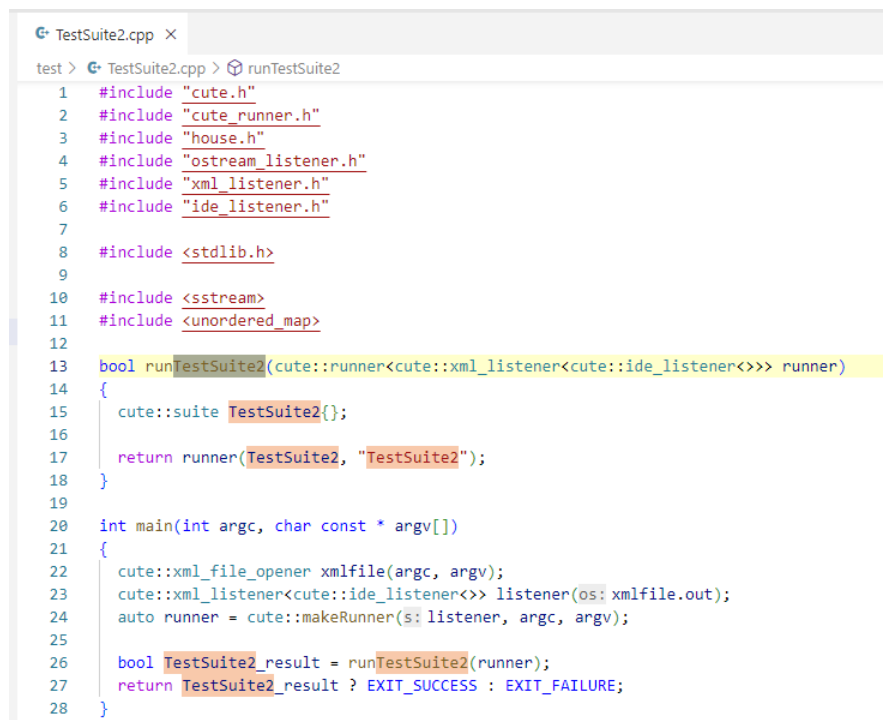
Description:	This test should ensure that the project generation command works. This functionality allows it to set up new C++ projects including a CUTE testing project. This project generation is based on a template project.
Preconditions:	<ul style="list-style-type: none">• CUTE extension installed and activated• An empty folder opened as workspace
Input / Interaction:	<ol style="list-style-type: none">1. Open commands execution terminal (CTRL+Shift+P)2. Type CUTE: Generate project and run the command3. Wait until the project setup is finished
Expected Output:	<p>The command to set up a new CMake based project is available and can be called using CTRL+Shift+P. If Visual Studio Code has a single and empty workspace opened the project template gets set up in that empty workspace. The structure of the template project can be found in the figure below.</p>  <pre>VSCodeCUTETESTING ├── .cache ├── .vscode ├── build ├── CUTE ├── executable ├── include ├── src ├── test │ ├── test.cpp │ └── CMakeLists.txt</pre>

Figure 72: CUTE Project Template

Test-13: generate suite file test

Description:	This test should ensure that the suite file generation command works. This functionality allows it to add a new test file to the project. The new test file contains a test suite definition with all required boilerplate code.
Preconditions:	<ul style="list-style-type: none">• CUTE extension installed and activated• cuteExtensionTestProj folder opened as workspace
Input / Interaction:	<ol style="list-style-type: none">1. Open commands execution terminal (CTRL+Shift+P)2. Type CUTE: Create testsuite file and run the command3. Enter the suite name into the field that opens up4. Confirm name and run command5. Wait until the code generation process finished
Expected Output:	The command to set up an additional test suite file is available and can be called using the CTRL+Shift+P command execution shortcut. The command requires some user input for the suite name. After the user has set a name, the file is set up using the boilerplate template code for new suites. The CMake configuration file is updated accordingly.



```
TestSuite2.cpp ×
test > TestSuite2.cpp > runTestSuite2
1  #include "cute.h"
2  #include "cute_runner.h"
3  #include "house.h"
4  #include "ostream_listener.h"
5  #include "xml_listener.h"
6  #include "ide_listener.h"
7
8  #include <stdlib.h>
9
10 #include <sstream>
11 #include <unordered_map>
12
13 bool runTestSuite2(cute::runner<cute::xml_listener<cute::ide_listener<>>> runner)
14 {
15     cute::suite TestSuite2{};
16
17     return runner(TestSuite2, "TestSuite2");
18 }
19
20 int main(int argc, char const * argv[])
21 {
22     cute::xml_file_opener xmlfile(argc, argv);
23     cute::xml_listener<cute::ide_listener<>> listener(os: xmlfile.out);
24     auto runner = cute::makeRunner(s: listener, argc, argv);
25
26     bool TestSuite2_result = runTestSuite2(runner);
27     return TestSuite2_result ? EXIT_SUCCESS : EXIT_FAILURE;
28 }
```

Figure 73: Create New Suite Command

Test-14: CUTE TEST snippet test

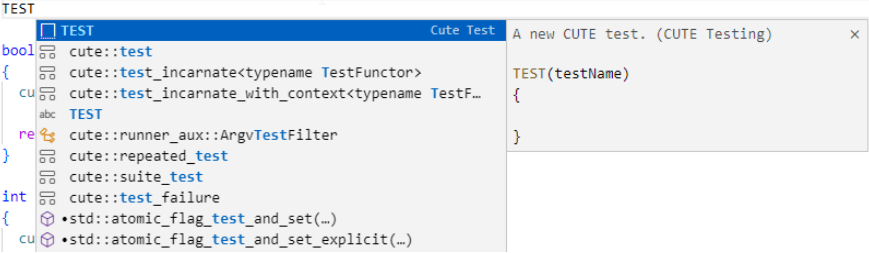
Description:	This test should ensure that the TEST snippet, that can be used to easily create a new CUTE test, is available and works as intended. The TEST snippet should be available in all cpp files, and the name templating should be activated, when selecting the snippet.
Preconditions:	<ul style="list-style-type: none">• CUTE extension installed and activated• cuteExtensionTestProj folder opened as workspace
Input / Interaction:	<ol style="list-style-type: none">1. Open test.cpp file in the editor2. Navigate to a location within the file outside of any functions3. Start typing TEST and press enter as the TEST snippet shows up in the suggestions
Expected Output:	<p>The TEST snippet is available in a cpp file and provides support with creating new CUTE tests. The snippet offers to set a test name after its expansion was chosen. The snippet shows up in the VS Code recommendations after starting to type TEST in a cpp file.</p> 

Figure 74: TEST Snippet

Test-15: assert failure comparison test

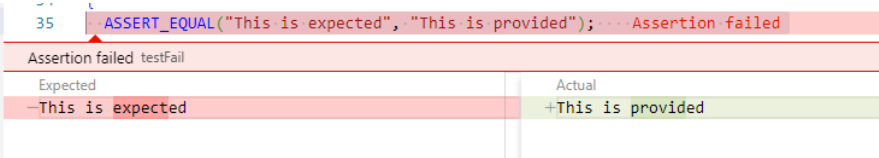
Description:	This test should ensure that the assert failure functionality works as intended. If a CUTE test should fail due to an assertion failure, a window should show up that allows an easy comparison between the expected and the actual outcome of that test.				
Preconditions:	<ul style="list-style-type: none">• CUTE extension installed and activated• cuteExtensionTestProj folder opened as workspace• CMake project is built• Tests are discovered and loaded into the test explorer				
Input / Interaction:	<ol style="list-style-type: none">1. Select the test Failtest in the test explorer and click the run test button2. Wait until the test run finished with errors.				
Expected Output:	<p>If a test failed because of an assertion failure, a window shows up inside the code editor underneath the test definition. Inside this window the value expected by the test and the actual value provided by the tested functionality are shown. To make the reason for the assertion failure clearer, a colorized visualization of the differences available. The image below shows the assertion failure visualization of the FailTest test.</p>  <pre>35 ASSERT_EQUAL("This is expected", "This is provided");... Assertion failed</pre> <table><thead><tr><th>Expected</th><th>Actual</th></tr></thead><tbody><tr><td>-This is expected</td><td>+This is provided</td></tr></tbody></table>	Expected	Actual	-This is expected	+This is provided
Expected	Actual				
-This is expected	+This is provided				

Figure 75: Assertion Failure Visualization

8.1.1 Test Protocol

This section contains the test protocol for the manual tests regarding the functional requirements. The test protocol for the CUTE extension covers all outcomes of the final test run, that was carried out before releasing the CUTE extension in the VS Code marketplace.

Test	Status	Comment	Date
Test-1	Passed	The tests get discovered as expected. All four tests are available through the test explorer and directly from within the code. The unregistered tests are not marked as test cases and therefore cannot be started from anywhere.	2022-06-12
Test-2	Passed	The tests get discovered as expected. The reliability of this functionality heavily depends on the systems performance. Unregistered tests are not marked as tests and therefore they are not shown in the test explorer.	2022-06-12
Test-3	Passed	The tests get discovered as expected. All four tests are available through the test explorer. Only the tests that were defined using the TEST(...) macro are marked within the code and can be started from there.	2022-06-12
Test-4	Passed	The debugger is started as expected. The execution pauses at the breakpoint and the execution time of a test is shown correctly within the test explorer. The result outcome is visualized in the test explorer and within the code.	2022-06-12
Test-5	Passed	The test is executed as expected. The test run is started with the single selected test. After the test run is finished, the test result is shown in the test explorer and is visualized within the code. The run duration is shown in the test explorer.	2022-06-12
Test-6	Passed	The tests are started as expected. The test run contains all tests that belong to the TestSuite test suite. As soon as a test has finished, its outcome is shown in the test explorer. The result is also visualized within the code.	2022-06-12
Test-7	Passed	The unregistered test is marked by a yellow line and within the problems view. The quick fix to register the test case to an existing test suite is available and works as expected. After the quick fix was applied and the document was saved, the warning disappears.	2022-06-12

Test-8	Passed	The unregistered test is marked by a yellow line and within the problems view. The quick fix to register the test case to a new test suite is available and works as expected. The name of the new test suite can be set by the user.	2022-06-12
Test-9	Passed	The unregistered RunSuite(...) function is marked by a yellow line and within the problems view. The quick fix to register the test suite to a test runner in the main function is available and works as expected.	2022-06-12
Test-10	Passed	The test that is defined using the CUTE(...) legacy syntax is marked by a yellow line and within the problems view. The quick fix to update the syntax and use the TEST(...) macro is available and works as expected.	2022-06-12
Test-11	Passed	The warning pop-up is shown and offers the option to update all legacy test declarations. After the update is finished, all updated files are saved, and the warnings are no longer shown within the code and also no longer appear in the problems view.	2022-06-12
Test-12	Passed	The project is set up as expected. It contains a library, an executable, and a test executable. The CMake configuration is set up according to the projects structure and offers an easy way to compile the solution.	2022-06-12
Test-13	Passed	A new cpp file is created and added to the CMake configuration. The test file contains the required boiler plate code including a RunSuite(...) function. The suite name can be set by the user.	2022-06-12
Test-14	Passed	The TEST(...) snippet is available from within cpp files. After starting to write TEST, the snippet appears in the recommendations view. When the snippet is selected, a test function gets set up.	2022-06-12
Test-15	Passed	The assertion failure is marked as such in the test result. The difference between the expected value and the provided value is visualized in the test result which can be found next to the test's implementation location.	2022-06-12

8.2 Nonfunctional Requirements (NFR) Testing

In this section the tests regarding the nonfunctional requirements (NFR) can be found. The project's code and design were analyzed for each of the defined nonfunctional requirements. The definition of the nonfunctional requirements (NFR) can be found in the chapter Requirements Analysis.

