HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Student Research Project

Googletest to CUTE: a step further

University of Applied Sciences Rapperswil

Spring Term 2019

Author:	Muriel Thévenaz
Advisor:	Peter Sommerlad
Technical Advisor:	Toni Suter

Task Description

At the beginning of the semester, advisors and student agreed on the task that had to be done within the given time. The task description was part of the Wiki in the project repository (IFS private GitLab[9]).

Task description

Last edited by **Thevenaz** just now

MOTIVATION

C++ is a very powerful programming language that has the reputation to be very tricky and error proned. That means: Maybe even more than in Java or C#, good testing is a key component of writing robust softwares in C++!

When someone searches for "best testing framework c++" on Google, Googletest pops out as the most popular developer choice. And the fact that Google obviously supports its own product doesn't explain this result alone: Googletest is indeed a widely used C++ testing framework. The problem is that Googletest doesn't comply with important quality standards regarding to software testing. It mainly relies on macros, which implies that the program logic is hidden and therefore not very understandable for the developer. It also supports nonfatal failures, what means that it's allowed to write a test that doesn't stop the main testing function from going on even if it fails. Besides, Googletest works as an external library that has to be installed and compiles on her own.

There are other ways of testing C++ code. CUTE is one of them. It's a header-only library that supports Test Driven Development (TDD) and always complies with the newest C++ standards, since its maker is part of the standardization team.

> CUTE is an easy to use C++ unit testing framework that leverages modern C++ libraries and features. CUTE is orthogonal, easily extendable and much simpler than other C++ unit testing approaches (...).

Googletest	CUTE
Based on macros	Based on functions
External library	Header-only library
Supports nonfatal failures	Prevents from nonfatal failures
	Supports TDD

In order to motivate developer to use CUTE instead of Googletest, it seems important to provide them with a way of converting existing test projects from Googletest to CUTE with only a few clicks. That would allow them to test CUTE, and if they like it, they can use it directly on their ongoing project. Initiated by the IFS, the institute behind CUTE, and supervised by Prof. Peter Sommerlad, head of this institute, this project aims to develop exactly such a conversion tool.

PROJECT GOALS

The aim of the project is to produce an Eclipse plug-in that can convert a Googletest project to a CUTE project. Fall 2018, another student team worked on the plug-in and developed a first solution. This solution covers a simple scenario with an unique test file and is therefore not suitable for most projects. The idea is to build up on this first solution, refactor it where needed, and potentially even make the plugin robust enough for a first release.

EXPECTED RESULTS

Main features

- Implement conversion for test projects with several files (most common case)

For the moment, the plug-in works only for simple Googletest projects with just one file. But most of the real projects need more than one file to be tested. Therefore, the most urgent next step to be done is to implement a solution suitable for such cases, converting a test project with several files.

- Implement conversion for TEST_F

Googletest works with two different kind of tests implemented in two different classes: tests without (TEST()) and with (TEST_F()) fixtures. The latest provides a way to initialize test objects (a fixture) by overriding a SetUp()-method, use it for different tests, and delete it by overriding the TearDown()-method. The current solution of the plug-in doesn't cover TEST_F at all. Since this kind of tests are very common in the practice, it would be important to implement a solution for these too.

- Automate the "Add CUTE Nature" step

In the current solution, it is necessary to add the CUTE headers manually (CUTE -> Add CUTE Nature). This step can be easily forgotten, leading to a test project that doesn't work. It would be a good thing to automate this step, so that the developer using the plug-in doesn't have to think about it.

Optional features

- Implement a solution for missing assertions

The current solution is able to convert the basic assertions, but there is still assertions that are not covered yet, and when these are used, an error is thrown and the conversion can not be completed. It would be good to implement a mapping solution for these assertions, at least for most of them.

- Fix little bugs

While working on the project, some problems with the current solution will surely be encountered. A list of each of these problems will me maintained in order to be fixed one by one according to their importance.

TIME MANAGEMENT

Milestone	Title	Deadline	Description
M0	Project Start	18.02.2019	Official start of the Semester, and therefore of the project
M1	Multiple Test Files	08.04.2019	The plug-in works with several test files
M2	TEST_F	06.05.2019	The plug-in works with TEST_F
M3	Code Freeze	20.05.2019	The coding work is finished
M4	Project End	31.05.2019	Code and documentation are handed to Prof. Sommerlad

DELIVERABLES

- Poster (as PDF only)
- Color print (two-sided)
- Monochrome print (one-sided)

Abstract

"Googletest to CUTE: a step further" is a student project conducted over a semester (15 weeks) for the Institute for Software (IFS) at the University of Applied Sciences Rapperswil. The aim of the project was to amend CUTE[7], the C++ testing framework integrated into the IFS IDE Cevelop[10], with a way of automate the conversion of Googletests[4] to CUTE tests. Another student team created an initial version of that conversion plug-in before. It was able to convert all main Googletest assertions, but only for a single file at once. Since most of the real-life projects need more than one file with tests, the most urgent next step was to implement a solution suitable for tests spreading over several translation units. Another important goal was to implement a conversion for Googletest's TEST_F test registration macro, that provides a way to initialize test objects (a fixture) by overriding a SetUp()-method and delete it by overriding a TearDown()-method. Since tests with fixtures are common practice, it seemed important to implement a solution for these too.
The final plug-in was tested on the project Libphonenumber[5] that uses the macros TEST (without fixture) and TEST_F (with fixtures). The plug-in was able to convert all test case files at once, fulfilling the first project goal. Fixture classes could also be converted properly. Some problems remained though, preventing the converted project to compile. First, some advanced Googletest constructs[3] couldn't be converted (in just one test file). Then, the converted project couldn't resolve some dependencies, since Libphonenumber uses test classes as C++ "friends" whose names were changed during the conversion. A further remaining problem was a type error comparing pointers with ints (the number 0 representing nullptr) that happened to be used several times in Libphonenumber.

Management Summary

The present management summary presents the result of a semester project dedicated to the conversion of Googletest[4] to CUTE[7]. The project took place over 15 weeks during Spring 2019 and was conducted by one student, Muriel Thévenaz, under the supervision of one technical advisor, Toni Suter, and one main advisor, Peter Sommerlad.

What is this document about?

The Institute for Software (IFS)[8] has been developing plug-ins for Eclipse CDT (as in "C/C++ Development Tooling")[1] for many years. One of its main projects is a C++ testing framework named CUTE, integrated into the IFS IDE Cevelop[10]. CUTE is a header-only library that supports Test Driven Development (TDD) and always complies with the newest C++ standards.
Today, one of the widest used C++ testing frameworks is Googletest.[2] But in contrast to CUTE, Googletest relies on macros for automatic test registration, what implies that the program logic is hidden and therefore not very understandable for the developer. Furthermore, Googletest doesn't comply with "agile" quality standards of unit testing by supporting nonfatal failures, what means that it's allowed to write a test that doesn't stop the main testing function from going on even if it fails. Besides, Googletest works as an external library that has to be installed and compiles on its own.

Googletest	**CUTE**
Based on macros	Based on functions (what minimizes the use of macros)
External library	Header-only library
Supports nonfatal failures	Prevents from nonfatal failures
	Supports TDD

Comparison between Googletest and CUTE

In order to motivate developers to use CUTE instead of Googletest by making it easier to switch to the more modern framework, IFS wants to provide a way of converting existing testing projects from Googletest to CUTE with only a few clicks. The aim of the present student work was to develop exactly such a conversion tool: an Eclipse plug-in that can convert a Googletest project to a CUTE project. Another student team created an initial version of that conversion plug-in before. It was able to convert all main Googletest assertions, but only for a single file at once, and was therefore not yet suitable for real-life projects since they often have more than one test file. The present student project has

built up on this first solution, refactored it where needed, and brought the plug-in one step further toward its first release.

What does the plug-in do?

