

SA

Googletest to CUTE Converter

University of Applied Sciences Rapperswil

HS 2018

Author: Sascha Gschwind & Renato Venzin
Advisor: Peter Sommerlad & Hansruedi Patzen
Industry Partner: Institute for Software (IFS) HSR

1. Abstract

Googletest is a widely used C++ testing framework. CUTE is a testing framework developed by Peter Sommerlad and integrated into the Cevelop IDE through the CUTE plugin. There should be an easy way for developers to switch from Googletest to CUTE, so that more projects test and hopefully switch to CUTE. This project aims to fill this gap and provide a plugin which helps convert Googletest projects into CUTE projects.

The goal was to create a refactoring that can be used to convert entire projects or at least provide a foundation on which a future project can build upon. The result is a refactoring able to convert projects consisting of a single source file. The most commonly used assertions are supported, with a few exceptions which do not have an equivalent assertion in CUTE. This project acts as a basis for converting a simple project. The project can be easily extended in order to support converting bigger projects.

2. Management Summary

The management summary contains a brief description of the project. It contains the initial position, the approach to the project, the results and a demonstration.

2.1. Initial position

Googletest is a widely used testing framework. CUTE is a testing framework developed by Peter Sommerlad, the framework is included in the CUTE plugin for Cevelop. Today there is no way to simply switch from Googletest to CUTE. If a developer wants to convert an entire project from Googletest to CUTE they would need to do lots of manual conversions. This should be automated using a refactoring, which should be the result of this project.

2.2. Approach, Technologies

Since we were completely new to plugin-development and the ILTIS framework, as well as quite unexperienced in regards to the Abstract Syntax Tree (AST), we had to work highly agile using Scrum constructs [Ken Schwaber, 2018] (Sprints, Backlog) since we could not plan much in advance.

Technologies

Technology-wise there was not much to decide, the CUTE plugin is written in Java [Oracle, 2018b], therefore our addition to it is written in Java as well. Since we wrote an Eclipse plugin it only made sense to use the Eclipse IDE for development [Eclipse, 2018a].

For coordination of work and meeting protocols we used gitlab, provided by the IFS [GitLab, 2018].

Continuous Integration & Deployment was handled already, since we are in the CUTE environment [IFS, 2018b].

2.3. Results

A plugin which successfully converts a project consisting of a single file containing one or more testsuites (or testcases in Googletest). This allows for an easy conversion of a simple project. The entire process is automated, from the creation of the headers and suitefiles of CUTE to the deletion of the old Googletest files. There are still some constructs that are not yet supported in this plugin. They will be added in future work and will be described in detail in the technical report.

2.4. Demonstration

This section is meant to illustrate the refactoring of a simple sample project. We chose a project with two testsuites (TestCases in Googletest) and two namespaces, to show most of the features already implemented.

2.4.1. Before

```
1 #include "gtest/gtest.h"
2
3 namespace {
4     bool foo2() {
5         return true;
6     }
7 }
8
9 namespace bar {
10    bool foo() {
11        return false;
12    }
13 }
14
15 TEST(TestCaseName, TestName) {
16     ASSERT_TRUE(foo2());
17     std::string str1;
18 }
19
20 TEST(TestCaseName2, TestName2) {
21     ASSERT_FALSE(bar::foo());
22     std::string str2;
23 }
24
25 int main(int argc, char **argv) {
26     ::testing::InitGoogleTest(&argc, argv);
27     return RUN_ALL_TESTS();
28 }
29
```

```
<terminated> (exit value: 0) ConverterTestProject.exe [C/C++ Application] C:\Users\c
[=====] Running 2 tests from 2 test cases.
[-----] Global test environment set-up.
[-----] 1 test from TestCaseName
[ RUN    ] TestCaseName.TestName
[      OK ] TestCaseName.TestName (0 ms)
[-----] 1 test from TestCaseName (0 ms total)

[-----] 1 test from TestCaseName2
[ RUN    ] TestCaseName2.TestName2
[      OK ] TestCaseName2.TestName2 (0 ms)
[-----] 1 test from TestCaseName2 (0 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 2 test cases ran. (1 ms total)
[ PASSED ] 2 tests.
```

Figure 2.1.: Googletest example project

This depicts a very simple GoogleTest project consisting of a single file. This project contains two testcases (suites in CUTE), two tests and two namespaces, a named and an anonymous namespace.

2.4.2. Starting the refactoring

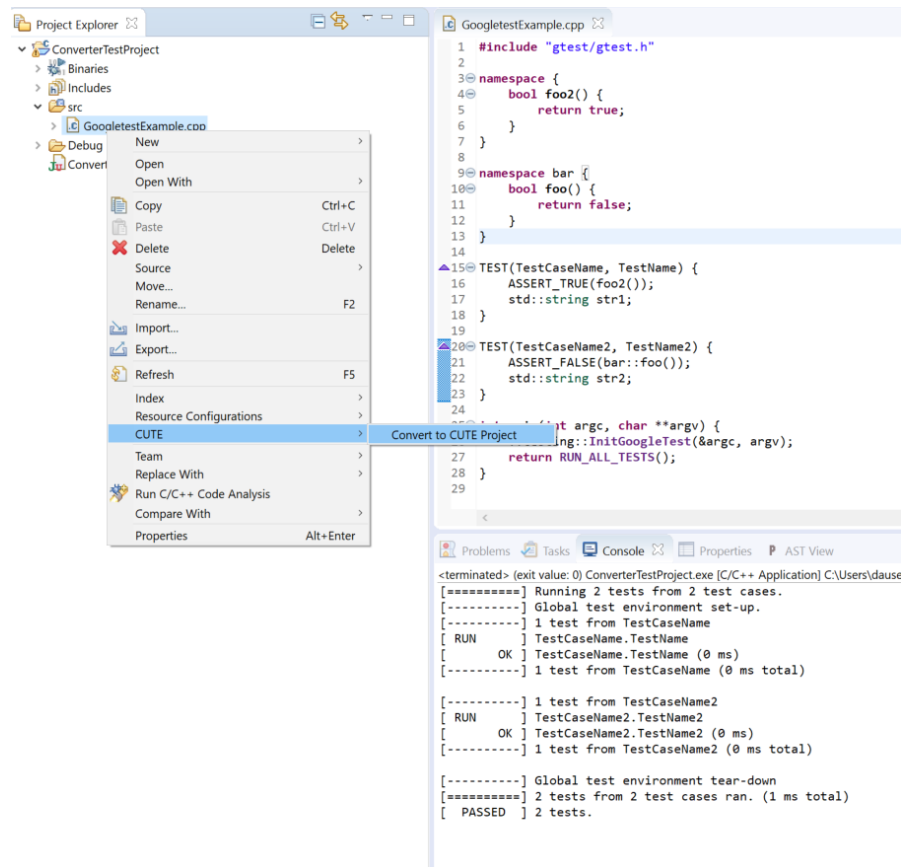


Figure 2.2.: Where the refactoring can be found

The refactoring can be found in the CUTE menu when right-clicking on a C++ file in the project explorer. The selected file is the one which gets converted.

2.4.3. Refactoring wizard

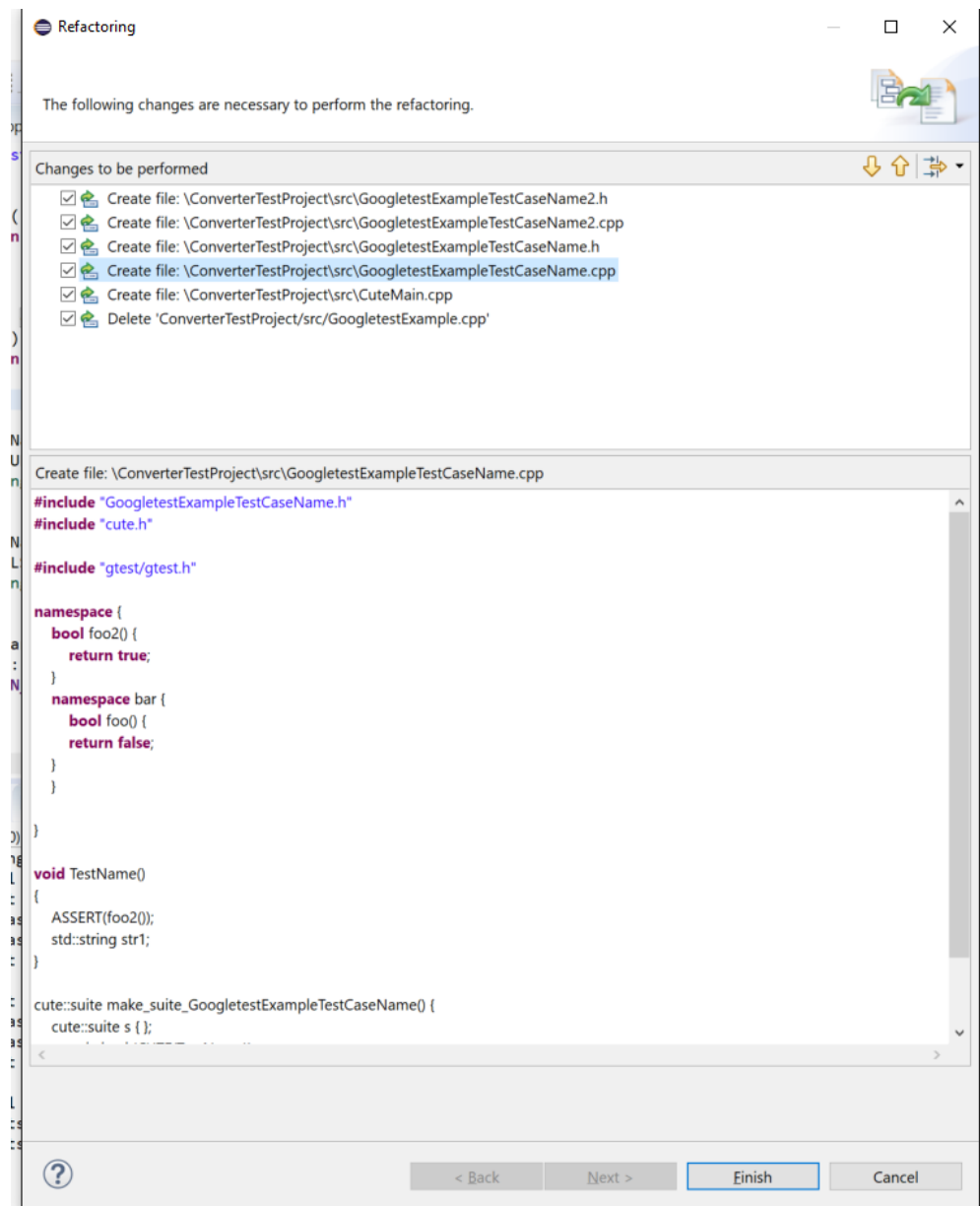


Figure 2.3.: Refactoring Wizard showing the changes

In this wizard all the changes are visualized. Here individual changes can be deselected and the changes can be viewed before applying them.

2.4.4. Result

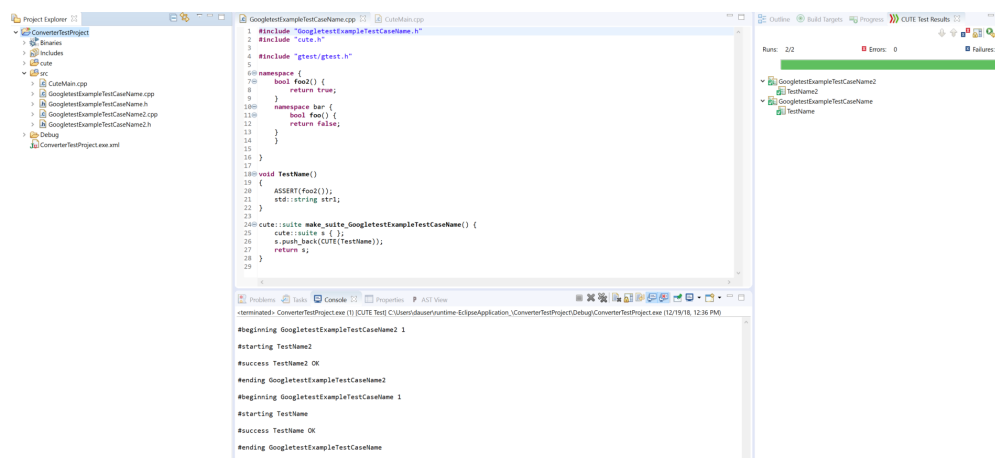


Figure 2.4.: The converted project which now uses the CUTE plugin

After the refactoring concludes, the CUTE files get created and the no longer needed files get removed. After adding the CUTE nature, the project can be built and executed as a CUTE test suite.