8.2.1 Test Protocol

This section contains the protocol for the analysis of the code and design regarding the non-functional requirements. The protocol covers all analysis results of the final CUTE extension version. The analysis was carried out on the version of the CUTE extension that was later released to the VS Code marketplace.

NFR	Status	Comment	Date
NFR-1	Passed	A set of useful user notifications is built into the CUTE extension. If something should go wrong the user gets notified about that malfunctioning in an understandable way.	2022-11-06
NFR-2	Passed	The CUTE extension can handle misconfigurations such as using code-based test discovery when no language server provider is available and notifies the users about the problem. Missing debug extensions do not lead to the extension crashing.	2022-11-06
NFR-3	Passed	The test discovery using the executable based approach is pretty much instant and it takes less than 10 seconds until the discovered tests are registered in the test explorer. The code-based approaches' performance depends heavily on the used language server provider. Clangd performs well, and test discovery does not take much longer than the executable based approach. The test execution and result visualization logic does not create a big performance impact.	2022-11-06
NFR-4	Passed	The CUTE extensions user interface is based on the official VS Code test explorer. The controls are familiar to many developers and therefore do not require any training. The CUTE extension configuration possibilities are described understandably.	2022-11-06
NFR-5	Passed	A CI/CD pipeline [13] is set up and makes sure that no breaking changes or code style violations are merged into the master branch or published to the VS Code marketplace. The test coverage from unit-, integration- and system tests is over 80% when the tests are executed in all possible configurations (Clangd/CppTools).	2022-11-06
NFR-6	Passed	Sonarqube [14] is set up and integrated into the CI/CD pipeline on GitLab [13]. The code metrics show that the code is well maintainable. According to the analysis, there are no duplicated lines within the project. The maintainability and reliability are rated A and the targeted 80% of test coverage are achieved. Additional manual system tests extend the quality measures and cover all functionalities. This is a further prevention of breaking changes that get into productivity.	2022-11-06

9 Conclusion

This chapter contains the conclusion of the CUTE extension for VS Code project. To provide an overview of the whole course of the project, a result evaluation can be found in this chapter. Further, a comparison of the reached goals and the open work is provided. At the end of this chapter a foresight will be provided, in which the potential next steps are described and explained.

9.1 Summary

In the scope of this thesis a project was started with the aim to develop an integration of the CUTE testing framework into Visual Studio Code. The project was split into four phases according to the RUP [73] definition. After the project was successfully started and all required tools were set up during the inception phase, the planning of an extension design was started in the elaboration phase. During this evaluation phase the key requirements were analyzed and different approaches to implement the functionality behind them, were evaluated. After the elaboration phase was finished, it was clear which features can be supported and what their implementation should look like. The biggest risks could be mitigated during this elaboration phase, as planned during the project planning. During the development phase the actual CUTE extension for VS Code was designed and implemented based on the findings from the elaboration phase. After the development phase was finished, a Visual Studio Code extension was available, that offers a feature rich and easy to use integration of the CUTE testing framework [10]. During the final transition phase of the project, the CUTE extension for VS Code was published to the marketplace from where it could be installed afterwards. Further all required parts were documented, and the final presentation was prepared.

9.2 Result evaluation

The CUTE extension for Visual Studio Code that was created in the scope of this thesis, offers a rich set of functionalities. The CUTE integration into VS Code offers all mandatory features such as the starting of test runs covering one or multiple tests, and the visualization of the test results using a red/green bar experience. In addition to these minimal requirements, a large set of addition features is offered by the CUTE extension for VS Code. On the one hand the CUTE extension provides support with setting up new CMake based test projects, or the configuration of new test suites in separate cpp files. The comfort of a TEST snippet, that allows the creation of new tests without having to write the whole boiler plate code, is provided as well. Beside these test writing support functionalities, the CUTE extension analyzes the test code and warns the developers about potential code problems. These code problems include unregistered test cases, unregistered test suites and test declarations using the legacy CUTE(...) syntax. The extension does not just simply discover the problems, but also offers powerful quick fixes to mitigate all of them. The extension offers some configuration possibilities, for example to set the test discovery mode. The CUTE extension supports multiple language server provider implementations (Clangd [19] / C/C++ for VS Code by Microsoft [20]) and is tested under Windows and Linux. In comparison to the Cevelop plug-in for CUTE [5] a larger set of convenience tools is provided and an improved user experience is available as the CUTE extension builds on familiar user interface controls. All base functionalities offered by the Cevelop plug-in can be covered using the VS Code extension, where they can be used in an even user friendlier way.

9.3 Reached goals / open work

During this project a VS Code extension could be planned and developed, that offers a rich set of functionalities that in some points even exceed the ones offered by the Cevalop plug-in [5]. The planned tasks could be finished within their planned time frame. The only open user story is about a separate tool to convert existing Cevalop based solutions to CMake [9] based projects that can be opened and used in Visual Studio Code.

Beside the functional requirements which all could be implemented and for which an adequate test coverage of over 80% exists, the non-functional requirements could be fulfilled. The test protocol for the manual system tests and the evaluation of the non-functional requirements, can be found in the chapter Testing of this documentation.

9.4 Future view

The next step is to use the CUTE extension for Visual Studio Code in the scope of a C++ module at OST Eastern Switzerland University of Applied Sciences and get user feedback directly from students. Based on this feedback the next steps can be defined and areas identified, that might need changing. In addition to the validation of the CUTE extension in a practical environment, it should be considered to implement the earlier mentioned converter that should provide the functionality to convert Cevalop projects to CMake based projects, that can be used in Visual Studio Code in combination with the CUTE extension.

10 Project Management

In this chapter of the documentation an overview about the project management of the project CUTE extension can be found. This overview should describe the management and planning tasks of the project. Thereby the main focus is set the project planning, the organization and further topics of the project structuring. The planning, which is described in this chapter, builds the baseline for the project CUTE extension.

The scope of this chapter is limited to the project duration of the project CUTE extension, which is being realize in the context of a bachelor thesis in FS2022 at the OST University of Applied Sciences. This chapter is updated throughout the whole project.

10.1 Organization

10.1.1 Project Contributors

- Dominic Klinger, dominc.klinger@ost.ch
- Christian Bisig, christian.bisig@ost.ch

10.1.2 Initiator / Supervisor

- Thomas Corbat, thomas.corbat@ost.ch

10.1.3 Expert / Examiner

- Guido Zraggen - Examiner, zraggen@gmail.com
- Frieder Loch - Co-Examiner, frieder.loch@ost.ch

10.2 Work Breakdown Structure

The graphic below shows the initial work breakdown structure of the CUTE extension project. At the beginning of the project, during the elaboration phase, the project description was analyzed and split into distinct components. In a later step, these components were assigned to a specific project phase according to the Scrum+ (Combination of Scrum [72] and RUP [73]) phases. These components built the baseline for the epics which were defined afterwards. As shown in the graphic below, each of the components belongs to one out of the three distinct categories Coordination, Infrastructure or Application.

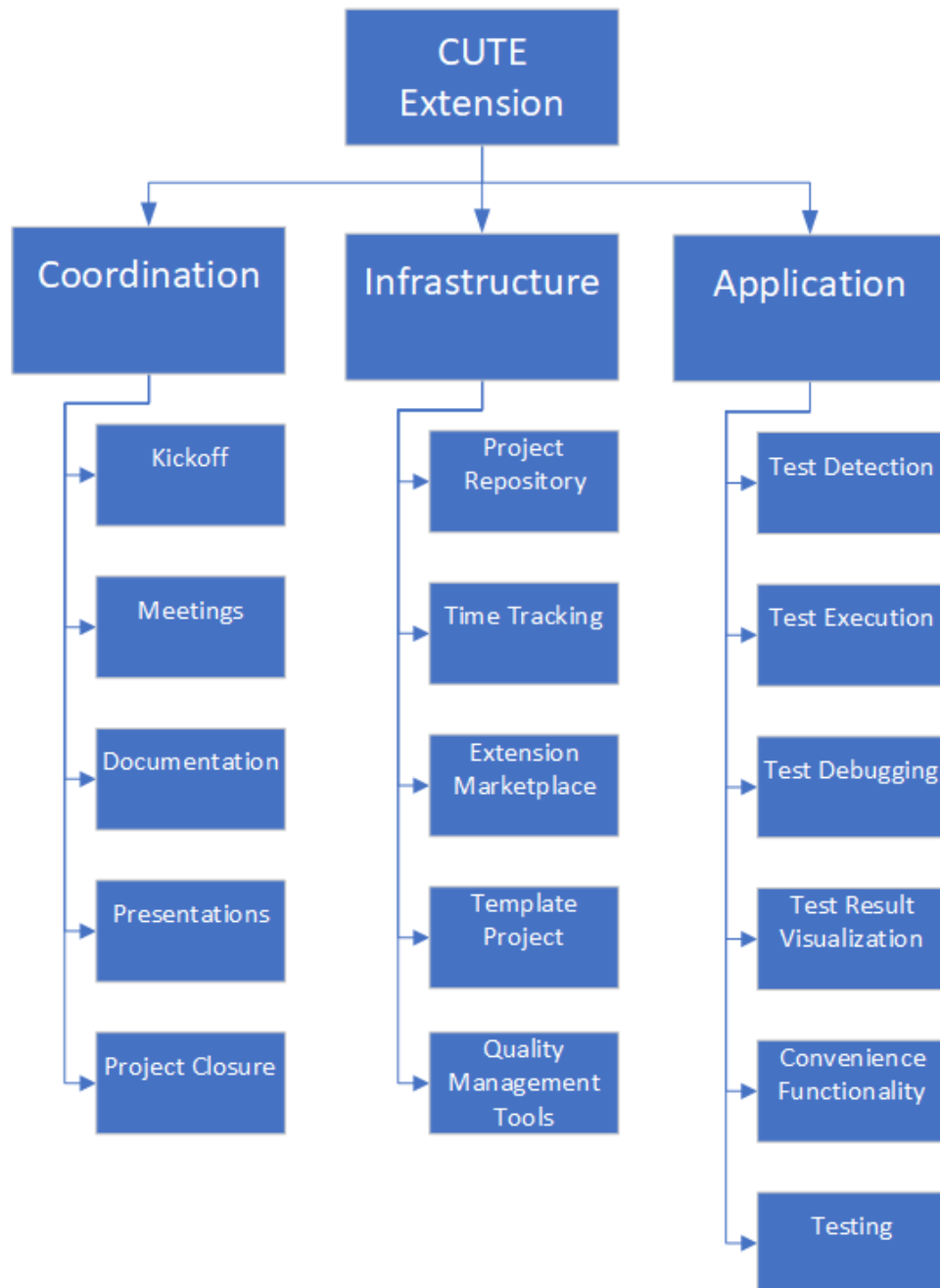


Figure 76: Work Breakdown Structure Diagram

10.3 Cost Estimate

This section contains an overview of the estimated costs in the form of time spent. In addition to the effort estimation the start and end dates of the project can be found in this section.