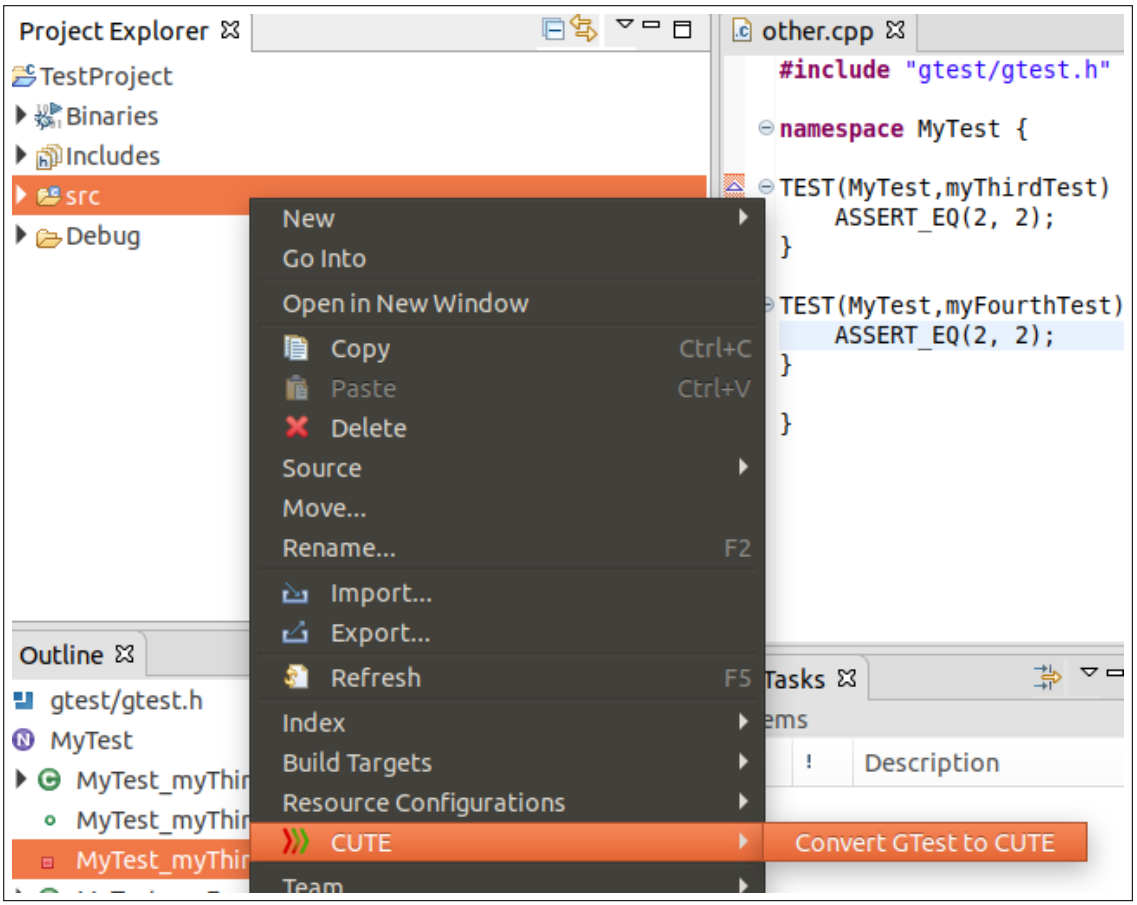
Commonly, a project needs more than one file with tests. Therefore, the conversion plug-in had to be changed in order to become suitable for tests spreading over several translation units. The actual plug-in supports traversing a project, including different levels of directories, to find all implementation files that contain one or more Googletests to convert. The tests and all main assertions are converted to their CUTE counterparts (some less important assertions are still not supported), and are registered as they should. A header file is also created for each converted file to comply as good as it gets with the CUTE best practice of having one implementation/header pair per test suite (depending on the original Googletest file, more that one test suite may appear in one implementation/header pair). Towards the end of the conversion process, the plug-in creates a new "Cute_Main.cpp" file to run the CUTE tests.
The plug-in is able to handle the conversion of both normal Googletests (TEST macro) as well as tests with fixtures (TEST_F macro), which allow several tests to use the same initial setup, cleaning the system afterwards. In CUTE, the reuse of test data exists too, but looks a little differently. The plug-in conducts the required changes. If a file implements TEST and TEST_F together, the conversion keeps working.
When something goes wrong - for instance because some assertions are not supported by the plug-in -, warning notices help the user to better understand what happened. If the problem is not of severe nature, the conversion can be pulled through. This way, a small problem easily handled manually doesn't stand in the way of converting a whole project.

How to use the conversion plug-in?

Before starting, make sure that the *prerequisites* for the conversion are satisfied:

- Your C++ (test) project is imported in Eclipse CDT[1]
- It was imported as normal C++ project (and not, for instance, using CMake[11])
- The CUTE plug-in[7] is installed
- Your C++ project is built

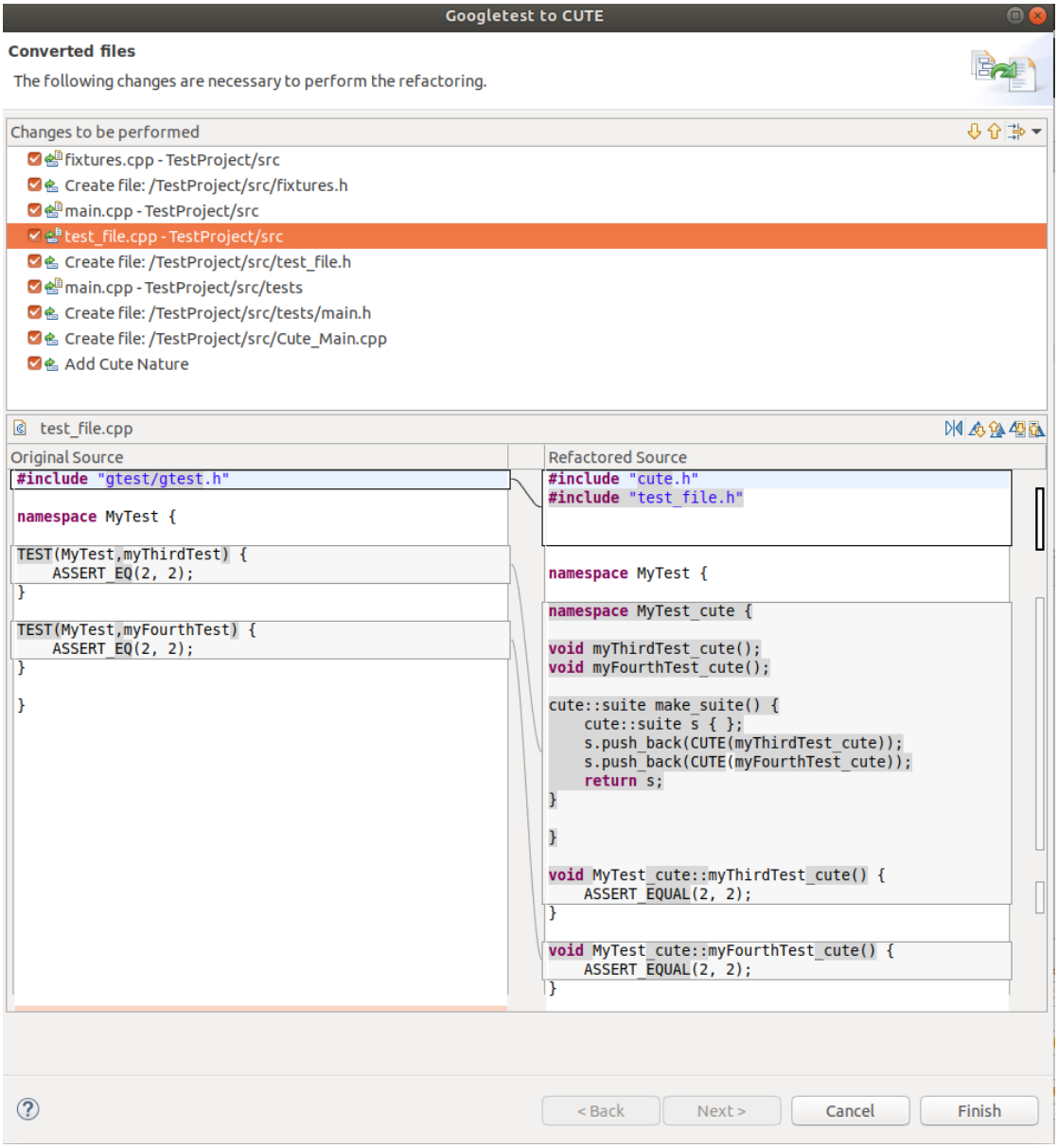
To *start the conversion*, right click on the project, folder or file you want to convert and choose "CUTE → Convert GTest to CUTE" in the menu.
After all files have been processed (depending on the size of the project, this step may take some time), the *changes are visualized* in a preview wizard. In the upper part of the wizard, you'll find a list of all suggested changes. Selecting one of them will show you more details concerning the change:



Starting the conversion process in Eclipse

- For any implementation file that contained a Googletest macro (TEST or TEST_F), you should be able to see the differences between the old and the new (CUTE) version of the file
- For any converted file, a header file should have been newly created and selecting it should show you the content of the new file
- Toward the end of the list, you should see the newly created "Cute_Main.cpp" file, which contains your new main function to run all the converted tests
- The last element of the list should be the change "Add CUTE Nature", which allows you to automatically add the nature needed to run the CUTE tests (selecting this change will show nothing)

You can select and unselect any change listed in the preview wizard. When you eventually click the "Finish" button, the conversion is *concluded* and all selected change are definitely



Preview from all suggested changes

operated. It is still possible to go back to your original files using the common "undo" Eclipse functionality. *Build* the converted tests and try to run them. Depending on the size and the complication level of the converted project, some adjustments still may be needed before the tests can run properly.

What are the next steps?

Towards the end of the project, the conversion plug-in was tested on an existing project found in the Internet: Libphonenumber, which is a Java, JavaScript and C++ library developed by Google for formatting phone numbers.[5] The project was chosen because it's large enough without being too large, and obviously, it uses Googletest for testing. The project implements both Googletest macros TEST and TEST_F.
Testing the plug-in on Libphonenumber has shown that the plug-in is now able to convert a whole test folder at once, fulfilling the first project goal. The fixture classes could also be converted properly. However, the converted Libphonenumber test project couldn't be compiled. Some problems remained that couldn't be easily solved by hand, showing the limitations of the plug-in in its actual state. Concretely, a further team could work on the following issues:

- The converted project wasn't able to resolve some dependencies, since Libphonenumber uses test classes as C++ "friends" whose names were changed during the conversion. It may be good to find a way to get around this problem.
- Some advanced constructs (like "::testing::AssertionResult")[3] couldn't be converted (this problem concerned only one test file though). A next step would be to look closely at these constructs and see if they're widely used in practice, to decide if they should be supported by the plug-in. Otherwise, some warning should be implemented.
- A further remaining problem was a type error comparing pointers with ints (the number 0 representing nullptr) that happened to be used several times in Libphonenumber. It's not clear if the problem can be solved working on the CUTE plug-in.
- In one test file, a problem occurred because a constructor in a fixture class was wrongly overloaded. In Googletest, fixture classes can implement both constructors and the SetUp()-function to initialize the test object. But CUTE only works with constructors. Therefore, some collisions can occur after the conversion of fixture classes. It may be good to have a closer look at the fixture classes implemented in the Libphonenumber project, to see how often constructors are used in such classes, and if the fixture class conversion should be extended to deal with more complicated cases, maybe even with fixture classes using constructors with arguments.
- Another main limitation of the plug-in is its difficulty handling projects imported over CMake[11]: the file conversion works, but can not be finalized, since the converted targets are only virtual. Researches have shown that CMake is widely used to build and import C++ projects into Eclipse. It would be an improvement to find a way to deal with such projects.
- For the Libphonenumber conversion, all assertions supported by the plug-in were sufficient. Nevertheless, it would be interesting to look at some other real-life projects to see if further assertions should be supported.