2.5. Outlook

These are the constructs that should be implemented, if and when the plugin is extended:

- Projects with several testfiles will be the first update that is needed. For more information about this, see chapter "Notes for updating the plugin" 11.1
- The remaining unsupported Asserts should be implemented where possible (e.g. where an equivalent or similar CUTE Assert exists). Otherwise a workaround should be found (for example when a project contains Deathtests or case equal comparison of strings)
- TEST_F testcases can be supported as well but would need additional logic so that they are handled correctly.

Contents

1. Abstract	
2. Management Summary	
2.1. Initial position	
2.2. Approach, Technologies	
2.3. Results	
2.4. Demonstration	
2.4.1. Before	
2.4.2. Starting the refactoring	
2.4.3. Refactoring wizard	
2.4.4. Result	i
2.5. Outlook	i
3. Assignment	2
3.1. Context	2
3.2. Problem	2
3.3. Task	2
3.4. Concrete task	3
3.5. Specifics	3
3.5.1. Existing infrastructure	3
3.5.2. Omittable documentation	4
3.6. Expected results	4
3.7. Appointments	4
3.8. Supervision	5
4. Introduction & Overview	6
4.1. Problem scope	6
4.2. Introduction	6
4.3. Preparation	8
4.4. Motivation	8
5. Projectplan	9
5.1. Project overview	9
5.1.1. General conditions	9

5.1.2.	Project organisation	9
5.1.3.	Time planning	10
5.1.4.	Project phases	10
5.1.5.	Milestones	11
5.2.	Risk management	12
5.3.	Work packages	12
5.4.	Infrastructure	12
5.5.	Quality assurance	13
5.5.1.	Process	13
5.5.2.	Sprint planning	13
5.5.3.	Meeting protocols	14
5.5.4.	Code quality	14
5.5.5.	Code style guidelines	14
5.5.6.	Continuous Integration	14
5.5.7.	Error tracking	15
5.5.8.	Testing	15
6.	Analysis	16
6.1.	Basic AST Analysis	16
6.1.1.	Variables	17
6.1.2.	Function without parameters	18
6.1.3.	Function with parameters	19
6.1.4.	Function with multiple statements	20
6.2.	Googletest to CUTE	21
6.2.1.	Documentation	21
6.2.2.	Open Source Projects using Googletest	21
6.2.3.	Mappings	22
6.2.4.	Test Runner	26
6.2.5.	Advanced concepts	27
6.2.6.	Examples	29
7.	Design	31
7.1.	SingleSuite Case	31
7.2.	MultiSuite Case	31
8.	Implementation	32
8.1.	Markers	32
8.2.	Writing the AST to a new file	33
8.3.	Nontest constructs	34
8.4.	Constructs which block the refactoring	34
8.5.	Testing the Refactoring	35

8.6. Deletion of Googletest files	36
9. Result	37
9.1. Reached	37
9.2. Not reached	37
9.2.1. Reasoning	37
10. Problems	39
10.1. Solved	39
10.1.1. Indexer running permanently, blocking the refactoring	39
10.1.2. Named Namespaces lead to redefinition	39
10.2. Unsolved	39
10.2.1. Projects that include several testing files	39
10.2.2. Same definition in anonymous namespace and outside of it	40
11. Conclusion	41
11.1. Notes for updating the plugin	41
11.1.1. Multifile Case	41
11.1.2. TEST_F tests	41
11.1.3. Remaining assertions and other Googletest constructs	42
11.1.4. More defensive programming	42
11.1.5. Remove namespace duplication	42
11.2. Time analysis	42
11.2.1. Overall time per label	43
11.2.2. Comparison of estimates and actual time	43
11.2.3. Comparison between teammates	45
12. Glossary	47
Appendices	51
A. Usage manual	52
B. Demonstration	60

3. Assignment

In this chapter the assignment will be defined. The chapter contains the context, problem description, tasks, expected results, appointments and the supervision of this project.

3.1. Context

Googletest is a widely used testing framework for C++. The IFS has developed their own testing plugin called CUTE. To help developers using Googletest to convert their tests to CUTE a plugin for Cevelop (IDE for C++ development based on Eclipse) should be developed. This gives developers using Googletest an additional motivation to migrate to CUTE. Another use for it is at conventions to show how easy it is to migrate from Googletest to CUTE.

3.2. Problem

The Googletest framework uses some bad practices when it comes to testing. For developers using Googletest and wanting to migrate to CUTE it should be possible to do this without rewriting all the tests manually. This is where the plugin comes into play. It converts the most common Googletest features to the CUTE syntax.

3.3. Task

The students doing this SA have to write an Eclipse plugin which migrates Googletest tests to CUTE tests. The conversions need to be tested with unit tests to confirm they are working.

3.4. Concrete task

The following list contains all the necessary tasks for the students:

- Familiarize themselves with the Eclipse plugin development.
- Familiarize themselves with the Googletest framework.
- Familiarize themselves with the CUTE plugin.
- Define a list of features that can be converted using the plugin. These conversions will be the core scope that needs to be developed by the students, hence it needs to be documented and approved.
- Integrate an extension to the CUTE project which does the Googletest to CUTE conversions defined in the list above.
- Prepare the weekly meetings with a Wiki page describing the tasks done / open and questions that need to be answered.
- Write a document describing the scope. This document needs to be approved by the supervisors.
- Write a final report describing the project in its final state including an abstract and a management summary (see the SA documents from the HSR for more details).
- Create a poster which gives an overview of the project (see the SA documents from the HSR for more details).
- Demonstrate the functionality of the implementation on an existing open-source project.

3.5. Specifics

3.5.1. Existing infrastructure

The students doing this SA are extending the already existing CUTE plugin. CI is already in place and does not need to be set up.

3.5.2. Omittable documentation

The following list describes the things that can be omitted:

- No architecture needs to be documented because the CUTE project is already documented. Only how the extension is integrated into the existing project should be documented.

3.6. Expected results

The expected results are the following:

1. Analysis document describing which Googletest functions can be converted to CUTE
2. CUTE extension that lets the developer convert existing Googletest tests to CUTE.
3. A final report including the problem description, an abstract, a management summary, a technical report, the time analysis and what goals have been reached.
4. A poster giving an overview of the project.

3.7. Appointments

Appointment	Description
17.09.2018	Begin SA
18.12.2018	Upload abstract to the online tool https://abstract.hsr.ch
21.12.2018 until 12:00	Submission of the final report.
21.12.2018	Upload the documents to archiv-i.hsr.ch
Every Wednesday 15:30	Weekly meeting with the supervisors

Table 3.1.: Appointments

3.8. Supervision

Peter Sommerlad The main supervisor. He will be in most of the meetings (if he has time to attend them) and will have the final word on most decisions.

Hansruedi Patzen The second supervisor. He will be in all the weekly meetings and is the main person to get help from and ask questions during the SA.

Additional help:

Thomas Corbat Main contact point for C++ related questions. He will help if there are questions and/or problems related to C++ specifics.

AnneMarie O'Neill Language advisor reviewing our documentation in terms grammatic and orthography.

4. Introduction & Overview

This chapter should give the reader an overview of the project. The reader should know the problem scope of this project and how the solution was approached.

4.1. Problem scope

Write an Eclipse plugin which refactors Googletest files into CUTE compatible files. The plugin shall convert the assertions, which are similar but not identical, in a way to keep the functionality of the tests as before. The plugin also needs to convert the Googletest TEST macros into ordinary functions. In order for this to work, the plugin needs to be able to handle the test registration. The refactored files should follow the cute guidelines (one header & implementation file per suite, test-registration in suite-file, one main file which runs all tests).

4.2. Introduction

The goal of this project was to convert Googletests [Google, 2018b, Google, 2018a] to CUTE tests [IFS, 2018b] using a refactoring. We added a plugin to the already existing CUTE project, which is a part of the Cevelop IDE developed by the Institute for Software at the HSR. [IFS, 2018a]

Using the ILTIS framework [IFS, 2016] developed by the Institute for Software and the Eclipse CDT core features, we had to manipulate the abstract syntax tree and save the changes in a refactoring.

There were multiple hurdles we had to overcome in order for us to be able to convert an existing Googletest based file. First we needed to get the abstract syntax tree using the frameworks available. This was quite easy since the ILTIS framework already provided a function for this.

The second task was to find the nodes in the abstract syntax tree using a visitor. [Erich Gamma, 1997] This was quite tricky because we had to identify what characteristics were unique in the nodes created by the Googletest macros. [Stroustrup, 2015] We decided to use multiple visitors to find different types we needed to convert. We used a visitor for the TEST macros itself, a visitor for the ASSERT and EXPECT statements a visitor for the namespaces and a visitor

for other definitions like variables, classes, structs, enums and functions. We also needed to capture all the preprocessor statements because we needed to include them in the new files as well. To help identify the nodes and to get the parameters we wrote a parser using the function `getSyntax()`. Because of a bug in the CDT we had to split the parser because the TEST macro node could not be parsed using the framework. Since the usage of the TEST macros is very restricted we were able to implement the parsing of these nodes using regex. Everything else we were able to parse using an actual parser mechanism.

Having collected all the nodes that need to be manipulated, we refactored them to the CUTE syntax. Because the CUTE plugin differentiates between a single testsuite project and one with multiple test suites we had to differentiate the refactoring too. For the single testsuite we manipulated the file abstract syntax tree of the file directly using the ASTRewrite [Oracle, 2018a]. In the other case, when multiple test suites were present, we had to copy the abstract syntax tree because the original one is always frozen. On this copied abstract syntax tree we were able to manipulate nodes directly. For example for the TEST macros we had to change the Declarator to one complying with the CUTE syntax. Since those changes are not visible in the abstract syntax tree unless the tree is written and the copy still targets the original file we had to use an Eclipse core class which is not open API, the ASTWriter [CCTLSU, 2018]. With this ASTWriter we were able to write down manipulated nodes to a string which could then be used to insert them at the right place in a template file. We then collected all these changes based on the template file and the original tree in a string which was used to create a file change that got added to a ModificationCollector. This made it possible that we get a preview of the new file during the refactoring and undo the changes if needed. In case of multiple test suites we created new files, one for each suite, and a new main file which runs the test suites.

The old file was still lying around though. Because there can never be two main functions and it is bad practice to leave dead code lying around, we needed to remove the old file using a DeleteFileChange that also got added to the ModificationCollector.

Because it was known from the start that we would not be able to implement every little detail of these testing frameworks during the SA period we had to make certain checks to prevent failing refactorings. For this we used so called Checkers [Checkerframework, 2018]. We implemented two visitors which report problems detected in the file. The checkers register these problems and show the user a marker inside his IDE. Before we start the refactoring we check if any of our problems are present and if so we abort the refactoring displaying an error message.

4.3. Preparation

To prepare ourselves for this SA we completed the tutorials from Vogella [Vogella, 2018]. These tutorials helped us understand how the plugin development works in Eclipse and what parts are needed to achieve our goal. Additionally, we refreshed our C++ skills with a book from Mr. Stroustrup [Stroustrup, 2015] and the slides from the C++ module we already visited. For the abstract syntax tree manipulation we read through an online article [Thomas Kuhn, 2006] published by the Eclipse foundation to get some idea of how AST manipulation works. We also read through the ILTIS documentation [IFS, 2016] to get an overview of the framework that we were using a lot during the project.

4.4. Motivation

Googletest is widely used and has many supporters, nevertheless there are some aspects of Googletest which are subject to discussion regarding best practices. For example, non-fatal assertions (EXPECT) can be quite useful when testing for several inputs or aspects of a function, but when looking at unit- or even microtesting best-practices, every test should only test for one specific case. When using nonfatal assertions one is quickly tempted to test for as much as possible in a single test, breaking exactly this. [Microsoft, 2018c]

Another point which might be up for discussion is the way Googletest handles test-registration, because tests are registered automatically through the macros used. This may be quite simple and easy to use, but having this functionality locked behind macros by using static initializers may sometimes lead to confusion or even problems. [Denim, 2012]

Developers should have the option to try out different testing environments, without the need to change code for hours. With our plugin, developers get the chance to try out CUTE and see if it might fit their project and if not, simply revert the changes.

Lastly, when you try to debug Googletest code, be it because of an unexpected behaviour or out of curiosity, you will quickly find yourself lost or confused, as the code is often unorganized and some functions seem to be spread over several places in order to obfuscate some implementations. Additionally, there are many code smells, making the code unpleasant to read and follow. This includes but is not limited to: Large class and deodorant comments.

5. Projectplan

The projectplan chapter contains some general information of the project as well as the milestones defined with a timeline. It also gives an overview of the steps we implemented to assure the quality of the work done.