Project Duration	15 Weeks
Number of Contributors	2 People
Weekly Workload	24h
Total Workload	720h
Project Start	21.02.2022
Project Deadline	17.06.2022

10.4 Time Planning

The time planning for this project is loosely based on the four phases defined in the Scrum+ process. These four phases are Inception, Elaboration, Construction and Transition. In this section of the chapter Project Management those four phases will be described and scheduled throughout the project duration.

10.4.1 Project Phases

The earlier mentioned phases define important deadlines throughout the project duration. Each of these deadlines therefore states an important milestone. The duration of each phase was defined based on the guidelines of Scrum+, which combines the RUP [73] phases with the agility of Scrum [72] (Sprints within each phase). Using this technique, the biggest part of the time available is assigned to the two phases Elaboration, in which the feasibility of the features is analyzed, and different approaches are evaluated, and Construction, during which actual development of the product takes place. In the table below the phases are listed and a start and end date is assigned to each of them. In a later step further milestones were defined and assigned to the corresponding phase. The milestone definition can be found in the next section of this chapter.

Phase	Start Date	End Date
Inception	21.02.2022	27.02.2022
Elaboration	28.02.2022	03.04.2022
Construction	04.04.2022	05.06.2022
Transition	06.06.2022	17.06.2022

10.4.2 Milestones

This section contains the defined milestones of the CUTE extension for VS Code project. Each milestone has a deadline and a general description. All of the below defined milestones can be found in the timeline graphic that is following in the section below.

Definition Declaration	M1 Project Plan <ul style="list-style-type: none">• Project plan template is set up• Basic planning on level Epic is finished• Project and develop environment is set up• Time tracking is ready to use
Date	06.03.2022
Definition Declaration	M2 Requirement Analysis <ul style="list-style-type: none">• Requirements analyzed• Important use cases documented• User stories created based on earlier defined use cases• Risk analysis initialized
Date	20.03.2022
Definition Declaration	M3 End of Elaboration <ul style="list-style-type: none">• Prototype for all use cases evaluated• Bigger risks mitigated• Plan for construction phase set up based on findings made during the elaboration phase
Date	03.04.2022
Definition Declaration	M4 Intermediate Presentation <ul style="list-style-type: none">• Presentation of the current project state in front of the supervisor and expert committee• Demonstration of the prototype that was created during the elaboration phase• Potential additional features integrated into the project planning• Let feedback flow into the planing of the construction phase
Date	05.04.2022

Definition Declaration	M5 Alpha <ul style="list-style-type: none"> • First version of the final extension is available • Tests can be discovered based on information received from the test executables • Tests can be executed and debugged • There is a template project available • There is a possibility to easily create additional test suites
Date	01.05.2022
Definition Declaration	M6 Beta <ul style="list-style-type: none"> • New functioning version of the extension is available • Includes additional convenience features such as warnings on unregistered tests or suites • Includes functionality to discover tests and suites based on language server information • Includes potentially added features from the intermediate presentation • The beta version of the extension is tested and fulfills the quality requirements
Date	29.05.2022
Definition Declaration	M7 Project Delivery <ul style="list-style-type: none"> • Final version of the extension is available • Includes all features which were evaluated during the elaboration phase • Bugfixes based on the beta version were implemented • Documentation about the project and the product is finished and covers all important points • The final version of the extension is tested and fulfills the quality requirements
Date	17.06.2022
Definition Declaration	M8 Final Presentation <ul style="list-style-type: none"> • Presentation of the final product in front of the supervisor and expert committee • Potential questions regarding the project and final product addressed
Date	21.06.2022

10.4.3 Timeline

The graphic below shows the estimated timeline at the beginning of the project. Thereby this timeline provides an overview of the four phases, the project milestones and the iterations which stretch over two weeks. For each iteration the most important epics are listed within the corresponding timeframe. This timeline shows which epic has to be finished by when in order to be able to successfully finish the whole CUTE extension project in the time available.

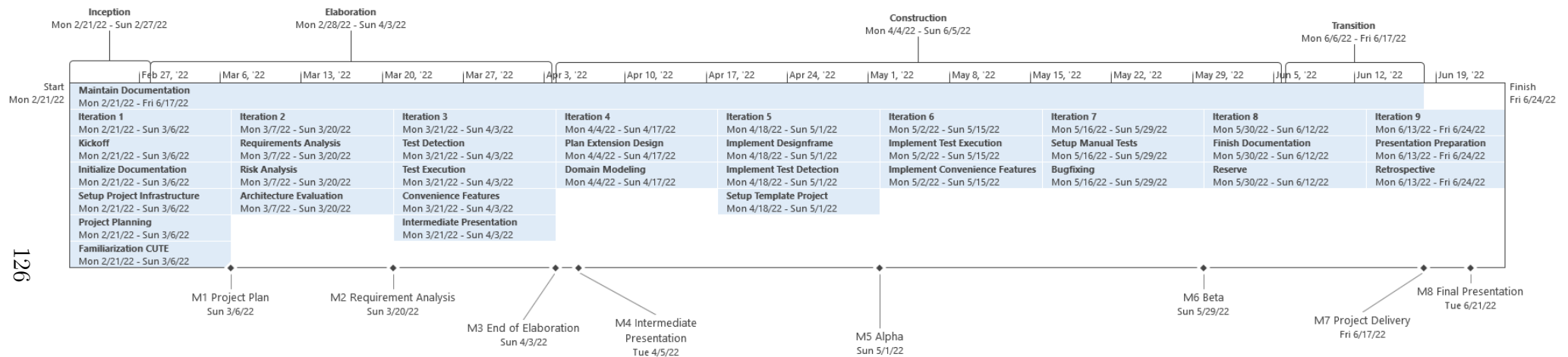


Figure 77: Project Timeline

10.4.4 Epic Estimate

The table below contains the first estimation for the most important epics of the CUTE extension Project. The epics below will be split into smaller tasks during the corresponding iteration. These tasks will then build the basis for time tracking and progress tracking. The actual spent effort for each epic can be found in the appendix in the section time tracking. This appendix section will further contain a more detailed listing of the spent time for each work item.

Inception	48 hrs
Kickoff	8 hrs
Initialize Project Documentation	20 hrs
Setup Project Infrastructure	20 hrs
Elaboration	252 hrs
Project Planning	24 hrs
Familiarization CUTE	20 hrs
Requirements Analysis	24 hrs
Risk Analysis	20 hrs
Architecture Evaluation	48 hrs
Test Detection	36 hrs
Test Execution	24 hrs
Convenience Features	24 hrs
Intermediate Presentation	8 hrs
Maintain Documentation	24 hrs
Construction	354 hrs
Plan Extension Design	42 hrs
Domain Modeling	12 hrs
Implement Designframe	24 hrs
Implement Test Detection	36 hrs
Setup Template Project	24 hrs
Implement Test Execution	36 hrs
Implement Convenience Features	48 hrs
Setup Manual Tests	24 hrs
Bugfixing	48 hrs
Maintain Documentation	24 hrs
Reserve	36 hrs
Transition	72 hrs
Finish Documentation	48 hrs
Presentation Presentation	16 hrs
Retrospective	8 hrs
Total	726 hrs

10.4.5 Workitems

The management of the work items such as tasks and user stories is handled over GitLab [74]. For each part of the project such work items can be created and assigned to the earlier mentioned milestones. Time is tracked through the project and always assigned to a task. The further in the future the implementation of a work item is, the more generic it is. The work items are refined over time and finally split into small enough tickets. There are also work items for the project management overhead like meetings, reviews, and documentation of the project.

10.5 Risk analysis

This section contains an analysis of the major risks, that could endanger the course of the project and potentially event the project's success. The risks were analyzed regarding their probability of occurrence and their expected maximal damage to the project. Based on this analysis actions were defined to mitigate the threats.

10.5.1 R1: CUTE does not fulfil requirements for VS Code test environment

Description:	There is no possibility to extend or adapt the CUTE framework in a way to make it fit the needs of the Visual Studio Code extension. (e.g. return the implementation location of test cases)
Maximum damage:	120 hours
Probability of occurrence:	25 %
Weighted damage:	30
Prevention:	Try different approaches at the beginning of the project during the elaboration phase. Also plan a fallback solution that is not based on information returned by the test executable itself.
Behave when entering:	Reduce the scope of the project through the cancellation of optional features such as the convenience tools. Use the already planned fallback solution to retrieve the needed information.

10.5.2 R2: VS Code Testing API does not fulfil the requirements

Description:	Visual Studio Code does not provide functionality that is needed to implement all required features. (e.g. allow access to language feature information)
Maximum damage:	60 hours
Probability of occurrence:	30 %
Weighted damage:	18
Prevention:	Create prototypes using different approaches early in the project during the elaboration phase. Find out what is possible and what isn't. Plan the features that will be implemented based on the findings from the elaboration phase.
Behave when entering:	Cancel the optional features e.g. convenience tools that depend on Visual Studio Code features that are not supported. Reduce the scope of the project. Find solutions that are not based on unsupported Visual Studio Code features for mandatory CUTE extension features. Check options with project supervisor.

10.5.3 R3: Incorrect handling of the requirements

Description:	Important requirements are getting forgotten or are wrong estimated in time effort
Maximum damage:	60 hours
Probability of occurrence:	15 %
Weighted damage:	9
Prevention:	The first task during the project should be the exact analysis of the requirements. Uncertainties should be clarified during the elaboration phase in a timely manner with the project supervisor. Time and effort estimations should be made in the project team using the four-eye principle.
Behave when entering:	Adapt the scope of the project after discussing the situation with the project supervisor.

10.5.4 R4: Wrong architectural decisions

Description:	There are different options available to implement a Visual Studio Code testing extension e.g. implement everything from ground up or take an existing extension as baseline for the project.
Maximum damage:	100 hours
Probability of occurrence:	20 %
Weighted damage:	20
Prevention:	Existing testing extensions for Visual Studio Code should be analyzed during the elaboration phase of the project. The different possibilities should be evaluated early in the project and based on the evaluation result a decision should be taken after a supervisor consultation.
Behave when entering:	A change of the project architecture should not lead to the need of reimplementing the entire functionality. Reuse already existing functionality in the new architecture. Reduce the scope to make up the lost time.

10.5.5 Risk Matrix

The graphic below shows the risk matrix in which the above analyzed risks can be found. The graphic clearly shows that risk R1 could become the biggest threat to the project success. Therefore, the decision was made to try different approaches to extract the required information from the CUTE testing framework during the elaboration phase. Further the decision was made to evaluate the possibility of creating a fallback solution that does not depend on the information from the executables themselves. Besides risk R1 the risks R2 and R4 should also be observed throughout the project. Both of these risks state an increased threat to negatively impact the course of the project. The extended prototyping throughout the elaboration phase should help to minimize these risks as well as the already mentioned risk R1. The extension prototype should show what is possible and what isn't and with this information impact the plan and feature set of the final CUTE extension product. These three risks, that state increased or sever threats to the project should be taken care of as early as possible in the project. Risk R3 is in the green are of the risk matrix. Therefore, the earlier described precautionary measures should be sufficient. The risks will be reanalyzed throughout the projects duration and the measures readjusted if necessary. The objective of the elaboration phase is to mitigate as much risk as possible to allow a smooth implementation of the final product.