Contents

List of Tables

Listings

List of Figures

Glossary

AST	The Abstract Syntax Tree is a representation of the syntactic structure of a source code in a tree form.
CDT	Eclipse CDT (for C/C++ Development Tooling) is an Eclipse extension developed to support the C and C++ programming languages.
CUTE	A C++ testing framework CUTE[7] developed by the IFS.
IFS	The Institute For Software at the University of Applied Sciences Rapperswil, which developped the C++ IDE Cevelop[10] as well as the CUTE[7] testing framework.
ILTIS Core	ILTIS Core is an Eclipse plug-in developed by the IFS and used by the IDE Cevelop.

1 Introduction

The Institute for Software (IFS)[8] has been developing plug-ins for Eclipse CDT (as in "C/C++ Development Tooling")[1] for many years. Students contribute to the development within the scope of their thesis, bachelor or semester projects. This document is the result of a semester project dedicated to the conversion of Googletest[4] to CUTE[7]. The conditions for the project (15 weeks and approximately 240 hours of work) were given by the HSR. The project took place during Spring 2019 and was conducted by one student, Muriel Thévenaz, under the supervision of one technical advisor, Toni Suter, and one main advisor, Peter Sommerlad.

1.1 Premise

One of the main projects at the IFS is a C++ testing framework named CUTE, integrated into the IFS IDE Cevelop[10]. CUTE is a header-only library that supports Test Driven Development (TDD) and always complies with the newest C++ standards, since its maker and supervisor, Prof. Sommerlad, is part of the standardization team. Today, one of the widest used C++ testing frameworks is Googletest.[2] But in contrast to CUTE (see Table 1.1), Googletest relies on macros for automatic test registration, what implies that the program logic is hidden and therefore not very understandable for the developer. Furthermore, Googletest doesn't comply with "agile" quality standards of unit testing by supporting nonfatal failures, what means that it's allowed to write a test that doesn't stop the main testing function from going on even if it fails. Besides, Googletest works as an external library that has to be installed and compiles on its own.

Googletest	**CUTE**
Based on macros	Based on functions (what minimizes the use of macros)
External library	Header-only library
Supports nonfatal failures	Prevents from nonfatal failures Supports TDD

Table 1.1: *Comparison between Googletest and CUTE*

In order to motivate developers to use CUTE instead of Googletest by making it easier to switch to the more modern framework, IFS wants to provide a way of converting existing testing projects from Googletest to CUTE with only a few clicks. The aim of the present student work was to develop exactly such a conversion tool: an Eclipse plug-in that can convert a Googletest project to a CUTE project. Another student team created an initial version of that conversion plug-in before. It was able to convert all main Googletest

assertions, but only for a single file at once, and was therefore not yet suitable for real-life projects since they often have more than one test file. The present student project has built up on this first solution, refactored it where needed, and brought the plug-in one step further toward its first release. Two main features are now supported by the plug-in: the conversion of a whole project at once instead of a single file only, and the conversion of tests with fixtures. The most important remaining limitation is the management of complicated test file structures with a lot of dependencies between the test files and such constructs as C++ "friend" classes.

1.2 Document structure

The following chapters go from the analysis of existing features in Googletest and their counterparts in CUTE, to the discussion of the project results. Next chapter (see chapter 2) presents how Googletest manages testing over a whole project as well as the Googletest construct designed to write tests with fixtures: TEST_F. For both, a possible conversion toward CUTE is discussed. Before going further, another chapter (see chapter 3) presents the features supported by the original plug-in developed before by another student team. After that, the challenges inherent from implementing the conversion over several files as well as for tests with fixtures are discussed (see chapter 4). The solution chosen to solve the problems are also presented in this chapter. Towards the end of the semester, the plug-in was tested on an existing project. The last chapter (see chapter 5) shows, with the help of this test, what the plug-in is able to support and what are its limitations at the end of the semester.

2 Analysis

In order to have the plug-in working on real testing projects, two main features have to be implemented. First, the conversion should work for more than one file at once. Furthermore, the plug-in should be able to handle the conversion of the Googletest construct designed to write tests with fixtures: TEST_F. This chapter discusses the challenges associated with both of those features.

2.1 Multifile structure

Commonly, a test project is composed of several files. Therefore, the conversion plug-in should be able to convert tests spreading over several translation units. This implies traversing the project, including different directories, and find the files that need to be converted. The plug-in also needs to take care of the test registration and the creation of some new files, like a new main file to run the tests as well as a header file for each converted file.

Googletest structure

The following example (see Figure 2.1) illustrates a simple Googletest project structure. It's composed of an "includes" folder that doesn't need to be converted, two other folders that contain test files (their names contain the word "test") as well as other implementation and header files referred to by the test files. Three test files are directly placed in the main folder. One of them use tests with fixtures (TEST_F) and will be discussed in the following chapter. Finally, there's the "main.cpp"-file: the heart of the Googletest project which contains the main()-function (see Listing 2.1). After having initialized the tests, this function invokes the RUN_ALL_TESTS() macro, returning 0 if all tests pass, and 1 otherwise.[4]

```
#include "gtest/gtest.h"

int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Listing 2.1: *Run all tests Googletest function*

Other than in most testing frameworks - and other than in CUTE -, the test registration with Googletest occurs automatically, without having the "main.cpp"-file referencing

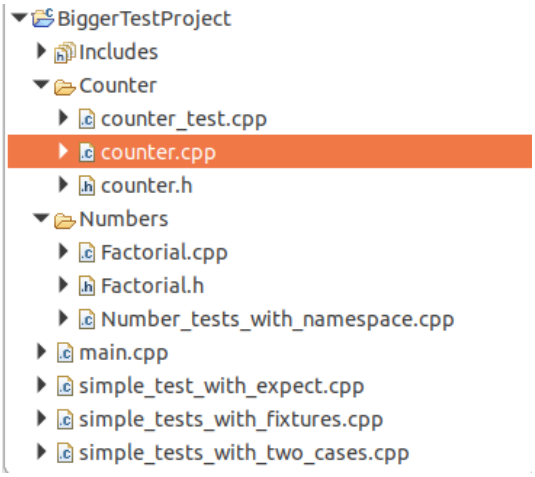


Figure 2.1: *Structure of a simple test project*

the other test files. The tests are neither registered in the test files themselves, the test functions just come one after the other (see Listing 2.2) and the rest happens over macros, so that the programmer doesn't have an insight about how it works exactly. The test functions have the recognizable signature TEST(NameOfTheTestCase, nameOfTheTest), where the NameOfTheTestCase is the name used for the whole suite (Googletest uses the term "case" instead of "suite"), and the nameOfTheTest is the name of the particular test. As illustrated in the example, where some tests belong to the test case "Counter" and others to the test case "OtherCounter", a test file may contain more than one test case.
The test file example also shows some other issues that the conversion plug-in has to take care of at file level (and not project level). First, the includes need to be managed: the "gtest/gtest.h"-include has to be replaced by the "cute.h"-include, while all other includes need to stay unchanged. Furthermore, Googletest uses both fatal (ASSERT) and nonfatal (EXPECT) assertions, while CUTE only uses fatal assertions, since nonfatal assertions are considered bad practice. Both issues were already (partially) solved in the original plug-in (see chapter 3).

CUTE structure

The conversion from a Googletest project towards a CUTE project pursues two goals that are sometimes conflicting. In order to be quickly understandable by the developer operating the conversion, the original file structure should be kept as far as possible. On the other hand, CUTE projects aim to follow some best practices, like having one implementation file and its corresponding header per test suite.[7] Different solutions were discussed for the conversion, going from one single file per test file to be converted, to even four files per converted test case. At the end, one of the simpler solutions was chosen, with one implementation/header-pair per converted file. Other than in the

```
#include "counter.h"
#include "gtest/gtest.h"

TEST(Counter, atInitialization) {
  Counter counter;
  ASSERT_EQ(0, counter.getCounter());
}

TEST(Counter, increment) {
  Counter counter;

  EXPECT_EQ(1, counter.increment());
}

TEST(OtherCounter, decrement) {
  Counter counter;
  counter.increment();
  counter.increment();

  EXPECT_EQ(1, counter.decrement());
}

TEST(OtherCounter, decrementWhenZero) {
  Counter counter;

  ASSERT_EQ(0, counter.decrement());
}
```

Listing 2.2: *Example of a simple test file*

original plug-in (see chapter 3), where a file containing two test cases would be split into two implementation/header-pairs, the new conversion would let the two converted suites in one single implementation/header-pair, with one namespace and test registration function per test suite (see Listing 2.3 and Listing 2.4).