5.1. Project overview

5.1.1. General conditions

Official project duration	14 weeks
Project members	2
Hours per week per person	17.14
Total hours	480
Official project start	19. September 2018
Official project end	21. December 2018

Table 5.1.: General conditions

5.1.2. Project organisation

We used a flat organisation for this project since we are only two members. Both of us had certain responsibilities. The responsible person had to make sure that the quality of his responsibilities is at least acceptable and make suggestions if it is not. Although we split the responsibilities we helped each other out with them. Everyone was part of the whole process.

Person	Responsibilities
Sascha Gschwind	Code quality, Testing, Implementation
Renato Venzin	Time analysis, Documentation, Implementation

Table 5.2.: Responsibilities

5.1.3. Time planning

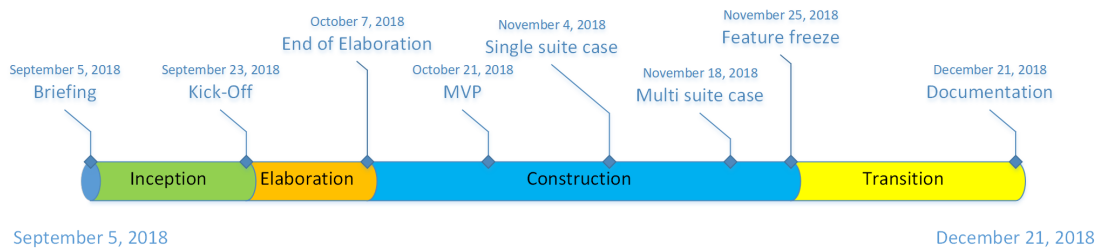


Figure 5.1.: Time planning

5.1.4. Project phases

We decided to use the RUP project phases which were recommended to us during the software engineering modules.

Inception During this phase we gather knowledge about the frameworks and set up the development environment.

Elaboration During this phase we elaborate on the gathered knowledge and expand on it. We make sure the project and documentation is set up correctly and set up the time tracking.

Construction In this phase the actual code for the project is written. We work on implementing the features and assure the code quality.

Transition The last phase of the project. In this phase we document our work and finish the project.

5.1.5. Milestones

Nr	Title	Date	Description
M0	Briefing	05.9.2018	Project briefing with Mr. Sommerlad. Gather information about the project and what we can do to prepare ourselves.
M1	Kick-Off	23.9.2018	Kick-Off Meeting with Mr. Sommerlad and Mr. Patzen. Official start of the project.
M2	End of Elaboration	07.10.2018	All the open questions are cleared. The development environment is set up and the tools are known and installed.
M3	MVP	21.10.2018	A minimum viable product (MVP) can be presented. The product should be able to convert one simple case.
M4	Single suite case	04.11.2018	The single suite case can be refactored with most of the specialties already handled correctly.
M5	Multi suite case	18.11.2018	The multi suite case can be refactored with most of the specialties already handled correctly.
M6	Feature freeze	25.11.2018	There will be no additional features added after this point. Most of the code is refactored and in a good state.
M7	Documentation	21.12.2018	All of the documentation is done, uploaded and handed out. The code is completely refactored and in a good state.

Table 5.3.: Responsibilities

5.2. Risk management

The biggest risk during the project is illness. All of the other risk factors can be neglected. The infrastructure runs on the servers from the HSR and should be backed up by them automatically.

If a team member gets sick for several weeks there is not much we can do. We will try to finish the work as early as possible so the other person can compensate in case of illness.

5.3. Work packages

We will use Gitlab [GitLab, 2018] as our work package container. You can create milestones and issue as well as document the time estimated and spent on those issues.

Our work packages will be grouped so we can later use these groups for the time analysis. We decided to use the following groups:

- Priority labels (High, Medium-high, Medium, Medium-low, Low)
- Phase labels (Inception, Elaboration, Construction, Transition)
- Documentation
- Environment Familiarization
- Implementation
- Meeting & Preparation
- QA
- Toolchain

5.4. Infrastructure

Most of the infrastructure decisions were already given by the IFS beforehand. We will be using LaTeX [LaTeX, 2018] for the documentation, Java [Oracle, 2018b] as our programming language, Gitlab [GitLab, 2018] for the work packages, as our git repository and for the continuous integration. The IDE we will be using is Eclipse Plugin Development Environment (PDE) [Eclipse, 2018b]. We decided to use Excel [Microsoft, 2018a] for the time analysis and Visio [Microsoft, 2018b] for other graphics needed. Our Git Tool will be Sourcetree [Atlassian, 2018] and we will

work on the LaTeX documents with either TeXstudio [Benito van der Zander, 2018] or Visual Studio Code [Microsoft, 2018d].

5.5. Quality assurance

5.5.1. Process

To assure the quality of our process we work one and a half days together and about half a day alone on the project. Even though we work together most of the time we decided to further define our process.

- Meetings
 1. Sprint planning meeting every Wednesday
 2. Meeting with our advisors afterwards also every Wednesday
- When working on an issue:
 1. Assign the issue to yourself
 2. Pull the source code from the git repository
 3. Create a branch for your work
 4. Commit regularly onto the branch
 5. Make a pull request (which triggers the continuous integration) when you are done with the issue.
 6. Record the time spent
 7. Close the issue once the continuous integration ran successfully and the work is reviewed by the other team member.

5.5.2. Sprint planning

At the end of every sprint we will discuss what went good and what went bad. We will discuss the next step and plan the next sprint accordingly. We will send the wiki with the current sprint and the outlook to the next sprint to the advisors including open questions that need to be answered during the meeting.

5.5.3. Meeting protocols

For every meeting there will be a protocol which will be uploaded to the Wiki in GitLab. The wiki will always have a certain structure which might change slightly depending on what is needed for the meeting:

- Participants
- Open tasks after last meeting
- Relevant documents
- Current sprint
- Next sprint
- Open questions

5.5.4. Code quality

To assure a certain code quality we try to review our code as much as possible. Every pull request has to be reviewed before it will be merged into the master branch. Additionally we try to get code reviews from our advisors which will happen sporadically.

5.5.5. Code style guidelines

To make sure our code confirms with the guidelines from the Institute for Software we will be using their code style guidelines including the code styling template which will automatically format the code to confirm with the guidelines.

5.5.6. Continuous Integration

Since we have a fork of the CUTE plugin the continuous integration is already set up for us on the repository. Every pull request gets checked by the continuous integration so we can make sure the master branch is always working correctly.

5.5.7. Error tracking

Upcoming errors will be recorded in our sprint planning tool on GitLab and handled with the priority they need. This way we have all our issues in the same place and have a good grip of them.

5.5.8. Testing

We wanted our tests to be long living and secure to changes. Since it does not make much sense to write many unit tests for our project we decided to use integration tests only with sporadic manual tests.

If we find some bugs in our software during the system tests we will write an integration test for the scenario to make sure the bug will stay fixed.

During development we try to write tests as early as possible so they can help us refactor the code later.

6. Analysis

When beginning this project, both of us were mostly or even completely new to plugin development as well as working with an AST. So we spent the first three weeks setting up our toolchain, reading tutorials and analyzing both Googletest and the CUTE plugin. This includes analyzing the AST of a simple class and the comparison between the two AST representations.

6.1. Basic AST Analysis

We wanted to analyze some common AST constructs to see how the AST of these code constructs looks like. Our goal was to better understand the AST and being able to predict AST constructs seeing code and vice versa. With this knowledge we expected to better understand what our refactoring has to do and how we could approach it.

6.1.1. Variables

We wanted to see how a simple variable declaration and assignment looks like in the AST. We expected this to be one of the simplest constructs and wanted to find out how we could approach finding these variables in the AST.

Code

```
1 int x = 13;
```

Listing 6.1: Variable declaration

AST

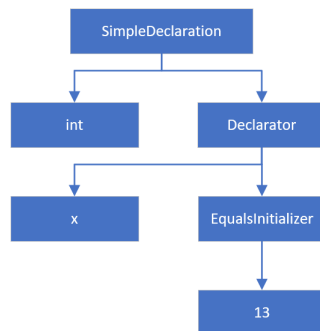


Figure 6.1.: Variable AST

Description

Code	AST Part
<code>int x = 13;</code>	A SimpleDeclaration will be made. The return type will be the first child of this element.
<code>x = 13</code>	A Declarator will be made. The variable name will be the first child of this element.
<code>= 13</code>	This is an EqualsInitializer with the value as its only child.

Table 6.1.: AST variable analysis

6.1.2. Function without parameters

Next we were interested how the simplest function possible looks like in the AST. We decided to take a look at a `void foo()` function and inspect the resulting AST.

Code

```
1 void foo() {  
2     return;  
3 }
```

Listing 6.2: Function without parameters

AST

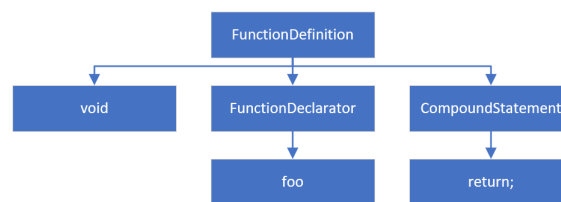


Figure 6.2.: Function without parameters AST

Description

Code	AST Part
Whole function	A FunctionDefinition will be made. The return type will be the first child of this element.
<code>foo()</code>	A FunctionDeclarator will be made. The function name will be the first child of this element.
Function body	A CompoundStatement will be made for the part inside the brackets holding the body of the function as its children.

Table 6.2.: AST function without parameters analysis

6.1.3. Function with parameters

The next logical step was to take a look at a function which also has some parameters. We were especially interested in the representation of the parameters in the AST because we needed them for the conversion.

Code

```
1 void foo(int x) {  
2     return;  
3 }
```

Listing 6.3: Function with parameters

AST

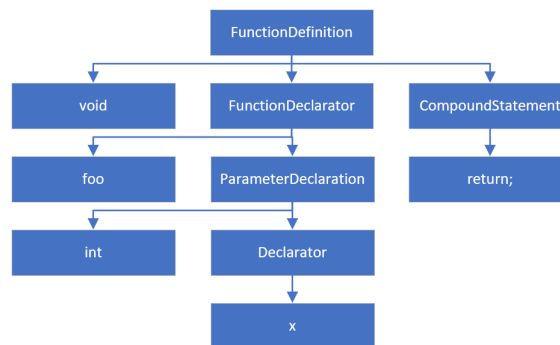


Figure 6.3.: Function with parameters AST

Description

Code	AST Part
Parameter	For each of the parameters there will be a ParameterDeclaration with its type as the first child. Inside the ParameterDeclaration there will always be a Declarator holding the variable name.

Table 6.3.: AST function with parameter analysis

6.1.4. Function with multiple statements

Next we wanted to see how a function which has multiple statement looks like in the AST. We wanted to know if these statements were unrelated nodes and if they were children of the body of the function. This was important because if our assumptions would hold true we would be able to convert the statements itself without having to know to which function they belong.

Code

```
1 void swap(int& x, int& y) {  
2     int temp = x;  
3     x = y;  
4     y = temp;  
5 }
```

Listing 6.4: Function with multiple statements

AST

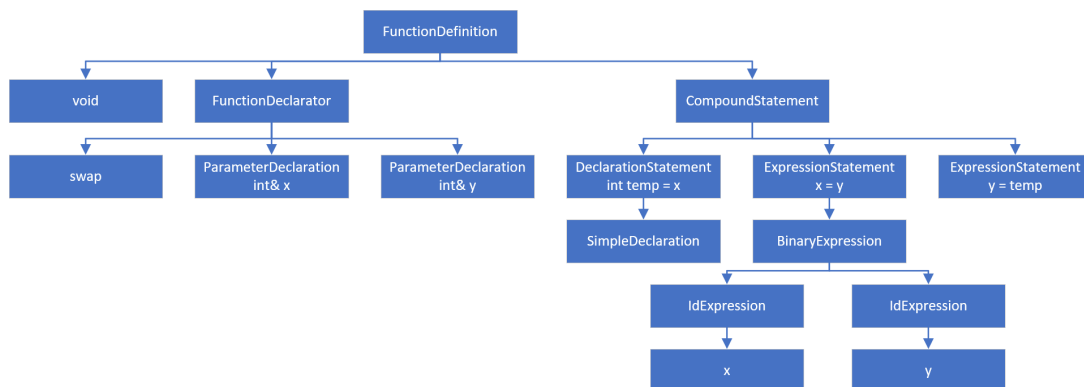


Figure 6.4.: Function with multiple statements AST

Description

Code	AST Part
Body	In this case we have a single DeclarationStatement (<code>int temp = x;</code>) which holds a SimpleDeclaration (which we already described in the variables section) and two ExpressionStatement (<code>x = y;</code> and <code>y = temp;</code>) nodes with a BinaryExpression which assign some value to a variable. There are many different types of statement nodes which can be seen in the Eclipse documentation [Eclipse, 2018a].