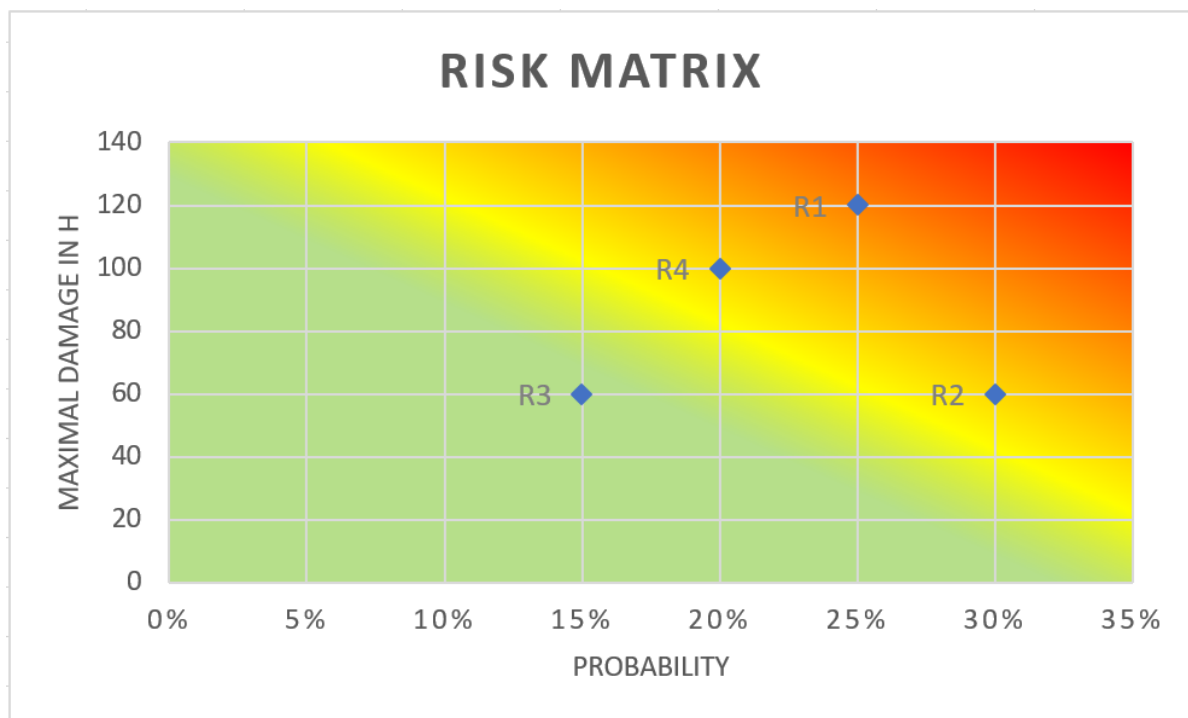


Figure 78: Risk Matrix

10.6 Quality Management

To ensure the quality of the Visual Studio Code CUTE extension the following measures have been defined. The table below contains an overview of the implemented quality measures with their corresponding scope and objective.

Measure	Scope	Objective
System Tests (Checklist)	Whole Project Duration	Ensure the functionality of the extension features. Make sure that all use cases can be covered by the Visual Studio Code CUTE extension.
Unit Tests	During Implementation	Guarantee that all components work as intended. This measure should help to improve the robustness of the code.
Integration Tests	During Implementation	This measure should ensure that the single components are able to interact with each other without any malfunctions.
Code Reviews	During Implementation (Merge Request)	Code reviews lead to an important knowhow exchange. Further they reduce the risk that the main branch gets broken through the implementation of the four-eyes principle.
Code Metrics	Whole Project Duration	From time to time code metrics should be evaluated. This measure should help to find code smells, security problems and duplicated code. These problems then will be mitigated through code refactoring.

10.6.1 Tools

To implement the above defined quality measures the following tools will be used throughout the project:

- Mocha [75] as testing framework for the implementation of unit and integration tests
- ts-mockito [76] as library for mocking typescript classes and interfaces
- Sonarqube [14] to evaluate the code metrics

10.7 Development Tools

For the development itself several tools like development containers and continuous integration pipelines were used. In this section a description of the used tools and workflows can be found.

10.7.1 Documentation

The documentation is written in LaTeX [78] and needs some packages for the build. To make writing the documentation as easy as possible, Development Docker Containers [79] are used in Visual Studio Code. These allow it to configure a Docker container so that the working environment is always the same and all project members can work with a similar setup. There is also the possibility to configure VSCode extensions to make the documentation writing easier and to find typos without any effort using a spell checker plugin [80]. Each push to the GitLab repository also triggers a pipeline that takes the same Development Container as a template. With each push, a PDF is offered for download so that the current version of the documentation is always available.

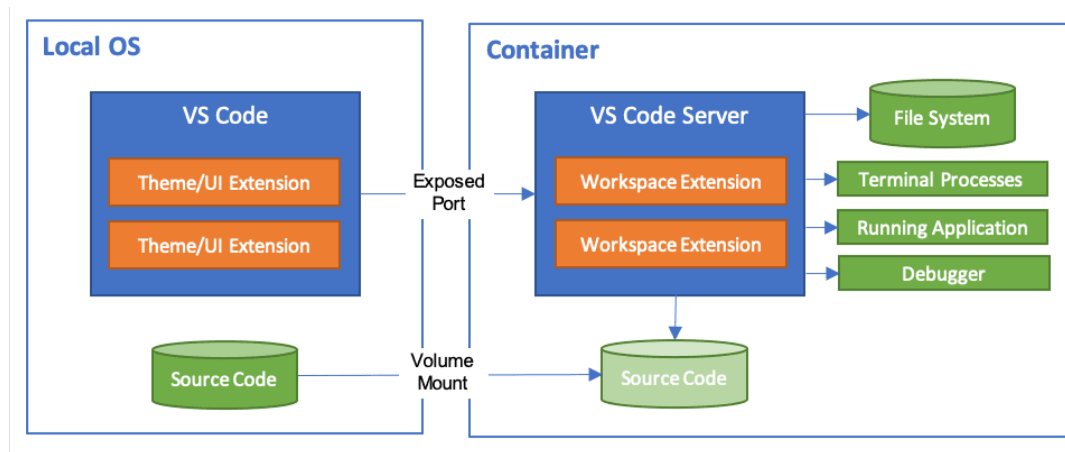


Figure 79: Development Containers Architecture [77]

10.7.2 Extension

This section contains a description of the CI/CD pipeline [13] that was set up on GitLab. With every push to the GitLab repository the pipeline gets triggered.

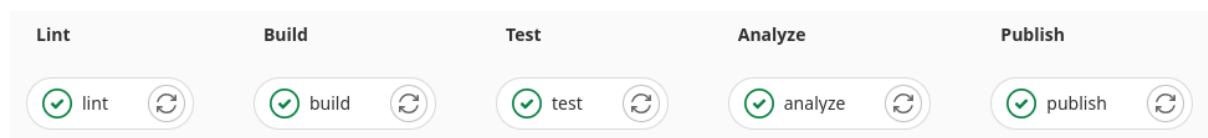


Figure 80: CUTE Extension Pipeline

The pipeline consists of the following stages: lint, build, test, analyze and publish. Linting is about formatting the code and following the style guides. The decision was made to use the AirBnB style guide, specialized for Typescript. The AirBnB code style is widely used and has also been used in some of our past projects, which is why we again relied on it. The build stage builds the project and provides the build output as artifact. If something goes wrong here, users get notified quite quickly about the fact that the pipeline is broken as the build failed. If e.g. dependencies are missing or other problems occur, this step fails and the developer who triggered the pipeline gets a notification. If this build stage passes, the system should be in a "clean" state and therefore have no influence on the development system. In the test stage all tests are executed. The project contains a list of unit, integration and system tests, which must finish successfully. With the execution of the tests, a test report for GitLab and a coverage report is generated, which is afterwards used in the next analysis step.

Tests







Suite	Name	Filename	Status	Duration	Details
Check Unregistered Test Code Warning Test	Code Diagnostics Test Suite Check Unregistered Test Code Warning Test			30.07s	View details
Check Unregistered Suite Code Warning Test	Code Diagnostics Test Suite Check Unregistered Suite Code Warning Test			30.03s	View details
Check Legacy Test Code Warning Test	Code Diagnostics Test Suite Check Legacy Test Code Warning Test			30.03s	View details
GetTestRunners test	ClangdCodeAnalyzer Test Suite GetTestRunners test			11.69s	View details
GetTestSuites test	ClangdCodeAnalyzer Test Suite GetTestSuites test			11.00s	View details
GetLegacyTestCases test	ClangdCodeAnalyzer Test Suite GetLegacyTestCases test			10.18s	View details

Figure 81: GitLab Test Report

The Analyze stage triggers the code analysis process with SonarQube. SonarQube afterwards provides code coverage information and shows code smells and potential bugs. It also shows problems with bad programming style, for example, if there are code duplications. The analysis step is always executed, no matter if one of the previous stages failed. In the last stage, the publish stage, the extension gets packaged. Here a VSIX-file is generated as result. With the VSIX file, the extension can then be installed in Visual Studio Code. If the pipeline runs after a corresponding tag was set on the source branch, the automatic publication to the Marketplace [8] is also triggered, so that the extension is made available in the published version in the Marketplace.

10.7.3 Development Server

For the this work a virtual machine was used. On the machine there is SonarQube, the GitLab Runner, and a LoadBalancer for some smaller information websites installed.

Error during latex code generation

Figure 82: Development Server structure

10.7.4 Code Style Guidelines

For the typescript implementation of the Visual Studio Code CUTE extension the decision was made to follow the extensive Airbnb javascript style guides [70] with some exceptions. There are some smaller exceptions made by the prettier plugin [81]. these exceptions concerns the bracketing, spacing and some smaller formatting in function headers.