```
#ifndef counter_test_H_
#define counter_test_H_

#include "cute_suite.h"

namespace Counter_cute {

  cute::suite make_suite();

}
namespace OtherCounter_cute {

  cute::suite make_suite();

}

#endif /* counter_test_H_ */
```

Listing 2.3: *Newly created header file for the converted implementation file "counter_test.cpp"*

The conversion plug-in also takes care of creating a new main file for the CUTE project (see Listing 2.5), while the old Googletest main()-function is deleted (if it stayed alone in a file, it means that the file that used to contain the main()-function may be empty at the end of the conversion). The new CUTE main file is created in any case, even if only one test file is converted and the others aren't. In this case, two main files may exist in

```
#include "counter.h"
#include "counter_test.h"
#include "cute.h"


namespace Counter_cute {

void atInitialization_cute();
void increment_cute();

cute::suite make_suite() {
  cute::suite s { };
  s.push_back(CUTE(atInitialization_cute));
  s.push_back(CUTE(increment_cute));
  return s;
}

}

namespace OtherCounter_cute {

void decrement_cute();
void decrementWhenZero_cute();

cute::suite make_suite() {
  cute::suite s { };
  s.push_back(CUTE(decrement_cute));
  s.push_back(CUTE(decrementWhenZero_cute));
  return s;
}

}

void Counter_cute::atInitialization_cute() {
  Counter counter;
  ASSERT_EQUAL(counter.getCounter(), 0);
}

void Counter_cute::increment_cute() {
  Counter counter;

  ASSERT_EQUAL(counter.increment(), 1);
}

void OtherCounter_cute::decrement_cute() {
  Counter counter;
  counter.increment();
  counter.increment();

  ASSERT_EQUAL(counter.decrement(), 1);
}

void OtherCounter_cute::decrementWhenZero_cute() {
  Counter counter;

  ASSERT_EQUAL(counter.decrement(), 0);
}
```

Listing 2.4: *Converted file "counter_test.cpp"*

the project at the end of the conversion: the original one of the Googletest project, and the new one for the newly converted CUTE test file.

```
#include "cute.h"
#include "ide_listener.h"
#include "xml_listener.h"
#include "cute_runner.h"

#include "Counter/counter_test.h"

bool runSuite(int argc, char const *argv[]) {
  cute::xml_file_opener xmlfile(argc, argv);
  cute::xml_listener<cute::ide_listener<>> lis(xmlfile.out);

  auto runner = cute::makeRunner(lis, argc, argv);

  cute::suite s1 = Counter_cute::make_suite();
  cute::suite s2 = OtherCounter_cute::make_suite();

  bool success = runner(s1, "Counter_cute");
  success &= runner(s2, "OtherCounter_cute");

  return success;

  return success;
}

int main(int argc, char const *argv[]) {
    return runSuite(argc, argv) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Listing 2.5: *New main file created for the CUTE project*

2.2 Tests with fixtures (TEST_F)

In software testing, when several tests use the same initial setup to run, it's common to initialize the setup in a separated method (setup), and to clean the system afterwards (teardown). Googletest provides a way to do that with the macro TEST_F. Since it's a widely used feature, it's important that the plug-in supports its conversion.

Fixtures in Googletest

Googletest provides a way of working with test fixtures by using the macro TEST_F instead of TEST. Each TEST_F sharing the same test case (or test suite in CUTE) initializes the same test object before running. When one test is finished, the fixture object is deleted: any test coming after that starts by setting up a fresh new fixture object, so that any change a test operates on a fixture object doesn't affect any other tests.
To implement a Googletest fixture class that a TEST_F would call (see Listing 2.6), four little steps have to be done[4]:

- First, the fixture class has to have the exact same name as the test case.
- Secondly, it has to derive from "::testing::Test" (or "testing::Test").
- Thirdly, the class body has to start with "protected" or "public".
- The fourth and last necessary step is to override the SetUp()-method inside the class, or to write a default constructor for the class.

The fifth step is optional, being to override the TearDown()-method (or implement a destructor) to clean the system after the test.

```
#include "gtest/gtest.h"

#include <queue>

class QueueTest : public ::testing::Test {
 protected:
  virtual void SetUp() {
    q1_.push(1);
    q2_.push(2);
    q2_.push(3);
  }

  // virtual void TearDown() {}

  std::queue<int> q0_;
  std::queue<int> q1_;
  std::queue<int> q2_;
};

TEST_F(QueueTest, isEmptyInitially) {
  ASSERT_EQ(0, q0_.size());
}

TEST_F(QueueTest, popWorks) {
  q2_.pop();
  ASSERT_EQ(1, q2_.size());
}

TEST_F(QueueTest, frontWorks) {
  ASSERT_EQ(2, q2_.front());
}
```

Listing 2.6: *Example of a test using fixtures in Googletest*

Fixtures in CUTE

Like in most testing frameworks, tests with fixtures are also supported by CUTE. There are three kinds of tests with fixtures in CUTE. The one using the macro CUTE_SMEMFUN (TestClass,test2) during test registration (see Listing 2.7) is the most similar to the Googltest TEST_F macro, since it "creates a new instance of TestClass to then call its member function test2 when the test is executed".[6]
Like in Googletest, the fixture class has to share its name with the test suite. Other than that, there are three main differences with Googletest fixture classes:

- The class (or struct) doesn't have to derive from any other class.
- All class members have to be public in order to be used by the tests.
- Only constructor and destructor can be used for setup and teardown.

Like in Googletest, the teardown is optional, since any class in C++ comes with a default destructor. Same as any other test in CUTE, it has to be registered. Other differences between the Googletest and the CUTE version of the example (see Listing 2.6 and Listing 2.7) - like the new namespace "QueueTest_cute" - aren't prerequisites for CUTE fixtures test, but the result of some implementation choices (see section 4.2).

```
#include "cute.h"
#include "simple_tests_with_fixtures.h"


#include <queue>

namespace QueueTest_cute {

struct QueueTest {
  QueueTest() {
    q1_.push(1);
    q2_.push(2);
    q2_.push(3);
  }

  std::queue<int> q0_;
  std::queue<int> q1_;
  std::queue<int> q2_;
  void isEmptyInitially_cute();
  void popWorks_cute();
  void frontWorks_cute();
};

cute::suite make_suite() {
  cute::suite s { };
  s.push_back(CUTE_SMEMFUN(QueueTest, isEmptyInitially_cute));
  s.push_back(CUTE_SMEMFUN(QueueTest, popWorks_cute));
  s.push_back(CUTE_SMEMFUN(QueueTest, frontWorks_cute));
  return s;
}

}

void QueueTest_cute::QueueTest::isEmptyInitially_cute() {
  ASSERT_EQUAL(q0_.size(), 0);
}

void QueueTest_cute::QueueTest::popWorks_cute() {
  q2_.pop();
  ASSERT_EQUAL(q2_.size(), 1);
}

void QueueTest_cute::QueueTest::frontWorks_cute() {
  ASSERT_EQUAL(q2_.front(), 2);
}
```

Listing 2.7: *Example of the same test as before using fixtures in CUTE*

3 Original Plug-in

Since the present "Googletest to CUTE"-conversion project builds on an earlier student project,[15] the implementation work didn't start from scratch. The original plug-in offered basic conversion functionalities built on choices made by the other team. It also showed some bugs and problems that had to be solved in order to make the plug-in more robust, or just to go on with the new implementation steps.

3.1 Functionalities from the original plug-in

The original plug-in was able to convert a single Googletest test file with one or more test cases (or test suites in CUTE) into a CUTE test project, implementing some basic functionality.

Conduct the conversion

A right click on the file to be converted allowed the user to choose the appropriate functionality in the menu (see Figure 3.1). The changes were then visualized and a "Finish" button had to be pressed for the conversion to be concluded. After the conversion, the original Googletest file didn't exist anymore, but it was still possible to go back to it with the commonly used "undo" Eclipse functionality. Before the new CUTE test could be executed, the user had to make sure that the CUTE Nature was added to the project (see Figure 3.2).

New file structure

The conversion started with one file, but if this file contained more than one test case (or test suite in CUTE), a new project structure was created during the conversion in order to comply with the CUTE best practice "one implementation file and its corresponding header per test suite".[7] Since CUTE (unlike Googletest) requires that the tests are registered for them to be run, the plug-in took over the registration work: first at the suite-level, and then at the project-level (in the file named "CuteMain.cpp", see Figure 3.3).

Implemented assertions

Most of the basic assertions were supported by the original plug-in (see Table 3.1). Further on, since non fatal assertions (EXPECT-assertions in Googletest) are not supported by CUTE and are considered to be bad testing practice, the team behind the original