Table 6.4.: AST function with multiple statements analysis

6.2. Googletest to CUTE

This section shows some Open Source projects using Googletest and will give the reader an overview of the Googletest functionality with the corresponding functionality in CUTE. It will also describe some key concepts in these test environments. This section is designed to help the reader understand the differences.

6.2.1. Documentation

Links to the documentation of the Googletest project and the CUTE plugin. The information provided here was compiled using these resources.

Googletest Documentation

Googletest Documentation [Google, 2018b]

Googletest advanced Documentation [Google, 2018a]

CUTE Documentation

CUTE Documentation [IFS, 2018b]

6.2.2. Open Source Projects using Googletest

The following Open Source Projects are currently using Googletest and could be possible customers to this plugin. Some of them will be way too big to be a target customer at first.

Name	Link to project
Chromium project	http://www.chromium.org/
LLVM compiler	http://llvm.org/
Protocol buffers	https://github.com/google/protobuf
OpenCV	http://opencv.org/
tiny-dnn	https://github.com/tiny-dnn/tiny-dnn
Talcoin	https://github.com/talt5/Talcoin
Uni10	https://gitlab.com/uni10/uni10/tree/master
Netbackup	https://github.com/tetofonta/NetBackup
Jyocoin	https://github.com/kamuluprashanth/jyocoin

Table 6.5.: Open source projects using Googletest

6.2.3. Mappings

Assertions

List the assertions available in Googletest and (when applicable) the corresponding CUTE assertion to which they can be mapped.

Notable Behaviours & Differences between Googletest and CUTE

List differences between the two testing environments which need to be considered when rewriting projects.

Fatal assertions (ASSERT) Fatal assertions are the usual assertions you see in every language. In Googletest these assertions begin with `ASSERT_`. If one of these assertions fails, the test will fail also. The first assertion that fails triggers the test failure hence the following assertions will not be checked anymore.

Non fatal assertions (EXPECT) Non fatal assertions are a concept used in Googletest for assertions that do not force the test to fail immediately. The test will continue, until the first fatal assertion fails or the end of the test is reached. Afterwards, if a non fatal assertion failed the test will fail too. This concept is not supported in CUTE, therefore these statements be converted to ordinary fatal assertions.

String comparison In Googletest there are multiple assertions targeted at string comparison. It would still be possible though to compare strings using the `ASSERT_EQ(actual, expected)` assertion which would compare the strings using their address. Since you should never compare strings using the address, this behaviour will not be supported after the conversion to CUTE.

User Defined assertions Googletest allows users to define their own assertions. These could be hard to find and need to be analyzed further, take a look at the reference projects to get a grasp of them and if these are even used in actual projects.

Naming in Documentation Googletest has a different naming for testcases and suites than other testing frameworks. This only applies to their documentation but should still be considered when reading through it.

Meaning	Googletest Term	ISTQB Term
exercise a particular program path with specific input values and verify the results	TEST()	Test Case
A set of several tests related to one component	TestCase	TestSuite

Table 6.6.: Naming in Googletest

Naming of tests Googletest has a different way of naming their tests: `TEST(TestCaseName, TestName){}` where CUTE uses: `TestName(args)`

Stream message into Assertion Googletest allows the programmer to stream a message into the assertion. Anything that can be streamed to an ostream can also be streamed to an assertion-macro.

Example: `ASSERT_EQ(x.size(), y.size()) << "vectors x and y are of unequal length";`
(wide strings are translated to UTF-8)

Invoking Tests Googletest implicitly registers their tests, `RUN_ALL_TESTS` runs *all tests* in the unit, including those from different source files.

Mapping overview

Basic assertions

These are the basic concepts from the Googletest documentation that might be interesting in the scope of the project. Most of them should be implemented in the Googletest to CUTE converter.

Googletest	CUTE
ASSERT_TRUE(condition)	ASSERT(condition)
ASSERT_FALSE(condition)	ASSERT(!condition)
ASSERT_STREQ(str1, str2)	ASSERT_EQUAL(str1, str2)
ASSERT_STRNE(str1, str2)	ASSERT_NOT_EQUAL_TO(str1, str2)
ASSERT_EQ(actual, expected)	ASSERT_EQUAL(expected, actual)
ASSERT_NE(val1, val2)	ASSERT_NOT_EQUAL_TO(left, right)
ASSERT_LT(val1, val2)	ASSERT_LESS(left, right)
ASSERT_GT(val1, val2)	ASSERT_GREATER(left, right)
ASSERT_LE(val1, val2)	ASSERT_LESS_EQUAL(left, right)
ASSERT_GE(val1, val2)	ASSERT_GREATER_EQUAL(left, right)
ASSERT_STRCASEEQ(str1, str2)	No equivalent
ASSERT_STRCASENE(str1, str2)	No equivalent
ASSERT_THROW(stm, exc_type)	ASSERT_THROWS(code, exc)
<i>ASSERT_FLOAT_EQ(val1, val2)</i>	ASSERT_EQUAL(val1, val2)
<i>ASSERT_DOUBLE_EQ(val1, val2)</i>	ASSERT_EQUAL(val1, val2)
ASSERT_NEAR(val1, val2, abs_err)	ASSERT_EQUAL_DELTA(exp, act, delta)

Table 6.7.: Basic mapping overview

Assertions written in *italic* are deprecated but should still be handled with the converter since they could still exist.

Advanced assertions

These are some concepts from the Googletest advanced documentation that might be interesting in the scope of the project. The concepts listed here are more advanced and might not fit into the scope or might not even be used in actual projects. This is why they are low-priority and will only be implemented if necessary and if there is enough time.

Googletest	CUTE
ASSERT_ANY_THROW(statement)	No equivalent
ASSERT_NO_THROW(statement)	No equivalent
ASSERT_PRED1(pred, val1)	ASSERT(pred(val1))
ASSERT_PRED2(pred, val1, val2)	ASSERT(pred(val1, val2))
ASSERT_PRED_FORMAT1(pred, val1)	ASSERT(pred(val1))
ASSERT_PRED_FORMAT2(pred, val1, val2)	ASSERT(pred(val1, val2))
ASSERT_THAT(value, matcher)	No equivalent
ASSERT_HRESULT_SUCCEEDED(exp)	No equivalent
ASSERT_HRESULT_FAILED(exp)	No equivalent

Table 6.8.: Advanced mapping overview

Key mapping differences

`EXPECT_X` All the non-fatal assertions use the same syntax as the fatal ones with the same suffix but instead of `ASSERT` the keyword `EXPECT` is used. Non-fatal assertions are not supported in CUTE and will consequently be converted to fatal assertions (see above).

`ASSERT_FALSE(condition)` Since there is no direct mapping possible the variable needs to be negated in the resulting CUTE assertion.

`ASSERT_EQ(actual, expected)` In the CUTE test environment the variables are swapped. Most testing frameworks use the parameter order (expected, actual), hence the variables need to be swapped here.

`ASSERT_STRCASEEQ(str1, str2)` There is currently no direct support for case insensitive string comparison. There are many possibilities how to approach this conversion. One would be to use a macro which compares the strings using C++'s `lexicographical_compare`. Another possibility would be to add this kind of test to the CUTE plugin making a direct mapping possible.

`ASSERT_STRCASENE(str1, str2)` Same as `ASSERT_STRCASEEQ(str1, str2)`, and as a result it should be handled similar for consistency reasons.

6.2.4. Test Runner

Googletest Test Runner

In Googletest the test runner consists of a single function called `RUN_ALL_TESTS()` where tests register themselves through the `TEST` and `TEST_F` macro.

CUTE Test Runner

In CUTE the test runner also consists of a single method `runAllTests(int argc, char const * argv[])` but the tests need to manually be added. For this the method `s.push_back(CUTE(testMethodName))` is used. If tests are spread across multiple files, all header files of these tests have to be included in the `Test.cpp` file (or wherever your `runAllTests` method is) so you can register the test methods.

6.2.5. Advanced concepts

This section contains advanced concepts available in Googletest. These are mostly concepts not available in CUTE. For further information about any of those constructs, check the Googletest documentations [Google, 2018b, Google, 2018a]

TEST_F TEST_F is a test-macro which is meant for cases where a test fixture makes sense, as in several tests need the same type of object. It takes the same arguments as ordinary TEST macros, with the only difference, that the TestCase needs to be the name of a test fixture class (derived from `::testing::Test`). The macro creates a fresh fixture for each TEST_F test, initialize it by calling its `SetUp()`, run the test, call the classes `TearDown()` and then delete the fixture. Each test has its own test fixture, if this were not the case, it would violate the basic principle, that all tests run independently from each other.

AssertionResult using `::testing::AssertionSuccess / ::testing::AssertionFailure` one can add information to Assertion messages by returning those instead of bool values.

Type assertions using `::testing::StaticAssertTypeEq<T1, T2>()`; one can test if two types are the same. This will do nothing if they match, otherwise the function call will fail to compile and the compiler error message will (likely) show the actual values of the Types. mainly useful inside template code.

Assertion placement Assertions that create a fatal failure (FAIL & ASSERT) can only be placed in void-returning functions.

Death test Googletest has some macros that address death tests. These are assertions that check if a certain statement causes the process to die. These are:

`ASSERT_DEATH(statement, regex)`

`ASSERT_DEATH_IF_SUPPORTED(statement, regex)` and

`ASSERT_EXIT(statement, predicate, regex)`

with their corresponding EXPECT_ counterparts. **statement** is a statement that is expected to cause the process to die, **predicate** is a function or function object that evaluates an integer exit status, and **regex** is a (Perl) regular expression that the stderr output of statement is expected to match.

Assertions in sub-routines Googletest has a concept for using assertions in sub-routines. When a fatal assertion in a sub-routine fails, the sub-routine will be cancelled but not the entire test. If a developer wants the entire test to fail they can use the macro `ASSERT_NO_FATAL_FAILURE(statement)` (with an `EXPECT` counterpart). The macro forces the test to fail but only if the sub-routine is executed within the same thread.

Concepts of CUTE missing in Googletest

Unconditional fail The `FAIL()` and `FAILM()` make the test fail unconditionally.

Range equality The `ASSERT_EQUAL_RANGES(expbeg, expend, actbeg, actend)` and `ASSERT_EQUAL_RANGESM(msg, expbeg, expend, actbeg, actend)` fail if the ranges defined by `expbeg, expend` and `actbeg, actend` are different.

6.2.6. Examples

Simple assert test

Googletest Code

```
1 #include "gtest/gtest.h"
2
3 TEST(SimplestTest, AssertTrue) {
4     ASSERT_TRUE(true);
5 }
6
7 int main(int argc, char **argv) {
8     ::testing::InitGoogleTest(&argc, argv);
9     return RUN_ALL_TESTS();
10 }
```

Listing 6.5: Googletest ASSERT_TRUE(cond)

Googletest AST

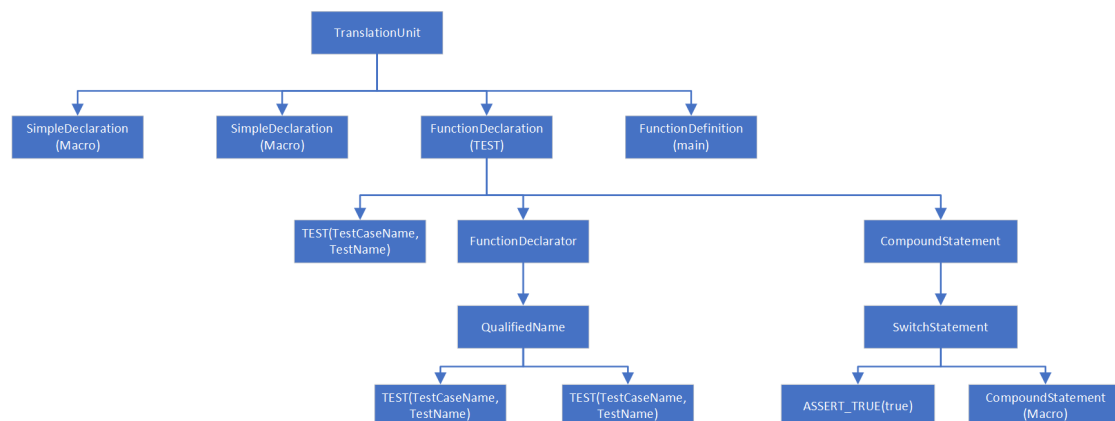


Figure 6.5.: Googletest AST

CUTE Code

```
1 #include "cute.h"
2 #include "ide_listener.h"
3 #include "xml_listener.h"
4 #include "cute_runner.h"
5
6 void thisIsATest() {
7     ASSERT(true);
8 }
9
10 bool runAllTests(int argc, char const *argv[]) {
11     cute::suite s { };
12     //TODO add your test here
13     s.push_back(CUTE(thisIsATest));
14     cute::xml_file_opener xmlfile(argc, argv);
15     cute::xml_listener<cute::ide_listener<>> lis(xmlfile.out);
16     auto runner = cute::makeRunner(lis, argc, argv);
17     bool success = runner(s, "AllTests");
18     return success;
19 }
20
21 int main(int argc, char const *argv[]) {
22     return runAllTests(argc, argv) ? EXIT_SUCCESS : EXIT_FAILURE;
23 }
```