11 Glossary

Assertion	Used for comparisons of expected and actual values. Used in software testing
Cevelop [3]	C++ IDE based on Eclipse CDT[6]
CDT [6]	C/C++ Development Tooling based on the Eclipse platform
CD	Continuous Deployment, pipeline that automatically publishes the extension on the marketplace
CI	Continuous Integration, pipeline that runs the automated tests and static code analysis always when a branch is pushed to the repository
CMake[9]	Is a family of tools to build, test and package software
Clangd [19]	Offers language support for C++ code. Can be installed as VS Code extension
Clang [16]	Provides language tooling e.g. compiler infrastructure for the C language family, to which C++ is assigned
CodeLLDB [59]	Is a Visual Studio Code extension that offers debugging capabilities based on the LLDB debugger
Commands [61]	VS Code offers an interface against which instructions, so called commands can be called. Every extension can provide custom commands.
CppTools [20]	Name of the C/C++ for Visual Studio Code extension that provides C/C++ support in Visual Studio Code
CUTE [2]	Stands for C++ Unit Testing Easier. Is a C++ testing framework
C++	General-purpose programming language
c4 Model [60]	A standard for the architecture description. Was used in the chapter Design
Debugger	Tool that allows it to step through a programs code statement by statement
DiagnosticCollection [64]	Is a collection that keeps track of a certain set of code diagnostics e.g. CUTE test warnings
Difference Viewer	User Interface that simplifies the understanding of assertion failures

Docker [79]	Offers OS-level virtualization. Allows it to deliver software in so called Containers
Eclipse	Popular integrated development environment (IDE) in the Java environment
Extension Manifest	Contains some definitions of an extension. Is equal to the package.json file in the extension root directory
Extension Entry Point	Contains the activate and deactivate functions that are called when the extension is activated or deactivated. Per default this is equal to the extension.ts file in the extension root directory
Epic	Larger body of work, that can be broken down into smaller tasks
GCC [15]	Offers compiler features for C++
GDB [85]	Portable debuggers that supports debugging C++ code amongst other languages. Runs on many Unix-like systems
Git	Distributed version control system on which the project's repositories are based
GitLab [74]	DevOps software where the project's work items and code repositories are hosted
Green/Red-Bar View	User Interface that visualizes the test outcomes in an easy graspable way
IDE	Integrated Development Environment that offers basic tools to write and test code
ISO25010 [12]	Standard to discover and define software quality measures
JSON	JavaScript Object Notation. Lightweight data format
JSON-RPC [57]	Lightweight remote procedure call protocol on which the language server protocol (LSP) builds
JUnit	Unit testing framework for the Java environment
Language Server Provider	Extension that provides language information for a specific programming language using the language server protocol (LSP) [18]
LaTeX [78]	Is a mark up language on which this documentation was built
Linux	Family of Unix-like operating systems based on the linux kernel
Linting	Step of the CI pipeline where the code style is analyzed against the code style guidelines

LLDB [86]	Is a debugger that supports debugging C++ code. Is the debugger component of the LLVM project
LSP [18]	Stands for Language Server Protocol which is a standardized protocol used by language server provider extension to communicate with the underlying language servers
Macro	Is a part of the code that gets replaced by a predefined value during the compile process
Mocha [75]	JavaScript test framework that runs on node
MSVC [17]	Stands for Microsoft Visual C++ which is a compiler for C++ amongst other languages
NFR	Non-functional requirement. All requirements that have nothing to do with a products functionality
Native Debug [58]	Visual Studio Code extension that offers debugging capabilities. Supports both GDB and LLDB
OST [1]	Easter Switzerland University of Applied Sciences
Quick Fix	Provide an easy option to mitigate a code problem. Can be accessed over the lightbulb symbol in VS Code
RUP [73]	Rational Unified Process. Software development process framework
Scrum [72]	Agile project management framework
Scrum+	OST internally known project management framework. Combination of Scrum and RUP.
Snippets	Are templates of code objects that are frequently used and therefore don't need to be typed manually every time. A snippet for CUTE test cases is offered by the CUTE extension
SonarQube	Tool that offers continuous code inspection on which it provides a set of metrics. Is part of the CI pipeline
StdOut	Standard output of an application. The normal output of a program gets sent to the standard output.
Testing API [37]	Native API that offers the basic testing infrastructure such as the test explorer view
Test Explorer UI [23]	Extension for VSCode that offers basic UI and run handling support for tests. Many adapters available. Deprecated since July 2021

TestMate [39]	Extension for VSCode that offers testing functionality for a set of C++ testing frameworks
TS-Mockito [76]	Is a mocking library for TypeScript
TypeScript	Programming language that is based on JavaScript with optional static type support. VS Code extensions have to be implemented in TypeScript
VS Code [4]	Short for Visual Studio Code. Open source and platform independent code editor by Microsoft
VS Code Marketplace [8]	Offers a large number of Visual Studio Code extensions that can be installed from there
Windows	Popular operating system by Microsoft
XML	Extensible Markup Language. File format of the test results

12 Bibliography

References

- [1] <https://www.ost.ch/>
- [2] <https://cute-test.com/>
- [3] <https://www.cevelop.com/>
- [4] <https://code.visualstudio.com/>
- [5] <https://cute-test.com/guides/cute-eclipse-plugin-guide/>
- [6] <https://www.eclipse.org/cdt/>
- [7] <https://www.ost.ch/de/forschung-und-dienstleistungen/informatik/ifs-institut-fuer-softwares>
- [8] <https://marketplace.visualstudio.com/items?itemName=CUTETest.cute-testing>
- [9] <https://cmake.org/>
- [10] <https://github.com/PeterSommerlad/CUTE>
- [11] <https://cute-test.com/img/cute-diff-view.png>
- [12] <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>
- [13] <https://docs.gitlab.com/ee/ci/pipelines/>
- [14] <https://www.sonarqube.org/>
- [15] <https://gcc.gnu.org/>
- [16] <https://clang.llvm.org/>
- [17] <https://visualstudio.microsoft.com/de/vs/features/cplusplus/>
- [18] <https://microsoft.github.io/language-server-protocol/>
- [19] <https://clangd.llvm.org/>
- [20] <https://github.com/microsoft/vscode-cpptools>
- [21] <https://www.ibm.com/docs/en/developer-for-zos/9.1.1?topic=formats-junit-xml-format>
- [22] <https://marketplace.eclipse.org/content/junit-tools>
- [23] <https://github.com/hbenl/vscode-test-explorer>
- [24] <https://github.com/hbenl/vscode-test-explorer#test-adapters>
- [25] <https://github.com/hbenl/vscode-example-test-controller>
- [26] <https://github.com/microsoft/vscode-docs/blob/vnext/api/extension-guides/testing.md#migrating-from-the-test-explorer-ui>
- [27] <https://marketplace.visualstudio.com/items?itemName=matepek.vscode-catch2-test-adapter>

- [28] <https://marketplace.visualstudio.com/items?itemName=fredericbonnet.cmake-test-adapter>
- [29] <https://marketplace.visualstudio.com/items?itemName=drleq.vscode-cpputf-test-adapter>
- [30] <https://marketplace.visualstudio.com/items?itemName=dampsoft.vscode-banditcpp-test-adapter>
- [31] <https://marketplace.visualstudio.com/items?itemName=betwo.b2-catkin-tools>
- [32] <https://marketplace.visualstudio.com/items?itemName=bneumann.cputest-test-adapter>
- [33] <https://marketplace.visualstudio.com/items?itemName=zcoinofficial.boost-test-adapter>
- [34] <https://marketplace.visualstudio.com/items?itemName=Moosecasa.vscode-acutest-test-adapter>
- [35] <https://marketplace.visualstudio.com/items?itemName=dprog.vscode-cppunit-test-adapter>
- [36] https://code.visualstudio.com/updates/v1_59
- [37] <https://code.visualstudio.com/api/extension-guides/testing>
- [38] <https://github.com/microsoft/vscode-extension-samples/tree/main/test-provider-sample>
- [39] <https://github.com/matepek/vscode-catch2-test-adapter>
- [40] <https://github.com/matepek/vscode-catch2-test-adapter/tree/legacy>
- [41] <https://github.com/catchorg/Catch2>
- [42] <https://github.com/google/googletest>
- [43] <https://github.com/onqtam/doctest>
- [44] <https://github.com/google/benchmark>
- [45] <https://github.com/matepek/vscode-catch2-test-adapter/pull/276>
- [46] <https://code.visualstudio.com/api/references/vscode-api#TestController>
- [47] <https://code.visualstudio.com/api/references/vscode-api#TestItem>
- [48] <https://github.com/matepek/vscode-catch2-test-adapter/blob/master/src/framework/ExecutableFactory.ts>
- [49] <https://github.com/matepek/vscode-catch2-test-adapter/blob/master/src/framework/GoogleTest/GoogleTestExecutable.ts>
- [50] https://github.com/PeterSommerlad/CUTE/blob/master/cute/cute_test.h
- [51] <https://github.com/microsoft/vscode-java-test/blob/main/src/runners/junitRunner/JUnitRunnerResultAnalyzer.ts>
- [52] <https://microsoft.github.io/language-server-protocol/implementors/servers/>

- [53] <https://code.visualstudio.com/docs/editor/intellisense#:~:text=IntelliSense%20is%20a%20general%20term,%2C%20and%20%22code%20hinting.%22>
- [54] <https://code.visualstudio.com/assets/api/language-extensions/language-server-extension-guide/lsp-languages-editors.png>
- [55] <https://code.visualstudio.com/api/references/commands>
- [56] <https://github.com/llvm/llvm-project>
- [57] <https://www.jsonrpc.org/>
- [58] <https://github.com/WebFreak001/code-debug>
- [59] <https://github.com/vadimcn/vscode-lldb>
- [60] <https://c4model.com/>
- [61] <https://code.visualstudio.com/api/extension-guides/command>
- [62] <https://refactoring.guru/design-patterns/composite>
- [63] https://code.visualstudio.com/docs/editor/refactoring#_code-actions-quick-fixes-and-refactorings
- [64] <https://code.visualstudio.com/api/references/vscode-api#DiagnosticCollection>
- [65] <https://refactoring.guru/introduce-null-object>
- [66] <https://code.visualstudio.com/api/get-started/extension-anatomy#extension-file-structure>
- [67] <https://code.visualstudio.com/api/references/contribution-points#contributes.snippets>
- [68] <https://code.visualstudio.com/api/references/contribution-points#contributes.commands>
- [69] <https://code.visualstudio.com/api/references/activation-events#onStartupFinished>
- [70] <https://github.com/airbnb/javascript>
- [71] https://en.wikipedia.org/wiki/Regular_expression
- [72] <https://www.scrum.org/resources/what-is-scrum>
- [73] https://en.wikipedia.org/wiki/Rational_Unified_Process
- [74] <https://about.gitlab.com/>
- [75] <https://mochajs.org/>
- [76] <https://github.com/NagRock/ts-mockito>
- [77] <https://code.visualstudio.com/docs/remote/containers>
- [78] <https://www.latex-project.org/>
- [79] <https://www.docker.com/>

- [80] <https://github.com/streetsidesoftware/vscode-spell-checker>
- [81] <https://prettier.io/docs/en/install.html>
- [82] <https://www.electronjs.org/docs/latest/tutorial/automated-testing>
- [83] <https://code.visualstudio.com/api/advanced-topics/extension-host>
- [84] <https://code.visualstudio.com/docs/getstarted/settings>
- [85] <https://www.sourceware.org/gdb/>
- [86] <https://lldb.llvm.org/>
- [87] <https://code.visualstudio.com/api/references/vscode-api#CodeActionProvider%3C%3E>
- [88] https://cmake.org/cmake/help/latest/variable/CMAKE_EXPORT_COMPILE_COMMANDS.html
- [89] <https://www.mingw-w64.org/>
- [90] <https://www.youtube.com/channel/UCx-sWK8pqdoQ4QRTUWrX2pw>