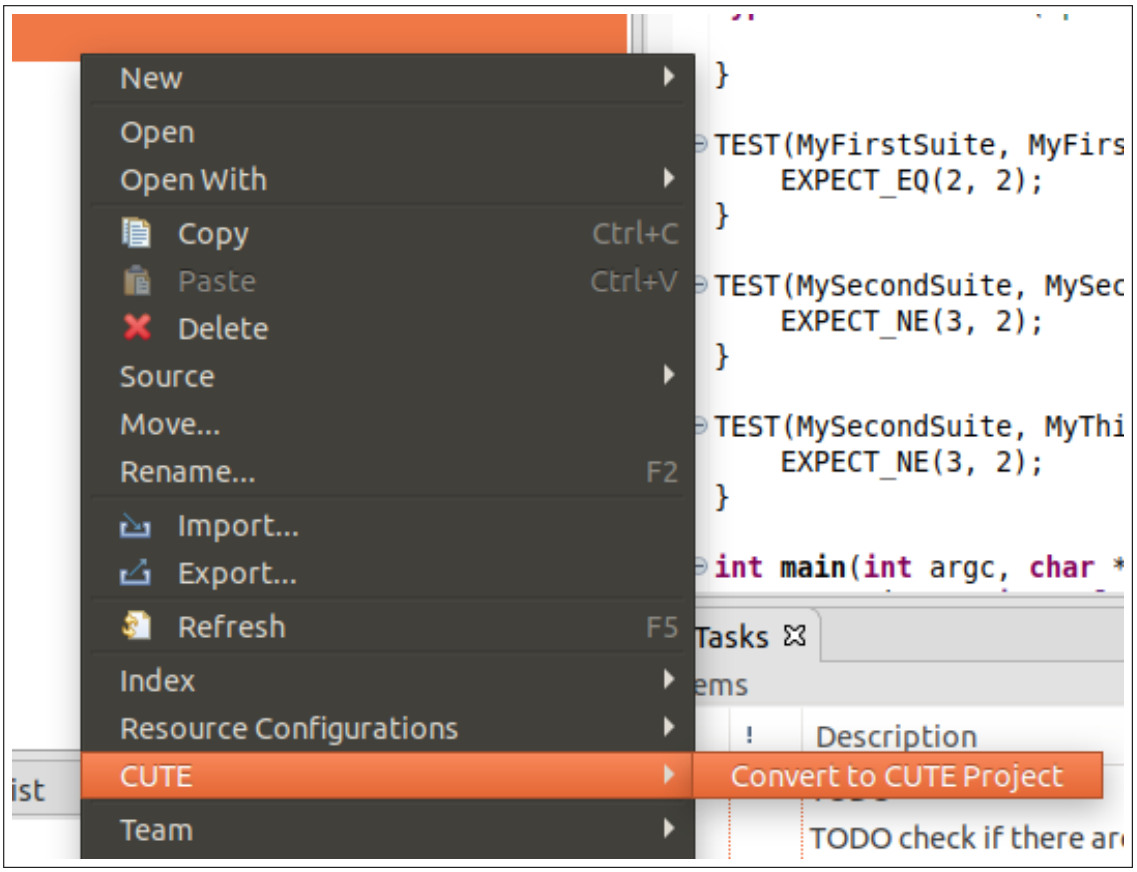


Figure 3.1: *Starting the conversion process in Eclipse*

plug-in decided to convert them to fatal assertions (ASSERT-assertions in Googletest). They also defined some basic conversion rules for mapping assertions that don't exist in CUTE.[15][p.26]

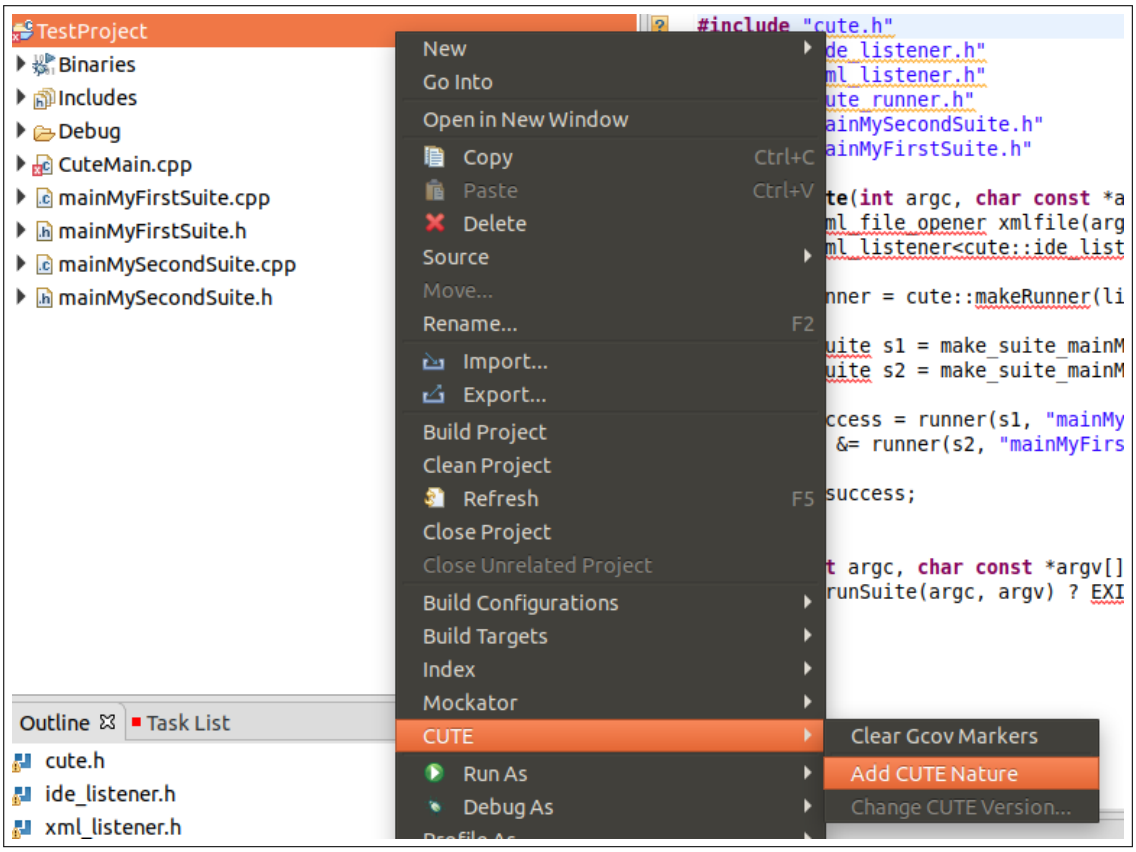


Figure 3.2: *Add CUTE headers to the project*

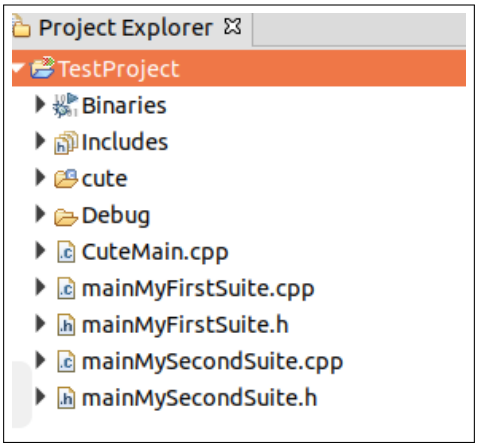


Figure 3.3: *Project structure after the conversion*

Googletest	CUTE
ASSERT_TRUE	ASSERT
ASSERT_FALSE	ASSERT
ASSERT_STREQ	ASSERT_EQUAL
ASSERT_STRNE	ASSERT_NOT_EQUAL_TO
ASSERT_EQ	ASSERT_EQUAL
ASSERT_NE	ASSERT_NOT_EQUAL_TO
ASSERT_LT	ASSERT_LESS
ASSERT_GT	ASSERT_GREATER
ASSERT_LE	ASSERT_LESS_EQUAL
ASSERT_GE	ASSERT_GREATER_EQUAL
ASSERT_THROW	ASSERT_THROWS
ASSERT_FLOAT_EQ	ASSERT_EQUAL
ASSERT_DOUBLE_EQ	ASSERT_EQUAL
ASSERT_NEAR	ASSERT_EQUAL_DELTA

Table 3.1: *Assertions supported by the original plug-in*

3.2 Technical details

The heart of the original plug-in was the GTestRefactoring-class (see Figure 3.4), which extended the CDT refactoring class CRefactoring, that comes with a lot of functionalities to conduct AST (as in Abstract Syntax Tree, the representation of code as a tree) based refactoring. Starting the conversion over the CUTE menu instantiated CTestRefactoring over the GTestRefactoringHandler, a class extending the RefactoringStarterMenuHandler-class from the ILTIS Core[16] and providing a simple way to start a CRefactoring with a refactoring wizard. Depending if a test file contained one or more suites to convert, a different manager (ConverterManager or SuiteConverterManager) helped with the creation of new code. The GTestMacroParser-class took over the parsing of the AST nodes to convert (TEST and assertions). Finally, two checkers prevented the refactoring from starting if not supported assertions or TEST_F macros were found in the translation unit.
To perform the conversion, the authors of the original plug-in implemented a pattern widely used for this kind of work: the visitor pattern. The basic idea is to have one object traversing a given structure (like a Googletest file) in order to find the places in the code that need to be changed, and then another one performing the actual changes.

> With the Visitor pattern, you define two class hierarchies: one for the elements being operated on (the Node hierarchy) and one for the visitors that define operations on the elements (the NodeVisitor hierarchy). You create a new operation by adding a new subclass to the visitor class hierarchy. As long as the grammar that the compiler accepts doesn't change (that is, we don't have to add new Node subclasses), we can add new functionality simply by defining new NodeVisitor subclasses.[14][p.333]

In the plug-in, the structure visited is the AST of the Googletest file to be converted.

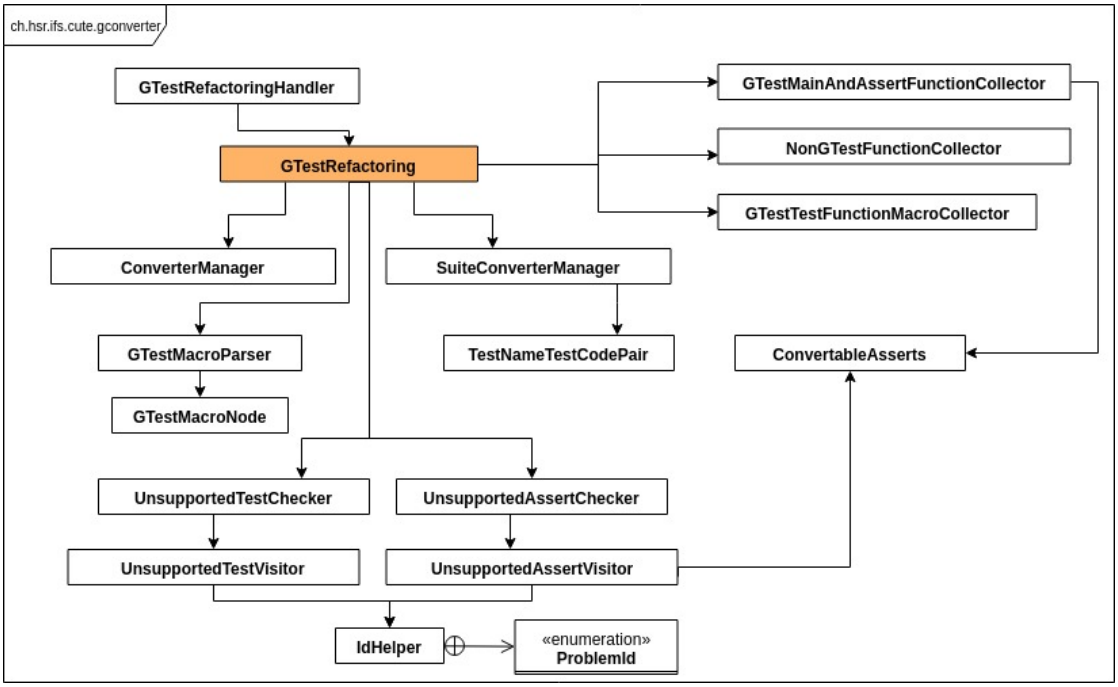


Figure 3.4: *Class diagram from the original conversion plug-in*

Altogether, the original plug-in used six visitors:[15][pp.6-7]

- One looking for the TEST macros
- One looking for the ASSERT/EXPECT statements
- One looking for the namespaces
- One looking for other definitions
- Two visitors reporting problems

In the original plug-in, the actual rewriting of the AST operated differently if a file contained one or more than one test case (or test suite in CUTE). With only one test case, it was possible to rewrite directly in the file with the ASTRewrite-class from the CDT. For the more complicated scenario, this simple approach didn't work though, since there's only one AST per file and it's not possible to just split it into different files. Therefore, the team used templates to define the correct structure of the files and the ASTWriter-class to write the new code into the template.