Listing 6.6: CUTE ASSERT(cond)

CUTE AST

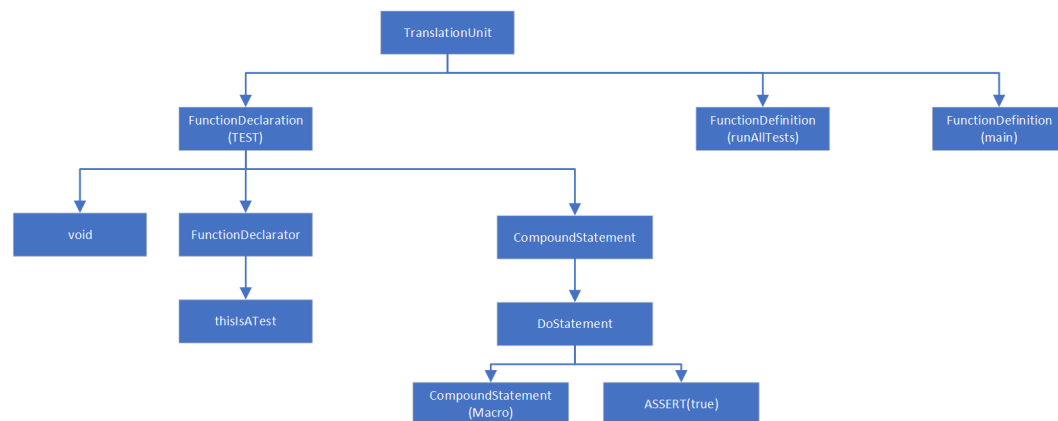


Figure 6.6.: CUTE AST

7. Design

Since the goal was to change code in an existing project, the only way to achieve this without heavy string-manipulation is to use the AST. This allows for a structured analysis of the code as well as a means to replace specific constructs easily, since the AST-nodes can be rewritten or replaced entirely. There were 2 major scenarios in our project, which get handled differently. Singlesuite and Multisuite (i.e. projects with only a single testsuite and those with several).

7.1. SingleSuite Case

Files which contain a single suite (or TestCase in Googletest) belong to this case. They can be refactored directly in the file. This was our starting point, being able to convert a single test in a single suite/testcase. First of all, the AST has to be combed for Nodes which need to be replaced/changed. This is done with a visitor, which visits all nodes. In the visitor we check if a node contains an assertion or is a TEST macro. The visitor collects those for the refactoring which takes these lists and performs the actual refactoring on them. These changes can be done directly on the AST, with an ASTRewrite inbetween which replaces the Nodes with a newly created Node containing the correct code.

7.2. MultiSuite Case

This was the next step to tackle. Since every suite should have its own header-and implementation-file [IFS, 2018c] there needs to be one AST per suite. In this case it is not possible so simply rewrite the AST. Therefore another approach was necessary. Here templates are used to put together the correct structure of the files. The Nodes are directly changed on a nonfrozen copy of the AST, then the ASTWriter is used to write the code into the template. When finished, the template is written to a new File using a resourcechange, in order for it to be reversible. After the files are created, the old Googletest files need to be deleted, since these would lead to redefinitions.

8. Implementation

In this chapter we describe what features were implemented as well as how we implemented them. It will contain a section for every implemented feature with a short problem description and the implemented solution.

8.1. Markers

There are some constructs in Googletest, for which there is no equivalent in CUTE. If a project contains one of those the plugin needs to be able to handle it, without crashing or leaving the code in an inconsistent state.

Solution Quite early on we decided to not go through with the refactoring if there is a single unsupported construct. Since this would lead to broken code. Now we were in need of a way for this issue to be communicated to the user. For this we were told to use markers depicting a problem. When the user tries to start the refactoring, check whether any of those markers exist, and if they do, the whole process is cancelled. In order to set and check for those markers, we chose to use the marker-category "codanProblem". This seemed to work just fine until we were reminded that basically all markers inherit from this class. So we needed to create a subcategory (subclass) of codanProblem which we would set and check for. We decided to go with a sort of "3-tier" inheritance model. Using a marker called gConverterMarker which we test for and then two submarkers which we actually set. One for each type of unsupported construct (currently TEST_F and the unsupported assertions). This way we can give the user feedback and a visual helper to show why the refactoring is not possible and where the offending code-part is.

8.2. Writing the AST to a new file

We started by only supporting a single suite. In this approach we were able to simply rewrite the AST of the containing file. When we extended our logic to support multiple suites we ran into a problem. According to CUTE best-practice, every suite should have its own header and implementation file. Now we needed to create two files per suite and our previous approach did no longer work seeing as new files do not contain an AST until they are processed by the preprocessor.

Solution We needed to somehow change the nodes and collect them into a single string which we could then write into the newly created files.

We began by reading the AST into a string on which we executed the refactoring as string manipulation. This is obviously very error-prone and ill-designed. Therefore, another way was necessary. Here our advisers told us to work on a copy of the AST, which is non-frozen, allowing us to directly replace the corresponding nodes. Afterwards we can write this "correct" AST with an ASTWriter which translates the given AST to a single String we can then write to the new file. We had one more obstacle with this approach, since Googletest is heavily based on macros we then had the problem that some constructs were still standing after the refactoring. Consequently, it was necessary to find another way to eliminate those. We already experimented with templates before and decided that these would work for the given problem. So instead of rewriting the entire AST, we rewrite single nodes, write those with the ASTWriter and then put them in a map. After refactoring all nodes we use the templates to create the suite files and fill in the node strings from the map.

8.3. Nontest constructs

The source files which we want to refactor, can contain code which does not belong to a test specifically. This includes: variables, functions, namespaces and others. These need to be moved to the new files as well. One specific problem here was that we have no way of knowing which functions or variables are needed for which tests.

Solution Since we do not know which tests use which variables/functions, we need to copy them once for every file we create. This would lead to nameclashes. Hence we decided to use an anonymous namespace for each file in which we pack all of the nontest constructs. With this we only have one more (quite specific) problem: When the file already contains an anonymous namespace which in itself contains a definition of a function or variable with the same name as one outside of the namespace. Also see the subsection "Same definition in anonymous namespace and outside of it" in the problem-section below.

8.4. Constructs which block the refactoring

As mentioned before, if there are any constructs which are not supported (yet) the refactoring cannot be executed. It would be possible to leave the assertions standing and only refactor what is currently possible, since the GoogleTest header is not getting removed.

Solution These assertions are checked but do not lead to test failures if they do not succeed. Therefore, it was decided to not go through with the refactoring if there are any unsupported constructs at all.

8.5. Testing the Refactoring

One essential part of every software, is testing the logic. In order to test the refactoring, there needs to be a way to specify the files before and after the refactoring. These need to be compared to check if the correct files with the correct content exist.

Solution ILTIS provides an easy way of doing with its highlevel testing. this is done via RTS-Files. Those specify the files and their content before and after the refactoring. The test then reads those files and creates the specified content in a "current" project. On these created files, the refactoring takes place. Additionally an "expected" project gets created, containing the files specified as result of the refactoring. After the refactoring concludes, the testing framework compares the two projects files one after one to see if the correct files with the correct content exist. The same is done when testing for checkers and their blockage of the refactoring. In case of markers, the testing framework creates the specified files and then checks if markers get set on the specified lines. When testing for the blockage of the refactoring, the framework creates the files and tries to execute the refactoring on them. It then checks for a failure returned by the refactoring, implying that the refactoring failed due to the markers. Since we are altering, creating and deleting files, we decided not to write any microtests which only test one specific function at a time. This would lead to large setups before each test in order to provide each test with the exact data it would receive. Additionally, all parts of the refactoring need to work together, in order for it to produce a workable AST.

In case of the multiSuite case, several files get created. The testing framework did not have a specific way of testing this, so we had to specify the before as an empty file. This lead to ResourceExceptions, since the files to be created already existed when the refactoring was running. This was solved with an extension of the ILTIS framework by Hansruedi Patzen, in order to allow for the testing of filecreation and -deletion.

8.6. Deletion of Googletest files

The Googletest source file is no longer needed, after the refactoring has concluded, and must be deleted in order to avoid nameclashes and compiler errors due to the existence of two main functions. The ModificationCollector, which is used to bundle all Modifications done in the refactoring, did not support ResourceChanges. Thus, it was not possible to include the deletion of files in the refactoring. One workaround would have been to simply delete the file. This would have led to problems, as the deletion would not be reversed in case of the user cancelling the refactoring.

Solution After an update to the ILTIS framework by Hansruedi Patzen, the ModificationCollector allows for DeleteFileChanges to be added to it.

9. Result

9.1. Reached

- Filecreation according to the CUTE best-practice. (e.g. one file per suite with the corresponding header)
- Conversion of most commonly used assertions supported
- Complete conversion of a simple project (with a single test source file)

9.2. Not reached

We reached everything we planned. The following points are aspects of the framework which we knew, could not be implemented during this project:

- Projects which span over multiple files
- TEST_F tests
- Deathtests
- Custom assertions

9.2.1. Reasoning

There were a few instances in which we implemented features twice, since our first approach was not suitable. Additionally there were quite a few problems/misunderstandings when working with the Eclipse CDT. This is to expect when working in a completely new environment but nonetheless led to some lost time. Especially when we tried to solve a problem one way and were then informed of an easier way. In the end we decided to focus on refactoring and documentation in order to make sure the code and documentation we turn in are up to standard rather than rush another feature. We already excluded many features of a "finished product" early on in the project, as these were bound to be complex or very time-consuming. Therefore we will not be discussing the explicit reasoning for each.

Projects which span over multiple files We would have loved to implement this feature, but were not able to due to time constraints. When we got to the point where the rest of the code was working well enough for us to start working on this feature, there were only 3 weeks left in the project, with lots of documentation untouched. Thus we decided to note our findings and work on our documentation and code-refactoring instead of starting to work on a new feature with so little time left. The findings about this part can be found in the section "Notes for extending the plugin" 11.1.

10. Problems

10.1. Solved

10.1.1. Indexer running permanently, blocking the refactoring

When you started the refactoring, but then cancelled it, the read-lock on the AST did not get released. This led the Indexer not being able to conclude, which blocked any further attempts of running the refactoring. The workaround was to close and reopen the IDE or stop the indexer manually.

Solution The read-lock on the AST and thus, the file, was not released when the refactoringwizard was cancelled. This has since been fixed by the IFS.

10.1.2. Named Namespaces lead to redefinition

When there are one or more named namespaces in the sourcefile, the refactoring led to redefinitions. Since we cannot tell which functions are used in which tests, we copy the namespace into every suitefile. This led to redefinitions, as the same namespace then existed in several files.

Solution In order to avoid nameclashes, we copy the named namespaces of a file into an anonymous namespace, just like any other nontest construct.

10.2. Unsolved

10.2.1. Projects that include several testing files

So far we are not able to support a project which spans over several source files. One obstacle here is finding a way to create a single main file which contains links to all suites in the project. For that we probably need another layer above the refactoring which collects all files and suites in order to be able to link them all in the main file. Additionally the upper layer would need to collect the refactorings as well, in order to complete all the "singlefile" refactorings in one big CompoundRefactoring.

10.2.2. Same definition in anonymous namespace and outside of it

Also see the paragraph 8.3 "nontest constructs" in the architectural and implementational decisions. Since we pack both definitions into the same anonymous namespace, the refactoring leads to a compilererror (redefinition) We do not yet have a solution for this. Since this case is very unlikely to exist in a real project (the definition in the namespace gets shadowed and is thus useless). This would (according to our current knowledge) need quite an extensive workaround which is (in our opinion) not worth the time needed to solve this.

11. Conclusion

In the end, the plugin is, despite some limitations, usable and can be used as base which can be extended in the future. The solution provides the ability to convert a project with a single Googletest sourcefile into a CUTE project. Projects which span multiple files are not supported in the current version. There are several features and improvements that will need to be implemented. The list with some explanations can be found in the following section 11.1.