13 Content Lists

List of Figures

1	Cevelop CUTE Plug-In	4
2	VS Code CUTE Editor Integration	5
3	VS Code CUTE Extension UI	6
4	VS Code CUTE Unregistered Test	7
5	VS Code CUTE Unregistered Suite	7
6	VS Code CUTE Legacy Syntax	7
7	VS Code CUTE YouTube Instructions	8
8	Use case diagram	13
9	Cevelop Green/Red-Bar View [11]	15
10	Cevelop Assert Difference Viewer	16
11	Test Explorer View	18
12	Structure diagram	26
13	Test discovery sequence diagram	27
14	Test run sequence diagram	28
15	LSP Comparison [54]	39
16	CUTE testing on YouTube [90]	42
17	Extension context diagram	43
18	Component diagram	47
19	Testing component diagram	51
20	Tools component diagram	53
21	CodeActionProvider diagram	54
22	CodeGenerationProvider diagram	55
23	QuickFixProvider diagram	56
24	CodeAnalyzer diagram	57
25	Debugger diagram	58
26	Environment component diagram	59
27	TestFinderMode Setting	67
28	LanguageClientMode Setting	68
29	TestRunHandler sequence diagram	69
30	TestDebugHandler sequence diagram	71
31	ExecutableTestFinder sequence diagram	72
32	CodeTestFinder sequence diagram	74
33	Find Test Runners sequence diagram	76
34	Find Test Suite sequence diagram	78
35	Get Runner Call sequence diagram	79
36	Find New Test sequence diagram	80
37	Find Legacy Test sequence diagram	82
38	Test Explorer	85
39	VS Code CUTE inline integration	86
40	VS Code CUTE run history	86
41	The VS Code CUTE project generator	87
42	Structure of a newly generated project	87
43	Legacy Syntax Settings	88
44	Legacy Syntax Update pop-up	88
45	Assertion Failure	88
46	Unregistered Test Editor Warning	89
47	Unregistered Test Existing Suite Fix	90

48	Unregistered Test New Suite Fix	90
49	Unregistered Suite Editor Warning	91
50	Unregistered Suite Fix	92
51	Legacy Syntax Editor Warning	93
52	Legacy Syntax Fix	93
53	Feature Comparison	94
54	SonarQube Project Size	95
55	Component Coverage Result	95
56	SonarQube Project Metrics	96
57	Test Explorer Clangd Code-Based Discovery Editor	98
58	Test Explorer CppTools Code-Based Discovery Explorer	99
59	Test Explorer Executable-Based Discovery Editor	100
60	Test Debugging Breakpoint	101
61	Test Debugging Result	101
62	Test Run Single Test	102
63	Test Run Test Suite Tests	103
64	Unregistered Test Warning	104
65	Unregistered Test QuickFix 1	104
66	Unregistered Test Quick Fix 2	105
67	Unregistered Suite Warning	106
68	Unregistered Suite Quick Fix	106
69	Legacy Syntax Warning	107
70	Legacy Syntax Quick Fix	107
71	Legacy Syntax Warning Pop-Up	108
72	CUTE Project Template	109
73	Create New Suite Command	110
74	TEST Snippet	111
75	Assertion Failure Visualization	112
76	Work Breakdown Structure Diagram	119
77	Project Timeline	123
78	Risk Matrix	127
79	Development Containers Architecture [77]	129
80	CUTE Extension Pipeline	129
81	GitLab Test Report	130
82	Development Server structure	131

Listings

1	CUTE Executable Interface	23
2	CUTE Executable StdOut Result	24
3	CUTE Executable XML Result	24
4	Test Executable Identification [48]	32
5	Executable based test discovery [49]	33
6	cute_test.h test instantiation	34
7	CUTE Executable StdOut Success	35
8	CUTE Executable StdOut Failure	35
9	CUTE Executable XML Result Success	36
10	CUTE Executable XML Result Failure	36
11	TEST macro v1	60
12	TEST macro v2	61
13	TEST macro v3	61
14	Extension Standard Folder Structure [66]	62
15	General Extension Properties	63
16	Extension Commands	64
17	Extension Snippets	64
18	Extension Entry Point	65
19	TestRun Folder Structure	70
20	cute_runner.h makeRunner implementation	77
21	CUTE Runner Instantiation	77
22	CUTE Runner Call Operators	78
23	RunSuite Function	79
24	Sample New CUTE TEST	81
25	CUTE Test Constructors [50]	83
26	Sample Legacy Test	84

A General Testimonial

This chapter contains an overview of the experiences that were made throughout the course of the project. The experiences are described separately for the areas Team and Communication, Meetings, Challenges, and finally from the personal perspective of each project member.

A.1 Team / Organization / Communication

The collaboration within the project team worked without any major difficulties. The therefore used collaboration tools such as GitLab and Microsoft Teams provided the needed flexibility to work time and location independently. The communication with the supervisor of this project always worked without any issues. The intermediate presentation brought up valuable findings, which were afterwards used for the further development of the product and for the documentation of the project. Due to health problems and the back then still difficult situation with the ongoing COVID-19 pandemic, the intermediate presentation had to be held online. The final presentation on the other hand will be held in person.

A.2 Meetings

Throughout the project meetings with the project supervisor were held on a weekly basis. The meetings were scheduled for Monday afternoon from 4:00 until 5:00. During these weekly meetings the project's progress and potential problems were discussed. Furthermore, the planning for the upcoming week was discussed and adjusted if needed. After the meeting with the project supervisor a short meeting was held within the project team to discuss the findings of the meetings and finalize the weekly planning.

A.3 Challenges

Throughout the course of the project challenging situations from a technical point of view occurred frequently. Especially during the elaboration phase, while creating prototypes approaches sometimes turned out not to be working. Thanks to the risk analysis and the thereby defined mitigation strategies, none of these setbacks could endanger the course of the project and a successful completion. The biggest technical difficulties had to do with the integration of the different language server provider. More detailed information about these problems can be found in the chapter Decisions.

A.3.1 Personal Experience

This section contains the experiences, that were made throughout the project, from a personal point of view. Each of the project members' views about the project and the experiences made throughout the project's duration can be found below.

A.3.2 Dominic Klinger

I was able to gain quite a bit of experience again last semester. The project gave enough opportunities to try things out again. Afterwards, I was able to implement what I had tried. I also was able to use my previous experience in Continuous Integration and still learned a lot of new things again. The projects are very individual, which makes Continuous Integration always a new challenge. I was also able to successfully apply what I had learned in previous semesters. For example, the basics of compiler construction. However, the project also showed where the problems are in such an unknown area. Especially with the tooling and the assembly of the pipeline it became apparent that some things would have been better suited at an earlier stage of the project. For example, the linting, which would have been very helpful from the beginning. So, we refactored parts of the project later because we didn't integrate the linter with the coding style guide in an early stage. Testing in general was a difficulty because this was a bit poorly described on how to best implement this. Especially in the area of testing coverage with NYC/Istanbul was quite a heavy matter. If JavaScript files were already loaded; it happens that the coverage is not captured because the instrumentation of the files by Istanbul has not yet taken place. In the end, I am very satisfied with the result and I am convinced that we have created a good product.

A.3.3 Christian Bisig

The project that was conducted in the scope of this thesis, was very fascination but from time to time also quite challenging as I've never developed a Visual Studio Code extension before and was not familiar with the integration possibilities of language information providers. The large number of technologies and methods which I had no prior experience in, made this assignment a very interesting task. Existing experience from different areas known from school or from work had to be combined in order to find an optimal solution. At the end of the project a solution can be presented, that fulfills all functional and non-functional requirements, which were defined based on the assignment during the project planning. Executing such projects in teams always provides the possibility to improve the project management and communication skills.

A.4 Conclusion

Beside many challenges throughout the course of the project, in the end a working product could be presented. The developed CUTE extension for Visual Studio Code covers all functional and non-functional requirements and is ready to use in the scope of the C++ modules offered at the OST Eastern Switzerland University of Applied Sciences. The next steps are described in the sections above. One of the most important next steps is to get in the feedback from actual users, that use the CUTE extension during their C++ exercises.

B Time report

This chapter contains an overview of the actually spent time. In the first section of the chapter an overview of the whole project can be found. Subsequent to this project overview, a more detailed listing of the time spent can be found. Thereby the time is split into the corresponding work items, that were created on GitLab. The time tracking was done using the GitLab time tracking functionality. An up to date overview of the spent time was made available on a dashboard, that was accessible by all team members and the project's supervisor.

The time spent is evenly split between the two project members and in total adds up to approximately 775 hours. In comparison to the 726 hours that resulted from the planning on epic level, this is an additional effort of approximately 6% which is an acceptable number.

B.1 Time stats

- total spent: 96d 7h 15m
- spent: 96d 7h 15m
- Dominic Klinger: 48d 3h
- Christian Bisig: 48d 4h 15m

B.2 Issues

Issue ID	Title	Spent
30	Testing	1d 6h
29	Programming	22d 2h
28	Analyzing	2d
27	Publishing	4h
26	Design Refactoring for NFR fulfillment	1d 5h
25	Setup UnitTesting for CUTE Extension	1d 6h
24	Implement TestDebugging Logic	1d 2h
23	Implement TestExecution Logic	1d
22	Implement TestDetection	2d 2h
21	Plan architecture	2d 2h 30m
20	Evaluate clangd extension as replacement for cpptools	1d 2h
19	Create command to create new TestSuite files	1d 1h
18	Create quickfix to convert legacy test declarations to TEST(...) declarations	1d 2h
17	Create quickfix to register tests to new testsuites	1d 1h
16	Create quickfix to register unregistered testcases to existing testsuites	7h
15	Find options to discover legacy test declarations	1d 7h
15	Documentation	8d 4h

Issue ID	Title	Spent
14	Find options to discover unused testsuites	1d 5h
14	Write Abstract & Management Summary	6h
13	Investigate options to warn users about unregistered tests	1d 4h 30m
13	Implementation documentation	1d
12	Decisions Documentation	1d 1h
12	Investigate features that make test development easier	1d 30m
11	Document Project Management	6h
11	Evaluate options to use vs-code snippets for code generation	2h
10	Documentation about prototype	
10	Prepare features for alpha version	5h
9	Plan exceptionhandling and fault tolerance	1d 4h 30m
9	Documentation template project	
8	Documentation about development environment	
8	Videodreh	3d 1h
8	Evaluate options to find test implementations	3d 6h
7	Test code listings	
7	Zwischenpräsentation	2h
7	Implement XML-Result parser	1d 2h 30m
6	Implement PlantUML	
6	Zwischenpräsentation vorbereiten	1d 4h
6	Investigate possibilities to run tests parallel	4h
6	Create alpha version for presentation	2h 30m
5	Integrate new CUTE Extension	
5	Investigate possibilities to debug tests	6h
5	System context (diagram)	4h
5	Meetings	3d 6h 45m
4	Setup unit / integration testing infrastructure for the extension	2h
4	Investigate existing VS Code Test Extensions	3d 6h
4	Non functional requirements	3h
4	Investigate testresult processing	2h
3	Create testextension prototype	1d 7h 30m
3	Familiarize with VS Code Extensions	3d 7h
3	Investigate test execution	2h
3	Functional reuirements	
2	Use cases / User stories	4h
2	Setup template project	1d
2	Investigate test registration (vs code testingApi)	4h
2	Setup Pipeline	
1	Risk analysis	5h
1	Initialize extension project template	3h
1	Investigate test detection	2d
1	implement –help flag	1h
1	Setup infrastructure	1d 6h

C Meeting protocols

This section contains all meeting protocols and within them the weekly planning. The meeting was held on a weekly basis and scheduled for Monday from 4:00pm to 5:00pm. The project team and the project supervisor took part in these weekly meetings. The protocols were maintained in OneNote to simplify collaboration within the project team.

C.1 Week1

Vergangene Woche:

- Infrastruktur aufgesetzt
 - Virtual Machine
 - Sonarqube
 - Timereport
 - Landing Page
 - Repository
 - GitLab Group Runner
- Template Testprojekt C++ mit CMake
- Erstellung Beispielextension für VS Code
- Erkundung bestehender Testextension für gTest (Funktionsweise, Testdetektion, etc.)