3.3 Bugs in original plug-in

Over the course of the project, some bugs were discovered in the original plug-in. Some of them could be resolved, others just weren't an issue anymore after having made some

fundamental changes in the project structure.

Resolved bugs

Check for the main()-function too specific

The method testing if a node is the main()-function operated with string comparison, only matching if the main function had the exact signature "main (int argc, char **argv)" (see Listing 3.1). Any other notation - even a space placed differently - didn't match. This method was therefore very error prone, which was especially problematic since it's an essential method in the program.

```
private boolean isMainFunction(IASTDeclarator declarator) {
  if (declarator instanceof IASTFunctionDeclarator && declarator.getRawSignature().startsWith("main(int argc
      , char **argv)")) {
    return declarator.getParent().getParent() instanceof IASTTranslationUnit;
  }
    return false;
}
```

Listing 3.1: *Method testing if a node is the main()-function in the Googletest file* ***before*** *the bug fix*

Solution: The method doesn't check the exact signature anymore, but if the name of the function is "main" (see Listing 3.2).

```
private boolean isMainFunction(IASTDeclarator declarator) {
  return declarator instanceof IASTFunctionDeclarator && declarator.getName().toString().equals("main") &&
      isRootMain(declarator);
}

private boolean isRootMain(IASTDeclarator declarator) {
  return declarator.getParent().getParent() instanceof IASTTranslationUnit;
}
```

Listing 3.2: *Method testing if a node is the main()-function in the Googletest file* ***after*** *the bug fix*

Googletest header not properly erased

After the conversion, the Googletest header "gtest/gtest.h" didn't appear in the newly created "CuteMain.cpp"-file, but wasn't successfully removed from the original file. It was a problem, since the whole Googletest library was then compiled when running the CUTE tests. It meant not only unnecessary work and unused storage, but also led to problems like calling Googletest instead of CUTE assertions.
Solution: It wasn't easy to solve the problem, since preprocessor statements are not visited in an AST. Toni Suter, technical advisor for the project, wrote the helper class

IncludeManager, which works with mutable lists and is able to remove headers according to a given pattern (see Listing 3.3).

```
private Optional<TextEdit> createIncludeRemovalEdit(String filePath) {
  MutableList<Pattern> includeDirectivePatterns = includesToRemove.get(filePath);
  IASTTranslationUnit ast = astCache.get(filePath);
  if(includeDirectivePatterns.isEmpty() || ast == null) {
    return Optional.empty();
  }

  MultiTextEdit multiTextEdit = new MultiTextEdit();
  for(IASTPreprocessorStatement stmt : ast.getAllPreprocessorStatements()) {
    if(!(stmt instanceof IASTPreprocessorIncludeStatement)) {
      continue;
    }

    IASTPreprocessorIncludeStatement includeStmt = (IASTPreprocessorIncludeStatement)stmt;
    String includeName = includeStmt.getName().toString();
    if(includeDirectivePatterns.anySatisfy(pattern -> pattern.matcher(includeName).matches())) {
      DeleteEdit deleteEdit = new DeleteEdit(includeStmt.getFileLocation().getNodeOffset(), includeStmt.
          getFileLocation().getNodeLength());
      multiTextEdit.addChild(deleteEdit);
    }
  }

  return Optional.of(multiTextEdit);
}
```

Listing 3.3: *Method removing a header according to a given pattern*

Short name for suites doesn't increase in main

In the newly created "CuteMain.cpp"-file, responsible for running the different test suites, the short name created automatically to register the suites didn't increase properly. The problem was that the incrementation happened within a lambda, and that in Java, any variable used in a lambda is final per definition: the variable can't be changed.

```
private String buildSuiteRunners(String fileName, Set<String> suites) {
  StringBuilder runnerString = new StringBuilder();
  String firstSuite = suites.stream().findFirst().get();
  String firstSuiteShortName = "s1";
  runnerString.append("bool success = runner(").append(firstSuiteShortName).append(", \"").append(fileName).
      append(firstSuite).append("\");").append(LINESEP);

  suites.stream().skip(1).forEach((suite) -> {
    int suiteCounter = 2;
    String suiteShortName = "s" + suiteCounter++;
    runnerString.append(TAB).append("success &=runner(").append(suiteShortName).append(", \"").append(
        fileName).append(suite).append("\");").append(LINESEP);
  });

  return runnerString.toString();
}
```

Listing 3.4: *Lambda building the main CUTE suite runner* ***before*** *the bug fix*

Solution: The whole suite registration was changed to a for-loop, so that the "suiteCounter"-variable could be increased successfully (see Listing 3.5).

```
public static String buildSuiteRunnersForMain(Set<String> suites) {
  StringBuilder runnerString = new StringBuilder();

  int suiteCounter = 1;
  for (String suite : suites) {
    String[] splittedQualifiedName = suite.split("::");
    String suiteName = splittedQualifiedName[(splittedQualifiedName.length-1)];
    String suiteShortName = "s" + suiteCounter;

    if (suiteCounter == 1) {
      runnerString.append("bool success = runner(");
    } else {
      runnerString.append(TAB).append("success &= runner(");
    }
    runnerString.append(suiteShortName).append(",\"").append(suiteName).append("\");").append(LINESEP);
    suiteCounter++;
  }
  return runnerString.toString();
}
```

Listing 3.5: *"For-loop" building the main CUTE suite runner* ***after*** *the bug fix*

Return type for the test functions wrongly defined

While Googletest TEST macros don't have any return type, CUTE test functions have one (void). In the original plug-in, this return type was simply added as String to the function declarator (see Listing 3.6), instead of being added as return type (which aren't part of a function signature). The conversion worked, but the newly created AST was inaccurate.

```
ICPPASTFunctionDeclarator decl = factory.newFunctionDeclarator(TEST_RETURN_TYPE + testName);
if (node instanceof IASTFunctionDefinition) {
  IASTFunctionDeclarator declarator = ((IASTFunctionDefinition) node).getDeclarator();
  rewriter.replace(declarator, decl, group);
  testsToRegister.add(testName);
}
```

Listing 3.6: *Conversion of the test function definition* ***before*** *the bug fix*

Solution: In order to obtain the right structure after the conversion, a separate node is now created for the return type. This node is then added just before the function declarator, as first child of the function definition node (see Listing 3.7).

Bugs solved through structural changes in the project

NullPointerException when main is not found

In the original plug-in, when the conversion started but the program didn't find the main()-function - either because it didn't exist, or because the signature didn't match -, the Program threw a NullPointerException and didn't go further. It would have been better to have a more defensing programming with a default behavior implemented for this case.

```
if (node instanceof IASTFunctionDefinition) {
  ICPPASTFunctionDeclarator newDeclarator = factory.newFunctionDeclarator(" " + suiteName + "::" + testName)
      ;
  rewriter.replace(((IASTFunctionDefinition) node).getDeclarator(), newDeclarator, group);

  IASTSimpleDeclSpecifier voidSpecifier = factory.newSimpleDeclSpecifier(IASTSimpleDeclSpecifier.t_void);
  rewriter.insertBefore(node, ((IASTFunctionDefinition) node).getDeclarator(), voidSpecifier, group);
}
```

Listing 3.7: *Conversion of the test function definition **after** the bug fix*

Solution: In the new plug-in, the conversion works with and without a main()-function. If a main()-function is found, it's just deleted, since a new one is created in any case.

Code replication among created files

All constructs that weren't the actual test functions (variables, other functions, and so forth) in the Googletest file were packed in an anonymous namespace that appeared in each newly created test suite file.[15][p. 34] That led to code replication among the files, and possibly even to naming collisions.
Solution: In the new plug-in, the files are not split anymore. Therefore, the non test constructs don't need to change anymore.

Main()-function considered as non test construct

This bug was similar to the "Check for the main()-function too specific" one, since it was also caused by a too specific check deciding if a function was the main function or not. However, the bug resulted in a different problem: the main function was not recognized as such and landed in the anonymous namespace created for all non test constructs found in the original file.
Solution: In the new plug-in, the test constructs don't need to be sorted out and the problem is not an issue anymore.

4 Implementation

The following chapter discusses the choices made while implementing both the conversion for more than one test file at the time, as well as the conversion from the Googletest TEST_F construct. The plug-in was also supplemented with warning notices, generated to help the user better understand what happened when something goes wrong (for instance because some assertions are not supported by the plug-in).