11.1. Notes for updating the plugin

This section shall act as a help for developers seeking to add features to the plugin. Here we note some of our findings regarding possible features and problems that arise with them.

11.1.1. Multifile Case

This should be the first feature that gets added when the plugin is extended. For this to work there are some changes that need to be made.

Changes

- Create a simple refactoring (the one already existing) for every file and bundle those into a single CompoundRefactoring
- Implement a layer before the (current) refactoring, pull out the creation of the main file. the Upper layer needs to know all suites and the name of the implementation file thereof in order to include those.

11.1.2. TEST_F tests

TEST_F tests are used when a test fixture is needed, they are based on an existing class of which an object gets created before the test and then deleted afterwards. The class needs to support a setUp() and a tearDown() method. These are automatically called through the Googletest macro. Analyze the macro some more on what it actually does in order to execute those steps manually.

11.1.3. Remaining assertions and other Googletest constructs

There are still some assertions missing. Some of which are not easily mappable to CUTE assertions, as well as some of which simply do not exist in CUTE. Deathtests and custom assertions fall into this category. Deathtests might be possible with a certain effort. Custom assertions on the other hand are pretty hard to even spot automatically, according to our current knowledge. Custom assertions can inherit from an existing assertion and expand the logic. Their name can be chosen more or less freely. Some more research and testing is necessary here in order to find out if it is possible to rebuild these in CUTE. Especially if and how it is possible to automate the process.

Problems

- Study the remaining Googletest assertions and find a way to convert them
- Study other Googletest constructs and how to convert them (Deathtests for example)

Next assertions One of the first assertions to add would be the `ASSERT_PRED(pred, value)`, since this can easily be mapped to `ASSERT(pred(value))`.

11.1.4. More defensive programming

In the current scope, the refactoring assumes that the code given to it is proper, working code. This is, so far, not a problem, since we are in a closed environment. Thus breaking the refactoring does not provide any benefit to the "breaker". Nevertheless, it would be nice to change the program to be more robust against broken code.

11.1.5. Remove namespace duplication

As of now every nontest construct (functions, variables, namespaces et cetera) is copied into each suitefile. This is necessary since we did not find a way to tell which tests use which functions. Find out if this is possible in order to remove the duplication.

11.2. Time analysis

This section shall bring some light into how the time in this project was spent.

11.2.1. Overall time per label

As visible in the graphic below, almost half the project time was spent on the implementation, while just over a quarter was spent on documenting.

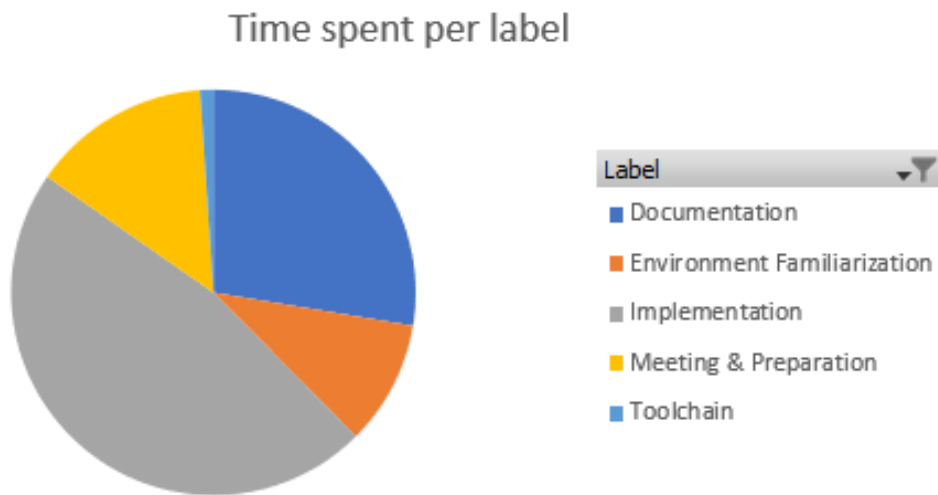


Figure 11.1.: Overall time spent per label

11.2.2. Comparison of estimates and actual time

This graph compares the time estimated and how much time we actually spent per sprint in order to get a grasp of how good our estimates were. Additionally, in the following images our effort over the course of the project is visible.

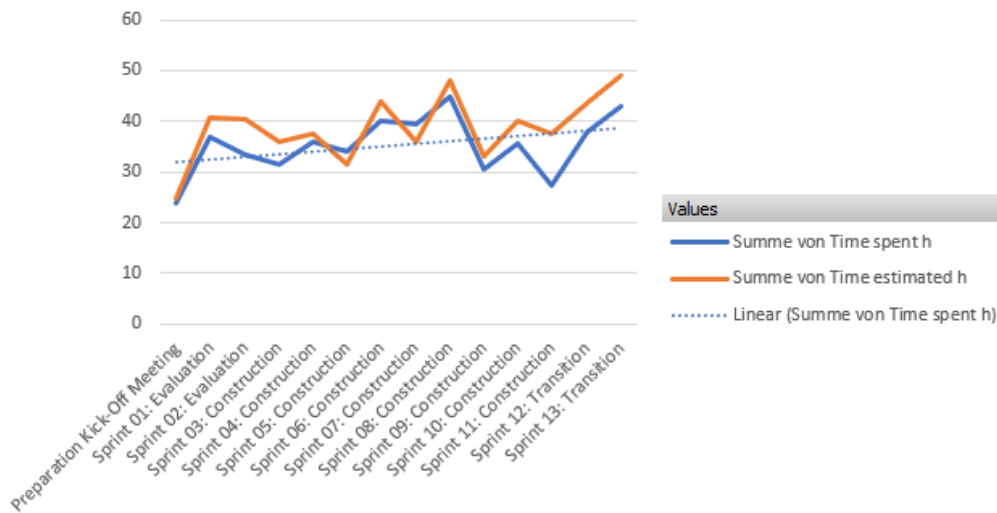


Figure 11.2.: Absolute time estimated and spent over the course of the project

As visible in the image above, we had quite a slow start. Over the course of the project we steadily increased our time spent with a clear peak in the middle, where most of the features were implemented. We also had most problems in the middle, leading to the only two sprints where we were above our estimates. The drop at the end of the project was due to illness. In the next figure we have a clearer comparison between estimated and spent time, the positive values represent sprints where our actual time exceeded the estimated time.

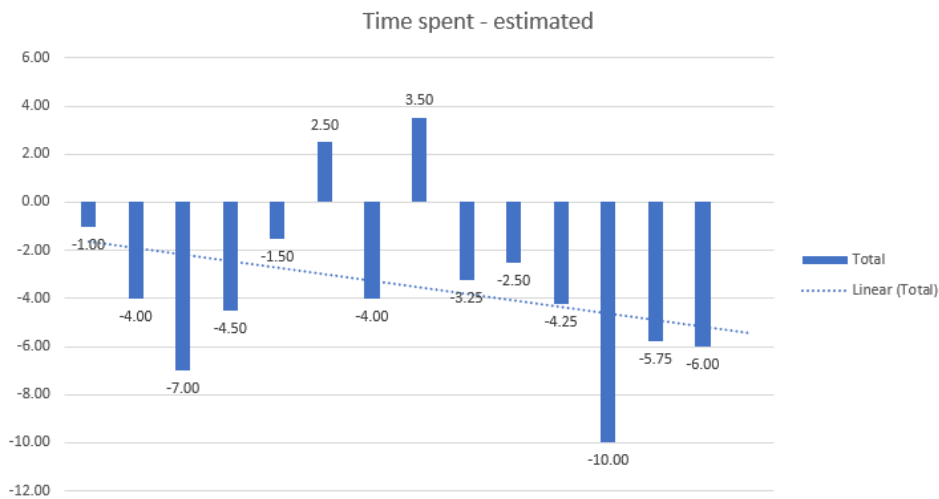


Figure 11.3.: Difference between estimated and actual time over the course of the project

11.2.3. Comparison between teammembers

Here we compare the time spent between the team members.

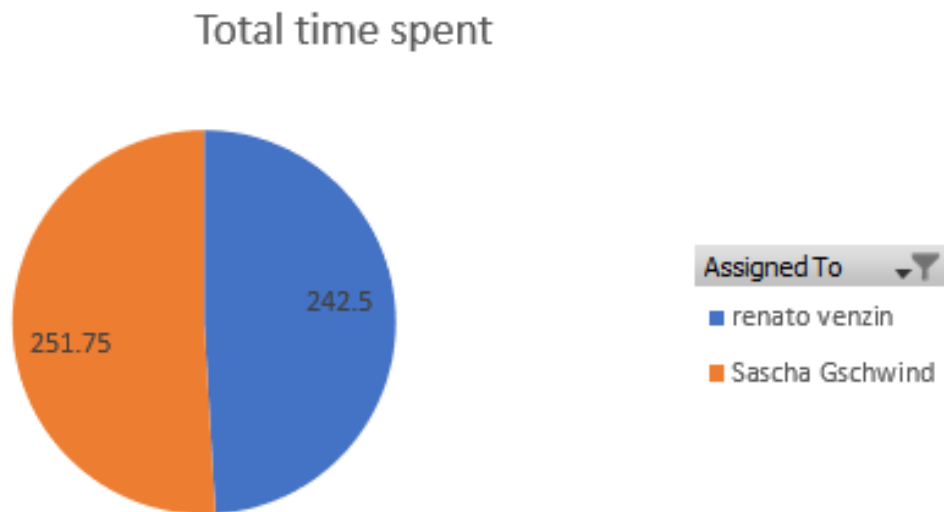


Figure 11.4.: Overall time spent per person

As visible in the graphic above, Sascha Gschwind spent about 10 hours more on this project than Renato Venzin. This is partly due to him being more experienced and taking over some of the more difficult tasks which took more time.

The next image shows how much time each team member spent per sprint. This graphic also shows that the two heaviest sprints were the ones where Sascha Gschwind spent quite some time more than Renato Venzin. As mentioned before, this is mainly due to him being more experienced and taking on harder tasks, which in these sprints took longer than expected, due to unforeseen problems. Also visible is that in sprint 11, Sascha Gschwind was quite a bit below the average time, this is due to him falling ill and not being able to work.

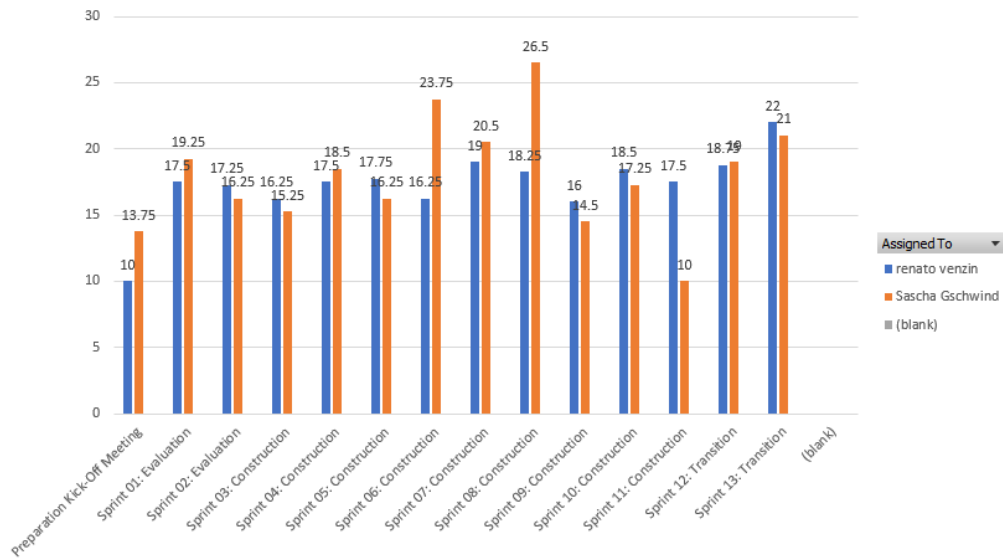


Figure 11.5.: Comparison of teammates over the course of the project

12. Glossary

AST Abstract syntax tree, the abstract notation of code, consisting of nodes depicting Code-snippets

Googletest Google testing framework

CUTE The testing plugin which was developed and is being maintained at the HSR

TEST_F Tests Tests based on an existing class which has a setUp and a tearDown method, special type of test in Googletest

Marker Tool to highlight a single, or multiple lines in a code-file

IFS Institute for Software

HSR University of Applied Sciences Rapperswil

HS Autumn semester

SA 'Studienarbeit' representing the current project.