Aktuell/Abklärungen:

- Abnahme LaTeX Template Dokumentation
 - Code listings testen
 - Bachelor thesis
 -
- Abklärung Einbindung in CUTE
(bestehende XML-Dateien nach ersten Durchlauf oder anderer Weg z.B. CLI Argument)
 - Harter Crash führt zu nicht kompletten Output
 - XML Output könnte malformed sein
 - Segfaults testen!
 - Doppeldeutigkeit ausschliessen bzw behandeln
 - List-Test Flag
 - Output via std::cout

Geplante Arbeiten:

- Erkundung von Testmate mit C++ Projekt basiert auf gTest (Handling und Funktionsabgleich)
- LaTeX Template finalisierung und erste erledigte Schritte dokumentieren und in Repository packen mit Buildpipeline
- Administration – Erstellung einzelner Workitems im GitLab Groupspace
- Erstellung eines Demo Typescript Projekts mit einer Pipeline für die Erstellung der Extension als Artifact, Einbindung in Typescript und Unit Tests

C.2 Week2

Vergangene Woche:

- TestMate Analyse google test
 - Test Discovery
 - VS Code testing api integration
 - Test Registration
 - Test Execution
 - Testresult Analysis – integration testexplorer
 - Analyseerkenntniss unter folgendem Link ersichtlich:
https://ostch-my.sharepoint.com/:o/g/personal/christian_bisig_ost_ch/EvRzdpAwGNIGLU4sQ2mzlhQBK1sISZ6LXSqh9YloBfPitQ?e=3jMBXt
- Aufsetzen einer Demo Extension (Prototyp für CUTE Testing-Extension)
 - Tests werden detektiert
 - Tests werden sauber im Testexplorer angezeigt
 - Tests können ausgeführt werden
 - Analyse der Testresultate mit Visualisierung der Ergebnisse in VS Code funktioniert
 - Unit Test Umgebung zum Testen der Extension wurde vorbereitet
 - Pipeline zum Builden der Extension (Artifact) wurde auf GitLab aufgesetzt
- LaTeX Template wurde finalisiert
 - Pipeline zur PDF Generierung aufgesetzt
 - Umgebung, die das Editieren der Dokumentation vereinfachen soll wurde aufgebaut
- Aktuell benötigte Workitems wurden in den dazugehörigen Projekten erstellt
- CUTE wurde um ein "--display-tests" Flag erweitert
 - Wird verwendet um die Tests zu detektieren
 - Output des Resultates auf der Konsole

Aktuell/Abklärungen:

- CUTE Erweiterung
 - Detektierung der .cpp Dateien und Zeilennummern zu den jeweiligen Tests
 - Varianten/Optionen?
 - Rückgabe der Informationen (Format)

Geplante Arbeiten:

- Prototyp CUTE Extension (demo-extension) finalisieren
 - Hardcoded Infos umbauen / Erarbeitung aller benötigten Informationen zur Testausführung
 - Verwendung der Resultate im XML (junit) Format
 - Unit- / Integrationtests ausarbeiten (Testabdeckung von 80% angestrebt)
 - Evaluieren der Möglichkeiten zum Debuggen von Tests
 - Entscheid der anzuwendenden Variante basierend auf Prototyp
- CUTE Erweiterung zur Zuweisung von .cpp Dateien und Zeilennummern zu den einzelnen Testfällen
 - Varianten evaluieren und Prototyp erstellen
- Entscheidungen Dokumentieren
- Workitems/Zeiterfassung nachführen um weiteres Vorgehen im Projekt zu planen

C.3 Week3

Vergangene Woche:

Demo-Extension Prototyp

Debugging der Testfälle funktioniert

`vadimcn.vscode-lldb`

<https://marketplace.visualstudio.com/items?itemName=vadimcn.vscode-lldb>

`webfreak.debug`

<https://marketplace.visualstudio.com/items?itemName=webfreak.debug>

`ms-vscode.cpptools`

<https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools>

Tests können parallel ausgeführt werden

Resultat wird aus XML Datei ermittelt

Gewisse hartcodierte Elemente wurden ersetzt und werden jetzt korrekt detektiert

Executables die CUTE Tests enthalten

Workspace Informationen

CUTE wurde um ein "--help" Flag erweitert

Wird verwendet um zu entscheiden, ob es sich beim executable um CUTE Tests handelt

Output des Resultates auf der Konsole

Aktuell/Abklärungen:

CUTE Erweiterung

Detektierung der .cpp Dateien und Zeilennummern zu den jeweiligen Tests

Aktueller Stand und gefundene Möglichkeiten diskutieren

Termin Zwischenpräsentation:

Vorschläge Frieder Loch:

Dienstag 5. April (1300-1800) - Di 13:00-15:00 (geht nur vor Ort)

Geplante Arbeiten:

Prototyp CUTE Extension (demo-extension) finalisieren

Workspace Überwachung zur Aktualisierung der Testfälle bei neuem Kompilieren von geänderten Tests.

Hartcodierte Infos umbauen / Erarbeitung aller benötigten Infos zur Testausführung

Zeilen und Datei Informationen aus CUTE

Verbessern der Stabilität und Zuverlässigkeit in Fehlerfällen

Fehlerbehandlungskonzept planen und umsetzen

Unit- / Integrationstests ausarbeiten (Testabdeckung von 80% angestrebt)

CUTE Erweiterung zur Zuweisung von .cpp Dateien und Zeilennummern zu den einzelnen Testfällen

Beste evaluierte Variante Implementieren und Funktionalität in CUTE anbieten

Entscheidungen Dokumentieren

Workitems/Zeiterfassung nachführen um weiteres Vorgehen im Projekt zu planen

C.4 Week4

Vergangene Woche:

- Demo-Extension Prototyp
 - Workspace Überwachung erweitert
 - Reagiert auf Dateiänderungen im Workspace (Create, Update, Delete)
 - Begrenzt auf Executables, die mittels -help Flag nach CUTE Tests überprüft werden
 - Hartcodierte Infos zur Datei und Zeile werden jetzt dynamisch aus CUTE gelesen
 - Evaluation von Fallbacklösungen zur Detektion von Testimplementationen wurde gestartet
 - Basierend auf Language Servern (FindReferences Funktionalität -> Detektion von Stellen an denen eine CuteTest Instanz erstellt wird, respektive das CUTE Makro verwendet wird. --> Get Declaration of the specific testmethods
 - Microsoft vscode-cpptools Extension
 - Clangd Extension
- CUTE wurde um Funktionalität zur Detektion von Testimplementationen erweitert
 - Erweiterung der CUTETest Klasse gemäss Besprechung vom 14.03.2022
 - Output des Resultates auf der Konsole

Aktuell/Abklärungen:

- Fallbacklösungen zur Detektion von Testimplementationen
 - Zusätzliche Language Server Varianten
 - Regex Variante

Geplante Arbeiten:

- Prototyp CUTE Extension (demo-extension) finalisieren
 - Verbessern der Stabilität und Zuverlässigkeit in Fehlerfällen
 - Fehlerbehandlungskonzept planen und umsetzen
 - Test Detektionsfunktionalität Testen (CUTE Erweiterung und Demo-Extension Integration)
 - Implementation Fallbacklösung basierend auf folgenden Ansätzen:
 - Language Server Funktionalität aus bestehenden Extension Installationen
 - Regex Suche als letzte Möglichkeit zur Detektion von Implementationen
 - Behandeln des Falles, wenn keine Informationen zur Location verfügbar sind
 - Unit- / Integrationstests ausarbeiten (Testabdeckung von 80% angestrebt)
- Entscheidungen Dokumentieren
- Workitems/Zeiterfassung nachführen um weiteres Vorgehen im Projekt zu planen

C.5 Week5

Vergangene Woche:

- Demo-Extension Prototyp
 - Fehlerbehandlung wurde analysiert und mit der Planung begonnen
 - CUTE Erweiterung bezüglich Test Lokalisierung wurde mit Prototyp evaluiert
 - Vscod Dokumentation bezüglich Extension Integrationen wurde analysiert
 - Tests mit diversen Languageserver Implementationen wurden durchgeführt
 - Prototyp bezüglich Fallbacklösung für die Test Lokalisierung implementiert
 - Basiert auf Vscod Commands API, das auf die installierten Extensions zugreift
 - Zuweisung von Testfällen zu Testsuiten basiert teilweise auf Regex
 - Dokumentation wurde weitergeführt

Aktuell/Abklärungen:

- Fallbacklösungen zur Detektion von Testimplementationen
 - Abklärungen bezüglich Zuverlässigkeitsanforderungen
 - Aktueller Prototyp kann bei Testsuite Zuweisungen fehlerhafte oder unvollständige Resultate liefern

Geplante Arbeiten:

- Prototyp CUTE Extension (demo-extension) alpha Version erstellen
 - Verbessern der Stabilität und Zuverlässigkeit in Fehlerfällen
 - Geplantes Konzept implementieren und dokumentieren
 - Fallbacklösung für Test Lokalisierung aufräumen
 - Hartcodierte Teile entfernen und Code optimieren
 - An richtiger Stelle aufrufen (nur wenn CUTE Erweiterung nicht verfügbar ist)
 - Behandeln des Falles, wenn keine Informationen zur Location verfügbar sind
 - Unit- / Integrationstests ausarbeiten (Testabdeckung von 80% angestrebt)
- Alpha Version Dokumentieren (Alle getroffenen Entscheidungen und Ansatzpunkte)
- Präsentation vorbereiten (Zwischenpräsentation und Demonstration)
- Workitems/Zeiterfassung nachführen um weiteres Vorgehen im Projekt zu planen

C.6 Week6

Vergangene Woche:

- CUTE-Extension alpha Version
 - Fehlerbehandlung wurde verbessert
 - Mit fehlerhaften Resultat XML Dateien kann umgegangen werden.
 - Fallbacklösung zur Detektion von Testfällen wurde eingebaut
 - Umgang mit nicht vorhandenen Informationen bezüglich Ort der Implementation von Testfällen wurde verbessert
 - Fallbacklösung zur Test Lokalisation
 - Detektion der cute::test Konstruktor Aufrufen implementiert
 - Basierend auf cute::test Referenzen – Regex matching von Klassenverwendung
 - Refactoring zum Stand einer Alpha Version (Hartcodierte Stellen ausprogrammiert)
 - Analyse zur Anwendung von VS-Code Snippets zur Vereinfachung der Testimplementation und Registrierung begonnen
 - Dokumentation wurde weitergeführt
- Zwischenpräsentation
 - Inhalt ausgearbeitet
 - PowerPoint Präsentation vorbereitet
 - Alpha Version der CUTE-Extension für die Demo vorbereitet
- CUTE
 - Bugfixing von TEST Makro

Aktuell/Abklärungen:

- Detektion von Test Executables
 - Bei nicht vorhandener CUTE Erweiterung bezüglich --help Flag
- Project Template
 - Umfang / Vorstellung abgleichen
 - Implementationsmöglichkeiten

Geplante Arbeiten:

- Prototyp CUTE Extension
 - Erarbeiten von Möglichkeiten zur Anwendung von Snippets zur Code Generierung
 - Erste Konkrete Anwendungen in Prototyp einbauen
 - Ausarbeiten von Möglichkeiten zum Aufsetzen von Template Testprojekten
 - Unit- / Integrationstests ausarbeiten (Testabdeckung von 80% angestrebt)
- Rückmeldungen aus Zwischenpräsentation analysieren und in die Planung einfließen lassen
- Entscheide bezüglich Code Generierung und Project Template dokumentieren
- Workitems/Zeiterfassung nachführen um weiteres Vorgehen im Projekt zu planen

C.7 Week8

Vergangene Wochen:

- CUTE-Extension
 - Detektion von nicht registrierten TestCases
 - Warnung – Wird im Texteditor angezeigt, wenn ein Testfall keiner Suite zugefügt worden ist und nicht direkt einem Runner übergeben wird
 - Quick Fix
 - Hinzufügen des Testfalls an bestehender TestSuite
 - [CUTE Extension Register Test to existing Suite](#)
 - Erstellen einer neuen TestSuite und Registrierung des Testfalls
 - [CUTE Extension Create and register new Suite](#)
 - Detektion von nicht ausgeführten TestSuites
 - Warnung – Wird im Texteditor angezeigt, wenn eine Testsuite keinem Runner übergeben wird
 - Quick Fix
 - Registrierung der Testsuite bei einem Runner in "main" Methode
 - Detektion von Legacy Syntax Test Deklarationen
 - Warnung – Wird im Texteditor angezeigt wenn das CUTE Makro verwendet wird zur Erstellung eines Testfalls
 - Quick Fix
 - Update zu TEST(...) Makro
 - [CUTE Extension LegacySyntax Update](#)
 - Template Projekt
 - Anpassungen entsprechend Snippet Funktionalität
 - Neuer Aufbau von Testdateien
 - Behebung von Bug bei mehreren TestSuites
 - Dokumentation wurde weitergeführt
- Zwischenpräsentation
 - Nachbesprechung wurde durchgeführt
 - Wichtige Punkte für den weiteren Verlauf und die Dokumentation wurden vermerkt

Geplante Arbeiten:

- CUTE Extension
 - Ausarbeiten von Extension Design
 - Komponenten mit jeweiligen Zuständigkeiten definieren
 - Dokumentieren der Klassenarchitektur
 - Implementation des Extension Grundgerüsts
 - Definieren von möglichen zusätzlichen Features für bessere Usability
 - Unit- / Integrationstests ausarbeiten (Testabdeckung von 80% angestrebt)
- Code Generierung für zusätzliche Testfiles ausarbeiten
- Workitems/Zeiterfassung nachführen um weiteres Vorgehen im Projekt zu planen

C.8 Week9

Vergangene Woche:

- CUTE-Extension
 - Clangd
 - Support für Clangd VSCode extension als Ersatz für microsoft cpptools
 - Verbesserte Zuverlässigkeit & Performanz
 - Teilweise Abweichungen in der Funktionalität - im Vergleich zu cpptools
 - Operator Referenzen können gesucht werden
 - Andere Benennung von Symbolen
 - Update von legacy Testdeklarationen
 - Option zum alle Testdeklarationen auf einmal upzudaten
 - Option zum einzelne Testdeklarationen upzudaten
 - Option zum Belassen der alten Testdeklarationen mit dem Verzicht auf den vollen Funktionsumfang der Extension
 - [CUTE Extension Update all legacy code sections](#)
 - TestSuite Dateien
 - Command zum Erzeugen von TestSuites in neuen Dateien
 - Registration in cmake File
 - Nach builden in Testexplorer verfügbar
 - [\(19\) CUTE Extension create new SuiteFile - YouTube](#)
 - Architektur / Design
 - Planung gestartet
 - Domainklassen analysiert und geplant
 - Unterstützung von cpptools und clangd extension
 - Template Method Pattern
 - Dokumentation wurde weitergeführt

Geplante Arbeiten:

- CUTE Extension
 - Dokumentieren und Umsetzen der Extension Architektur
 - Basierend auf Prototyp und Designplanung
 - Übernehmen der gewünschten Features aus Prototyp
 - Unit- / Integrationstests ausarbeiten (Testabdeckung von 80% angestrebt)
- Workitems/Zeiterfassung nachführen um weiteres Vorgehen im Projekt zu planen

C.9 Week10

Vergangene Woche:

- CUTE-Extension
 - Architektur & Design
 - TestCase-Discovery über .exe - Design wurde geplant
 - Umsetzung in Gang
 - TestCase-Discovery über LanguageInfos - Design wurde geplant
 - Umsetzung in Gang
 - TestExecution - Design wurde geplant
 - Umsetzung in Gang
 - TestDebugging – Design wurde geplant
 - Umsetzung in Gang
 - Clangd Integration
 - Performance Verbesserungen
 - CppTools
 - Stabilisierende Massnahmen
 - Dokumentation wurde weitergeführt
- CUTE
 - Namespace Bug wurde gefixt

Geplante Arbeiten:

- CUTE-Extension
 - TestExecution
 - Abschluss der Implementation
 - Dokumentation vom finalen Design
 - Testabdeckung
 - TestDebugging
 - Abschluss der Implementation
 - Dokumentation vom finalen Design
 - Testabdeckung
 - TestCase-Discovery
 - Abschluss der Implementation
 - Dokumentation vom finalen Design
 - Testabdeckung
 - Productivity Tools – Warnings & Quickfixes
 - Planung Design
 - Unit- / Integrationstests ausarbeiten (Testabdeckung von 80% angestrebt)
- Workitems/Zeiterfassung nachführen um weiteres Vorgehen im Projekt zu planen

C.10 Week11

Vergangene Woche:

- CUTE-Extension
 - Architektur & Design
 - TestCase-Discovery über .exe - Design wurde umgesetzt und grob dokumentiert
 - TestCase-Discovery über Code - Design wurde umgesetzt und grob dokumentiert
 - TestExecution - Design wurde umgesetzt und grob dokumentiert
 - TestDebugging – Design wurde umgesetzt und grob dokumentiert
 - Productivity Tools – Warnings & Quickfixes
 - Design wurde umgesetzt und grob dokumentiert
 - Erste Tests umgesetzt
 - Konfigurationsmöglichkeiten über Extension Settings
 - Testdiscoverymodes
 - Automatic
 - Executables
 - Code
 - Legacy Code Warnings
 - Turn on or off
 - CppTools LanguageServer Mode
 - Background
 - Explicit
 - Linux Support
 - Analyse der nötigen Änderungen in Gang
 - Erste Punkte umgesetzt
 - Dokumentation wurde weitergeführt
 - Features Dokumentation
 - Vergleich zu cvevelop Extension
 - CMakeParser

Geplante Arbeiten:

- CUTE-Extension
 - Linux Kompatibilitätsänderungen abschliessen
 - Evaluieren der Refresh Intellisense for Workspace Funktionalität zur Verbesserung der cpptools Stabilität
 - Unit- / Integrationstests ausarbeiten (Testabdeckung von 80% angestrebt)
- Dokumentation
 - Features Dokumentation abschliessen
 - Architekturdokumentation abschliessen
 - Konfigurationsmöglichkeiten dokumentieren
- Workitems/Zeiterfassung nachführen um weiteres Vorgehen im Projekt zu planen

C.11 Week12

Vergangene Woche:

- CUTE-Extension
 - Testing
 - Implement UnitTests
 - Coverage over 80%
 - Adapt last architecture changes to make all componetns testable
 - Linux Support
 - All features available under Linux by now
 - Dokumentation wurde weitergeführt
 - Introduction Cleanup
 - Domain Analysis Cleanup
 - Management Documentation Finish
 - Requirement Analysis Cleanup
 - Risk Analysis Cleanup
 - Clean Decisions Documentation
 - First Level of C4 Architectural Documentation
 - Linting
 - All Files complying to AirBnB code style guidelines
 - Exceptions made regarding Static methods
 - Needed to allow mocking of all components

Geplante Arbeiten:

- CUTE-Extension
 - Finish of all the linting changes
 - Manual testing and potential bugfixing
 - Publish CUTE Extension on VS Code marketplace
- Dokumentation
 - Finish C4 architectural documentation
 - Finish Implmentation Documentation
 - Finish Quality Standards Documentation
 - Manual testcases
 - Tests for NFRs
 - Setup Youtube channel for video instructions and demonstrations
- Workitems/Zeiterfassung nachführen um weiteres Vorgehen im Projekt zu planen

C.12 Week13

Vergangene Woche:

- CUTE-Extension
 - Testing
 - Implement additionaly System Tests
 - Test Coverage
 - Analyse SonarQube
 - TestAnpassungen für Coverage
 - Anpassung Pipeline
- Dokumentation wurde weitergeführt
 - Architektur Dokumentation abgeschlossen
 - Implementations Dokumentation abgeschlossen
 - Feedback punkte analysiert
- Beta Version Publiziert
- Bug Fixing
- Stability Improvements

Geplante Arbeiten:

- Letzte Tests fixen
- Release in den nächsten Tagen
- Dokumentation abschliessen
 - Setup Youtube channel for video instructions and demonstrations
 - Finish Quality Standards Documentation
 - Manual testcases
 - Tests for NFRs
 - Überprüfen der Rückmeldungen und Verbesserungen
 - Kontrolllesen und Verbesserungen
- Workitems/Zeiterfassung nachführen um weiteres Vorgehen im Projekt zu planen

C.13 Week14

Vergangene Woche:

- CUTE-Extension
 - Testing
 - Get tests running in CI/CD pipeline
 - Get tests running under Linux
 - Test Coverage
 - Fix bug in coverage report
 - Test Anpassungen für Coverage
- Dokumentation wurde weitergeführt
 - Abstract erfasst
 - Plakat aufgeräumt
 - Punkte aus Feedback analysiert und angepasst
 - Manuelle Systemtests beschrieben
 - Document Testprotocol (NFR & Manuelle Systemtests)
 - Technologie Dokumentation abgeschlossen
 - Youtube channel für Instruktions- und Demonstrationsvideos aufgesetzt
 - QR Code auf Poster
 - Kontrolllesen und Verbesserungen
 - Kapitel Dev Workspace
 - Kapitel Dev Pipeline
 - Kapitel Analyzing Linting
 - Kapitel CMake Parser
 - Macro Versionen fertig beschrieben
 - Referenzen nachgeführt
- Beta Version Publiziert
- Bug Fixing
- Stability Improvements

Geplante Arbeiten:

- Dokumentation abschliessen
 - ☐ ◦ Metriken Einpflegen
 - ☒ ◦ Kapitel Video / Benutzeranleitungen
 - ☒ ◦ Entscheidung MSYS2
 - ☒ ◦ Macro Beschreibung
 - ☒ ◦ Management Summary
 - ☒ ◦ Schlussbericht
 - ☒ ◦ Persönliche Erfahrungen
 - ☐ ◦ Anhang & Verzeichnisse
 - ☐ ◦ Drucken 15.06.2022
 - ☒ ◦ Poster finalisieren und einschicken 13.06.2022
 - ☐ ◦ Videos aufnehmen und Veröffentlichen 16.06.2022
 - ☐ ◦ Alle Bilder beschriften und auf Lesbarkeit prüfen
 - ☐ ◦ Sitzungsprotokolle
- CUTE Extension
 - Version 1.0.0 im Maruketplace Veröffentlichen 16.06.2022