4.1 Project Overview

Before going into the implementation details, the following class diagram should give an overview of the actual plug-in (see Figure 4.1). Like the original plug-in (see section 3.2), when the conversion process is launched over the CUTE menu, a handler (the GTestRefactoringHandler-class) starts the actual conversion with a wizard (now as nested class, since the old handler class couldn't be used anymore, see section 4.2). Some classes from the original plug-in disappeared, primarily the ones handling the non-support of TEST_F macros in the original plug-in. And new ones appeared to implement the new functionalities (exceptions, classes supporting the conversion of the TEST_F macros, etc.). Since nothing was changed on the conversion process of the assertions, the code handling it stayed the same. It just moved to different classes (for instance, the visitor looking for the main()-function was separated from the one searching for the assertions, and the manager classes were restructured). The classes handling the parsing of the nodes also stayed (almost) the same.
In the class diagram, the heart of the actual plug-in is highlighted in orange. A focus on these classes shows them with all theirs methods and variables (see Figure 4.2). The most relevant methods and variables are, again, highlighted in orange.

- The GTestProjectRefactoring-class is responsible for the conversion at project level: It collects all the translation units that might need to be converted within the project as well as all changes generated during the conversion process. At the end of the process, it also actually creates the changes, allowing them to show in the wizard (and eventually, to be finalized).
- The SingleFileRefactoring-class takes care of the conversion at file level, visiting the file to find the nodes to convert, registering the tests and creating a new header for the file.
- The ProjectConversionManager-class helps creating the "Cute_Main.cpp"-file to run the tests.

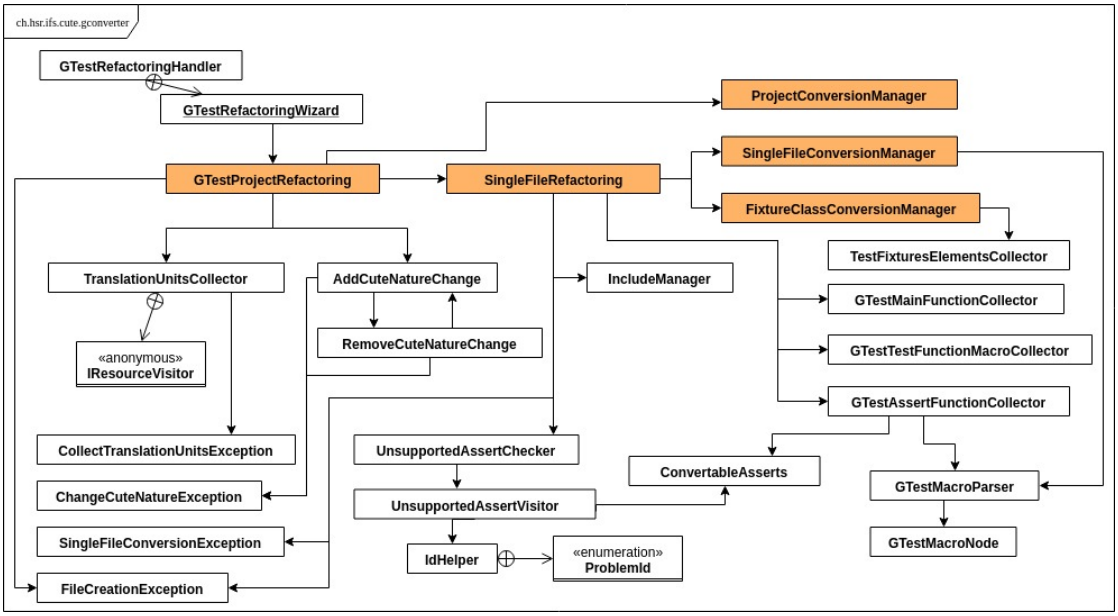


Figure 4.1: *Class diagram from the conversion plug-in*

- The SingleFileConversionManager-class takes care of the actual conversion work directly on the AST, converting the required nodes and creating new ones.
- The FixtureClassConversionManager-class is responsible for the conversion of the fixture class when TEST_F macros are found in the translation unit.

The same way as in the original plug-in, templates help the creation of the new headers and the "Cute_Main.cpp"-file (see section 3.2).

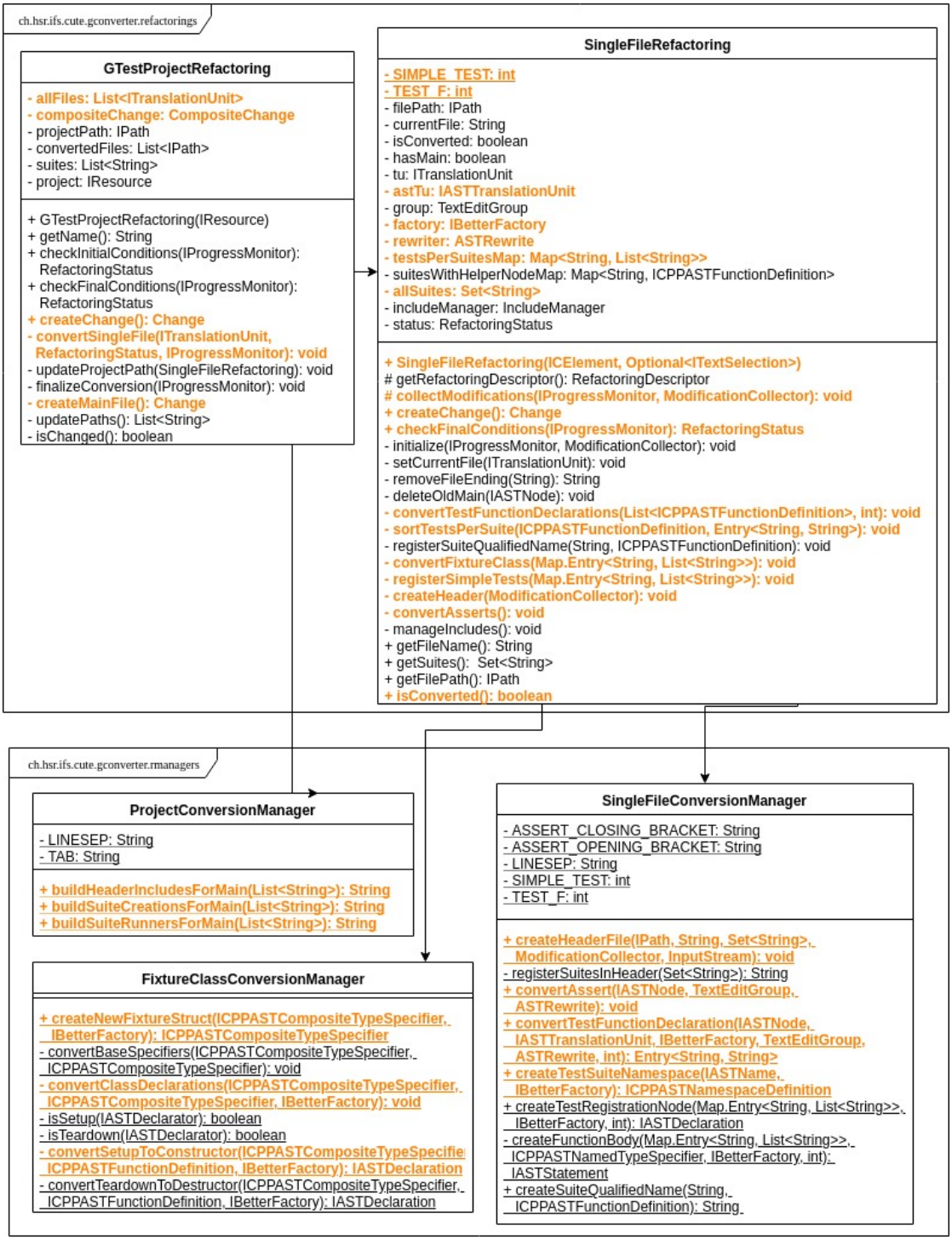


Figure 4.2: *Focus on the main classes of the conversion plug-in*

4.2 Multifile structure

In order to support the conversion of several files (or a whole project) at once, some fundamental changes had to be made in the original plug-in.

Start the conversion over a folder

In the original plug-in, the conversion could be started with a right click on the file to be converted (see chapter 3). A similar way of converting several files had to be implemented to let the user choose which files or folders he wants to convert. The start at project level also had to be supported.
Solution: The "plugin.xml"-file for the conversion plug-in (and not for the whole CUTE plug-in) was changed to extend the already existing CUTE menu with a new entry (see Listing 4.1). In order to have the menu appearing at project, folder and file level as well, the most general type of resource (being IResource) was set as "adapt type". The handler launched by the menu had to be changed too to work with all types of resources, and not only files (or more precisely: translation units).

```
<extension point="org.eclipse.ui.menus"
  id="ch.hsr.ifs.cute.gconverter.cuteMenuExtension"
  name="GConverter CUTE Menu Extension">
  <menuContribution locationURI="popup:ch.hsr.ifs.cute.ui.menuafter=ch.hsr.ifs.cute.ui.commands.
       AddRemoveCuteNatureCommand">
  <separator name="ch.hsr.ifs.cute.gconverter.separator1" visible="true">
  </separator>
  <command commandId="ch.hsr.ifs.cute.gconverter.ReplaceTestsCommand"
    label="Convert GTest to CUTE"
    style="push">
    <visibleWhen>
      <with variable="activeMenuSelection">
      <iterate ifEmpty="false" operator="and">
        <adapt type="org.eclipse.core.resources.IResource">
                        </adapt>
                  </iterate>
                </with>
              </visibleWhen>
  </command>
  </menuContribution>
</extension>
```

Listing 4.1: *Extract of the "plugin.xml"-file handling the menu entry to start the conversion*