MVP Minimum viable product

RUP Rational Unified Process

Bibliography

- [Atlassian, 2018] Atlassian (2018). Sourcetree. <https://www.sourcetreeapp.com/>. accessed 12.12.2018.
- [Benito van der Zander, 2018] Benito van der Zander, Jan Sundermeyer, D. B. T. H. (2018). Texstudio. <https://www.texstudio.org/>. accessed 12.12.2018.
- [CCTLSU, 2018] CCTLSU (2018). Astwriter class. <https://www.cct.lsu.edu/rguidry/eclipse-doc36/org/eclipse/cdt/internal/core/dom/rewrite/astwriter/ASTWriter.html>. accessed 05.12.2018.
- [Checkerframework, 2018] Checkerframework (2018). Eclipse checker framework. <https://checkerframework.org/manual/>. accessed 05.12.2018.
- [Denim, 2012] Denim, A. (2012). An interview with peter sommerlad. <http://demin.ws/blog/english/2012/05/19/peter-sommerlad-interview/>. accessed 17.12.2018.
- [Eclipse, 2018a] Eclipse (2018a). Eclipse documentation. <https://help.eclipse.org/neon/index.jsp>. accessed 26.11.2018.
- [Eclipse, 2018b] Eclipse (2018b). Eclipse pde. <https://www.eclipse.org/pde/>. accessed 12.12.2018.
- [Erich Gamma, 1997] Erich Gamma, Richard Helm, R. J. J. V. (1997). *Design Patterns : Elements of Reusable Object-Oriented Software*, volume 1. Pearson Education (US). ISBN 978-0-201-63361-0.
- [GitLab, 2018] GitLab (2018). Gitlab website. <https://about.gitlab.com>. accessed 05.12.2018.
- [Google, 2018a] Google (2018a). Googletest advanced documentation. <https://github.com/abseil/googletest/blob/master/googletest/docs/advanced.md>. accessed 26.11.2018.
- [Google, 2018b] Google (2018b). Googletest simple documentation. <https://github.com/abseil/googletest/blob/master/googletest/docs/primer.md>. accessed 26.11.2018.

- [IFS, 2016] IFS (2016). *ILTIS documentation*. HSR.
- [IFS, 2018a] IFS (2018a). Cvelop ide. <https://www.cevelop.com/>. accessed 26.11.2018.
- [IFS, 2018b] IFS (2018b). Cute documentation. <https://cute-test.com/guides/>. accessed 26.11.2018.
- [IFS, 2018c] IFS (2018c). Cute eclipse plug-in guide. <https://cute-test.com/guides/cute-eclipse-plugin-guide/#sourcefileorganization>. accessed 05.12.2018.
- [Ken Schwaber, 2018] Ken Schwaber, J. S. (2018). The scrum guide. <https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf>. accessed 05.12.2018.
- [LaTeX, 2018] LaTeX (2018). Latex website. <https://www.latex-project.org/>. accessed 05.12.2018.
- [Microsoft, 2018a] Microsoft (2018a). Microsoft excel. <https://products.office.com/en/excel>. accessed 12.12.2018.
- [Microsoft, 2018b] Microsoft (2018b). Microsoft visio. <https://products.office.com/en/visio/flowchart-software>. accessed 12.12.2018.
- [Microsoft, 2018c] Microsoft (2018c). Unit testing best practices with .net core and .net standard. <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices>. accessed 17.12.2018.
- [Microsoft, 2018d] Microsoft (2018d). Visual studio code. <https://code.visualstudio.com/>. accessed 12.12.2018.
- [Oracle, 2018a] Oracle (2018a). Astrewrite class. <https://help.eclipse.org/luna/index.jsp?topic=> accessed 05.12.2018.
- [Oracle, 2018b] Oracle (2018b). Java website. <https://www.oracle.com/ch-de/java/>. accessed 05.12.2018.
- [Stroustrup, 2015] Stroustrup, B. (2015). *Die C++ Programmiersprache*, volume 4. Hanser Fachbuchverlag. ISBN 978-3-446-43961-0.
- [Thomas Kuhn, 2006] Thomas Kuhn, O. T. (2006). Abstract syntax tree. https://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/. accessed 26.11.2018.

[Vogella, 2018] Vogella (2018). Eclipse ide plug-in development: Plug-ins, features, update sites and ide extensions. <http://www.vogella.com/tutorials/EclipsePlugin/article.html>. accessed 26.11.2018.

Appendices

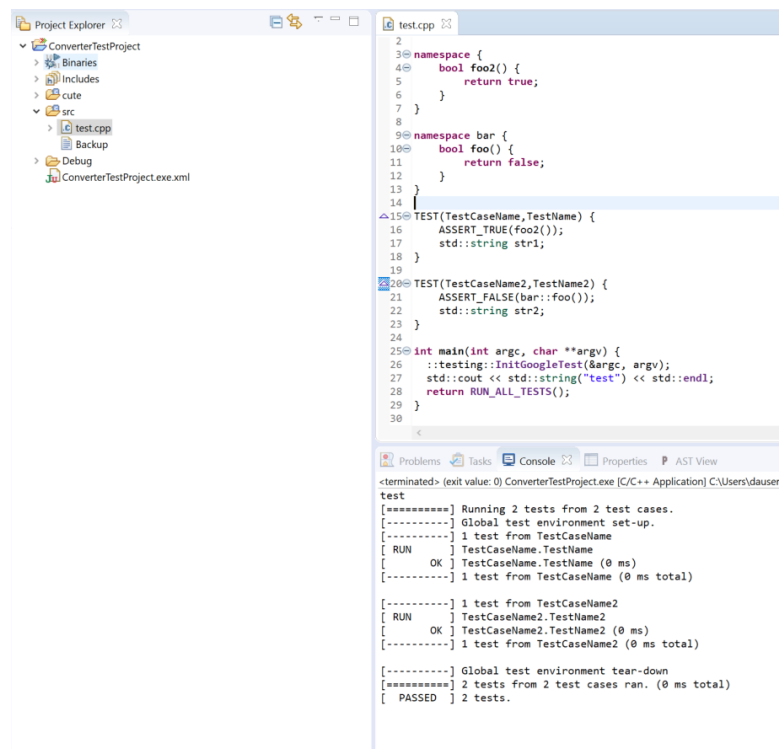
A. Usage manual

1 Prerequisites

To use this manual certain prerequisites have to be given.

- You need an IDE for C++ development that can install Eclipse plugins. We recommend using Cevelop as the IDE.
- The CUTE plugin has to be installed. You can read through the CUTE plugin tutorial <https://cute-test.com/guides/> for an installation guide.
- You should have a project using Googletest with a test file containing the tests and the main function which you want to convert.

Your project should look something like this:



The screenshot shows an IDE window with a Project Explorer on the left and a code editor on the right. The Project Explorer shows a project named 'ConverterTestProject' with folders for 'Binaries', 'Includes', 'cute', 'src', and 'Debug'. The code editor shows the file 'test.cpp' with the following code:

```
2
3 namespace {
4     bool foo2() {
5         return true;
6     }
7 }
8
9 namespace bar {
10    bool foo() {
11        return false;
12    }
13 }
14
15 TEST(TestCaseName, TestName) {
16     ASSERT_TRUE(foo2());
17     std::string str1;
18 }
19
20 TEST(TestCaseName2, TestName2) {
21     ASSERT_FALSE(bar::foo());
22     std::string str2;
23 }
24
25 int main(int argc, char **argv) {
26     ::testing::InitGoogleTest(&argc, argv);
27     std::cout << std::string("test") << std::endl;
28     return RUN_ALL_TESTS();
29 }
30
```

The console output at the bottom shows the test results:

```
<-terminated> (exit value: 0) ConverterTestProject.exe [C/C++ Application] C:\Users\dauser\ru
test
[=====] Running 2 tests from 2 test cases.
[-----] Global test environment set-up.
[-----] 1 test from TestCaseName
[ RUN    ] TestCaseName.TestName
[ OK     ] TestCaseName.TestName (0 ms)
[-----] 1 test from TestCaseName (0 ms total)

[-----] 1 test from TestCaseName2
[ RUN    ] TestCaseName2.TestName2
[ OK     ] TestCaseName2.TestName2 (0 ms)
[-----] 1 test from TestCaseName2 (0 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 2 test cases ran. (0 ms total)
[ PASSED ] 2 tests.
```

Figure 1.1: Project before (using Googletest)

2 How to use the Googletest Converter

1. Open the project in the IDE and open the file you want to convert.
2. Make sure there are no Googletest Converter warnings present. These warnings could be certain ASSERT / EXPECT statement that can not be converted or if you are TEST_F tests.

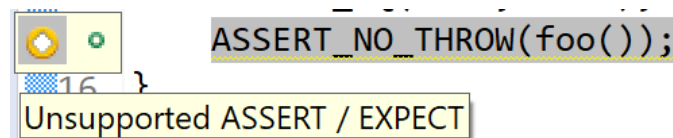


Figure 2.1: Warning example

3. Right click the file in the project explorer, choose the submenu 'CUTE' → 'Convert to CUTE Project'. This will open up a refactoring wizard. If you do not have any warnings the wizard will show you the changes, otherwise you will see an error message that the conversion did not take place.

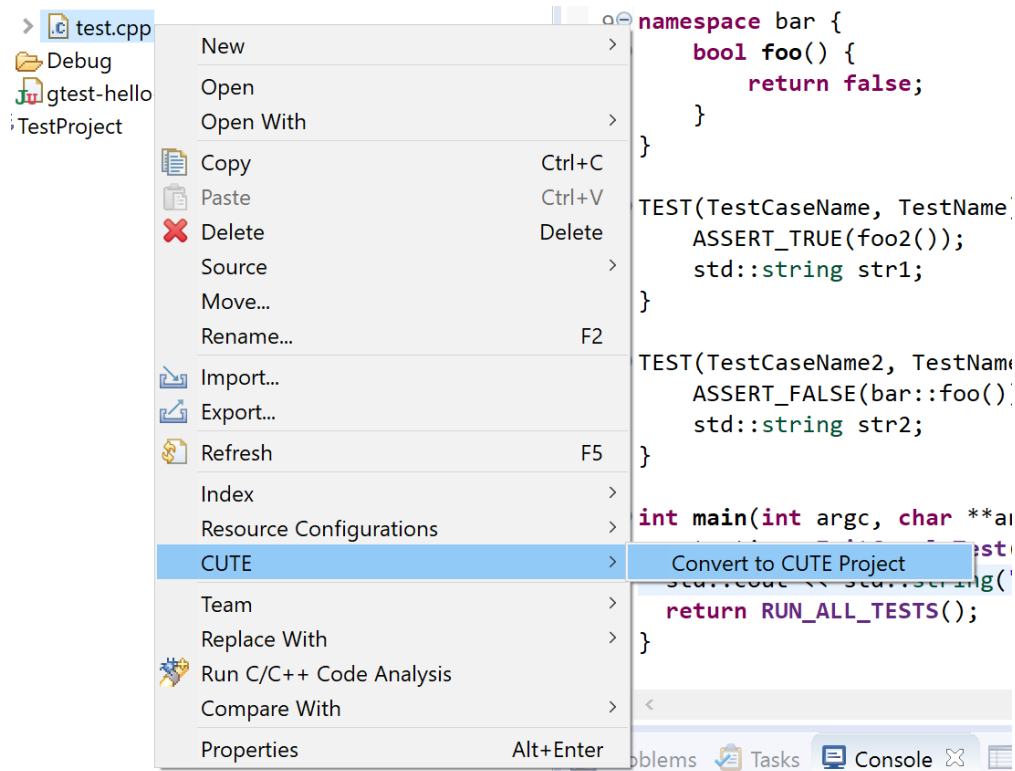


Figure 2.2: Menu location

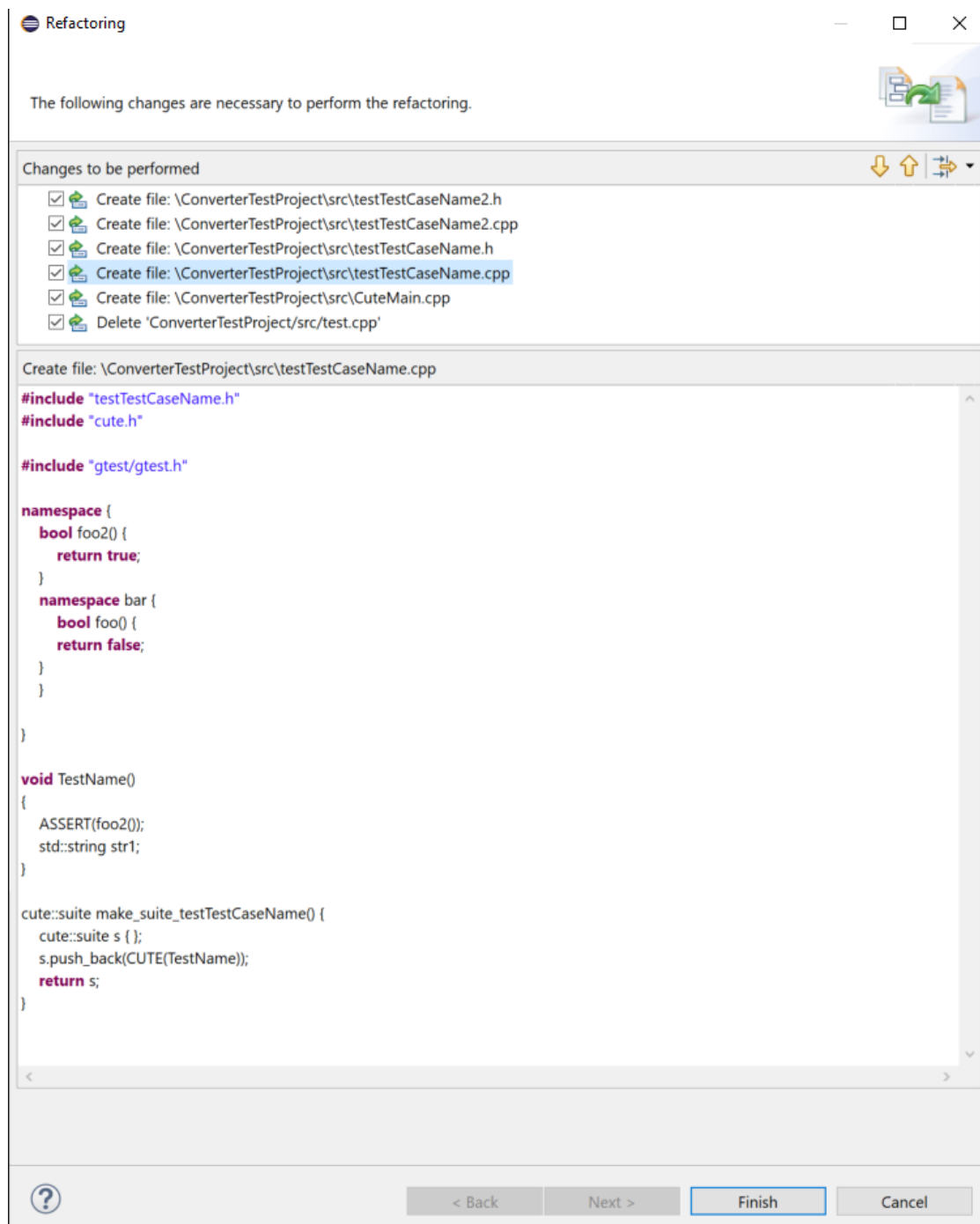


Figure 2.3: Refactoring wizard

4. Once you inspected the changes you can click on 'Finish'. This will execute the refactoring applying the changes to the project.
5. If you have not added the CUTE nature yet, you should right click the project and select 'CUTE' → 'Add CUTE Nature'. This adds the CUTE headers to the project so you can run the changed code. If you have already added the CUTE nature you can skip to step 3.

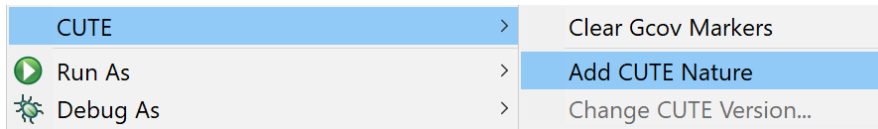


Figure 2.4: Add CUTE Nature

6. Once you have built the code you can right click the project to run the tests which are now using the CUTE testing framework.

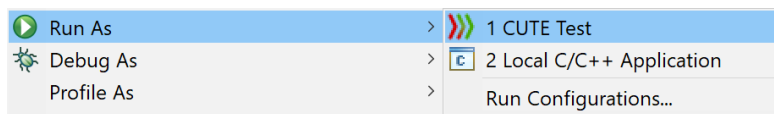


Figure 2.5: Run CUTE tests

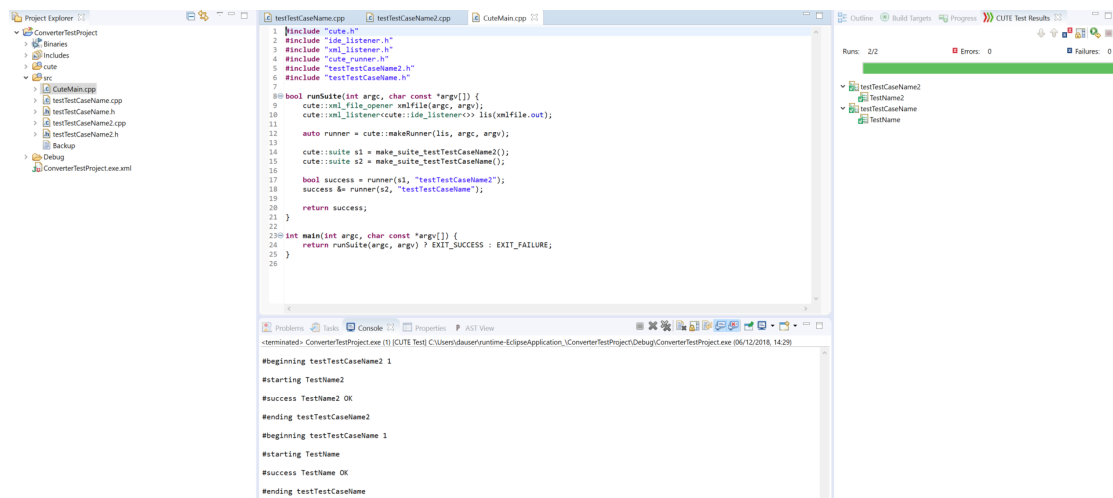


Figure 2.6: Project after (using CUTE)

3 Problem solutions

This chapter shows some common problems that can occur during the conversion and how to overcome these problems.

3.1 CUTE header not found

The one problem that occurred for us was that the CUTE header could not be found. The way to solve this problem was to remove and readd the CUTE nature.

3.2 Project can not be built after conversion

If this problem occurs, all you have to do is to clean the project and rebuild it again. The problem is most likely due to old files, that do not get overwritten when building the project. Once you have cleaned and rebuilt the project this problem should not occur again.

B. Demonstration

1 Demonstration steps

To demonstrate the functionality of the plugin we chose to use a pretty simple project. The reason for that is that complex projects usually consist of multiple test files, which is a function that is not yet implemented. Additionally those projects usually use one or more constructs that are not yet supported.

The chosen open source project for this demonstration is <https://github.com/guidohcosta/gtest-hello-world>.

- First we cloned the repository and included it to Eclipse.
- The next step was to make sure the Googletest include is made. For more information on how to do this check the Googletest manuals <https://github.com/google/googletest/blob/master/googletest/docs/>.
- After that the project looked like this:

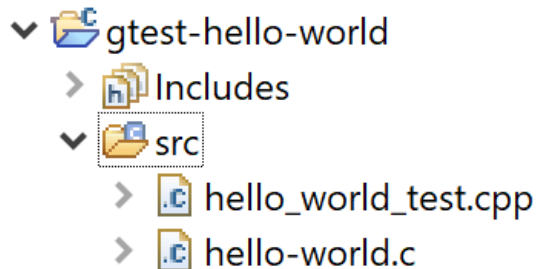


Figure 1.1: Project before (using Googletest)

- We then executed the tests to check if they run correctly:

```
[=====] Running 2 tests from 1 test case.  
[-----] Global test environment set-up.  
[-----] 2 tests from MathTest  
[ RUN      ] MathTest.TwoPlusTwoEqualsFour  
[       OK ] MathTest.TwoPlusTwoEqualsFour (0 ms)  
[ RUN      ] MathTest.Salsixa  
[       OK ] MathTest.Salsixa (0 ms)  
[-----] 2 tests from MathTest (0 ms total)  
  
[-----] Global test environment tear-down  
[=====] 2 tests from 1 test case ran. (0 ms total)  
[ PASSED  ] 2 tests.
```

Figure 1.2: Tests run with Googletest

- We then started the conversion through the corresponding menu:

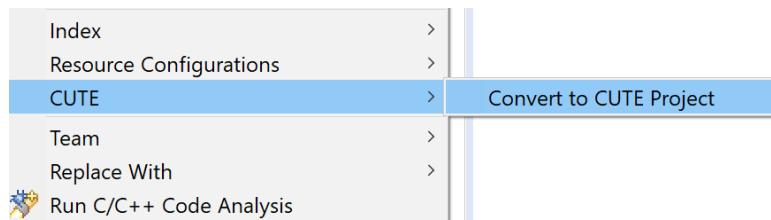


Figure 1.3: Convert to CUTE project

- The refactoring wizard opened showing us the changes that will be made:

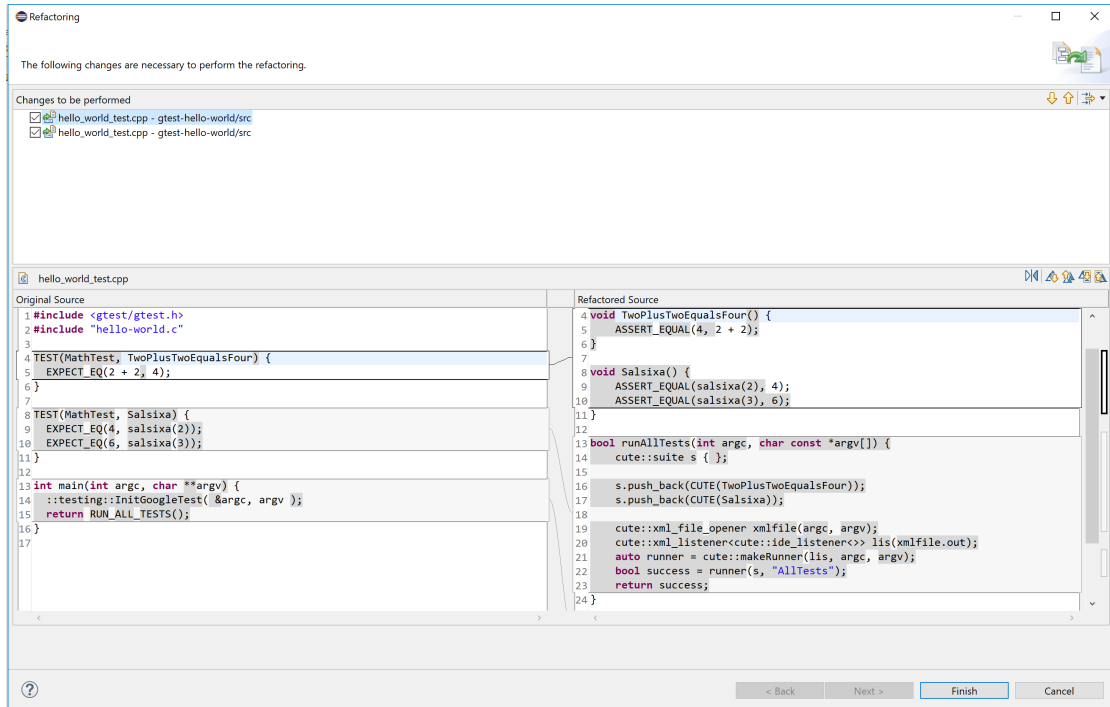


Figure 1.4: Refactoring Wizard

- After clicking on 'Finish' the refactoring took place.
- After that we added the CUTE nature to the project:

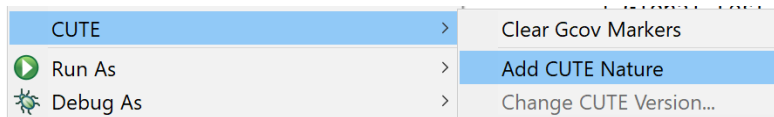


Figure 1.5: Adding CUTE Nature

- We then executed the tests using the CUTE framework.

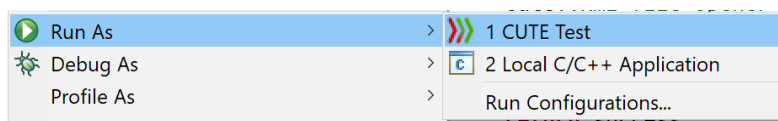


Figure 1.6: Run tests with CUTE menu

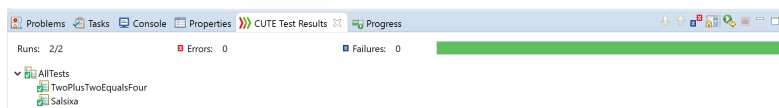


Figure 1.7: CUTE test runner