Collect all files to convert

To have the conversion working properly, only files that may contain Googletest constructs had to go through the refactoring process. Since the selection wasn't done in the "plugin.xml" file to have the menu working at all levels (see section 4.2), it had to be taking care of another way. The difficulty was to sort out files like those in the "Debug" folder that would compromise the refactoring process.
Solution: The solution was given by the technical advisor Toni Suter, who came up with the idea of working with a resource visitor (see Listing 4.2). Instead of visiting

one single file, like any other visitor used in the project, this visitor would traverse the whole project and look for the right resources. Only translation units and no header files are collected by the visitor. To ignore the "Debug"-folder, any derived resource is just skipped. The test if a file contains any Googletest construct or not is left to the class handling the refactoring at file level (see section 4.2).

```
project.accept(new IResourceVisitor() {
  @Override
  public boolean visit(IResource resource) {

    // ignore the Debug folder
    if(resource.isDerived()) {
      return false;
    }

    ICElement element = resource.getAdapter(ICElement.class);

    if(element == null) {
      return false;
    }

    if(element instanceof ITranslationUnit && !((ITranslationUnit)element).isHeaderUnit()) {
      files.add((ITranslationUnit) element);
    }
    return true;
  }
}
```

Listing 4.2: *Extract of the TranslationUnitsCollector-class that collects all files for the conversion*

Simplify the project structure after the conversion

The original plug-in created a new header/implementation pair per converted test suite (see chapter 3). In a bigger project containing lots of suites, and maybe a few per test file, this approach would have let to a file explosion and the user may have get confused. The structure after the conversion had to be simplified to stay similar to the structure before the conversion. The less changed the better, even if some basic best practices about testing had to be (partially) ignored (like having only one assertion per test, or only one test suite per header/implementation pair). In the end, these practices weren't applied by the user from start on. A working conversion with a result that is easy to understand for the user had to stay the highest priority.
Solution: In order to stay as close as possible from the original project structure, each converted file was only complemented with a header file. The plug-in erases the Googletest main function if it finds one, but doesn't delete the file that used to contain it. Towards the end of the conversion process, a "Cute_Main.cpp"-file is created to register all test suites to run the CUTE tests. Within a converted file, for each suite, all new constructs needed in CUTE - like the test registration - are put in a newly created namespace (see Listing 4.3) named by the test suite. To avoid name collisions, the suffix "_cute" is added at the end of each test name and namespace. It was decided to create a namespace per test suite to clarify the new file structure for the user after the conversion, and to avoid long names.

```
namespace MySuite_cute {

void myTest_cute();

cute::suite make_suite() {
  cute::suite s { };
  s.push_back(CUTE(myTest_cute));
  return s;
}
```

Listing 4.3: *Example of a newly created namespace that contains the test registration for a CUTE suite*

Handle the whole process as one single refactoring

The original plug-in worked with a refactoring class (CRefactoring) specialized in the refactoring of single translation units (see chapter 3). Handler and wizard were also working closely with this specialized class. To be able to handle the refactoring of a whole project as one single refactoring visualized in a single wizard and finalized as single step (that could be undone in only one step too), this original structure had to be changed.
Solution: To deal with this problem, a new refactoring class handling the whole project was created. Handler and wizard had to be changed too in order to work with this new refactoring class (which doesn't extend CRefactoring, a refactoring class that only works with single files). The refactoring class handling the whole project collects the files to be converted and passes them one by one to the originally implemented CRefactoring class (see Listing 4.4), which therefore takes care of the conversion at file level. It would have been more elegant to avoid calling directly such basic refactoring methods as checkFinalConditions() and createChange(), since these aren't meant to be explicitly called. However, this would have imply not working at all with the CRefactoring class and create the whole refactoring context by hand. This seemed disproportionate.

Automate the step adding the CUTE Nature

In the original plug-in, the CUTE Nature had to be added manually (see chapter 3). It seemed necessary to automate this step that would easily get forgotten.
Solution: Adding the CUTE Nature was implemented as change that would appear as such in the wizard, so that the user would see it and be able to select or unselect it. The AddCuteNatureChange-class implementing the change is an extension of the ResourceChange-class. Another class removing the CUTE Nature had to be implemented as well (see Figure 4.1) to allow undoing the change.

```
private void convertSingleFile(ITranslationUnit file, RefactoringStatus status, IProgressMonitor pm) throws
     SingleFileConversionException {
  CRefactoringContext context = null;
  try {
    SingleFileRefactoring singleFileRefactoring = new SingleFileRefactoring(file.getWorkingCopy(), Optional.
         empty());
    context = new CRefactoringContext(singleFileRefactoring);

    status.merge(singleFileRefactoring.checkFinalConditions(pm));

    Change fileChange = singleFileRefactoring.createChange(pm);
    if (!(fileChange instanceof NullChange)) {
      updateProjectPath(singleFileRefactoring);
      if (singleFileRefactoring.isConverted()) {
        IPath headerPath = singleFileRefactoring.getFilePath().append(singleFileRefactoring.getFileName()).
            addFileExtension("h");
        convertedFiles.add(headerPath);
        Set<String> fileSuites = singleFileRefactoring.getSuites();
        for (String suite : fileSuites) {
          if (suites.contains(suite)) {
            throw new SingleFileConversionException("File \"" + file.toString() + "\":\tThe file contains a
                test suite named \"" + suite + "\", which is allready registred for another test file");
          }
        }
        suites.addAll(fileSuites);
      }
      compositeChange.add(fileChange);
    }
  } catch (OperationCanceledException | CoreException e) {
    Activator.logMsg(e.getMessage());
    throw new SingleFileConversionException("File \"" + file.toString() + "\":\tAn unknown problem prevented
        the conversion.");
  } finally {
    context.dispose();
  }
}
```

Listing 4.4: *Method converting one single file by calling the SingleFileRefactoring-class*

4.3 Tests with fixtures (TEST_F)

To deal with TEST_F, the plug-in had to be able to change where needed each file using test with fixtures. That implied working with the same technics as the original plug-in did, being the visitor pattern and the rewriting of the AST (see section 3.2).

Collect the tests with fixtures

To be able to convert test with fixtures, the first step was to find the TEST_F macros that would indicate the presence of a fixture class.
Solution: The original plug-in already implemented a visitor looking for TEST_F macros, but this one was used to mark the lines where these macros appeared. To simplify the whole process, the visitor looking for the TEST macro was just complemented to collect TEST_F functions too. Since the conversion of fixtures didn't just imply some additional work, but also a different way of registering the tests, both TEST and TEST_F functions were collected separately.

Convert the fixture class

Converting the fixture class wasn't trivial. According to the analysis comparing fixture tests in Googletest and in CUTE (see section 2.2), three main changes had to be done:

- Delete the "::testing::Test" (or "testing::Test") from the class declaration;
- Remove the "protected" at the beginning of the class;
- Convert the SetUp()-(and TearDown())-method(s) to a constructor (or a destructor).

To be able to do that, the first step was obviously to find the fixture class within the file. **Solution:** To find the fixture class, a visitor looking for the right node was implemented (see the TestFixturesElementsCollector-class in). Then, it was decided to create a new "struct"-node with the same name as the original fixture class to ease the conversion, amongst others regarding to the removing of the "::testing::Test" declarator node. All elements of the original node - except the "::testing::Test" declarator node and all access modifiers - had to be copied into the new node. During this process, the SetUp() or TearDown() declarator - if found - would be changed to a constructor or destructor named, as it should, like the newly created struct. In the end, the newly created node would be put in the namespace containing all the newly created CUTE constructs (see Listing 4.5).

```
private void convertFixtureClass(Map.Entry<String, List<String>> suite) {
  TestFixturesElementsCollector fixturesCollector = new TestFixturesElementsCollector(suite.getKey());
  astTu.accept(fixturesCollector);

  ICPPASTCompositeTypeSpecifier fixtureClass = fixturesCollector.getClassNode();

  if (fixtureClass != null) {

    ICPPASTNamespaceDefinition testSuiteNamespace = SingleFileConversionManager
        .createTestSuiteNamespace(fixtureClass.getName().copy(), factory);
    ASTRewrite subrewrite = rewriter.replace(fixtureClass.getParent(), testSuiteNamespace, group);

    ICPPASTCompositeTypeSpecifier newFixtureStruct = FixtureClassConversionManager
        .createNewFixtureStruct(fixtureClass, factory);
    IASTNode newFixtureNode = SingleFileConversionManager.declareTests(suite.getValue(), newFixtureStruct,
        factory, TEST_F);

    IASTDeclaration testRegistration = SingleFileConversionManager.createTestRegistrationNode(suite, factory
        ,
        TEST_F);

    testSuiteNamespace.addDeclaration(testRegistration);

    IASTSimpleDeclaration newFixtureNodeDeclaration = factory
        .newSimpleDeclaration((ICPPASTCompositeTypeSpecifier) newFixtureNode);

    subrewrite.insertBefore(testSuiteNamespace, testRegistration, newFixtureNodeDeclaration, group);
  }
}
```

Listing 4.5: *Method converting the Googletest fixture class into a CUTE fixture struct*

Figure 4.1

Find overlapping between TEST and TEST_F conversion

Even if the conversion of TEST and TEST_F macros implied some differences, both share most of the conversion steps. To avoid repeating code, a simple way to mark the difference only where needed had to be thought of.
Solution: The class handling the refactoring at single file level works with two tags: SIMPLE_TEST and TEST_F (see Figure 4.2). These tags are used to split the common processes only were needed. Since the visitor collecting the test functions also separates the TEST from the TEST_F functions (see section 4.3), some steps can also be targeted by simply working with the appropriate test list.

4.4 Error handling

The original plug-in didn't implement any serious error handling: any problem would lead to the interruption of the conversion process, and it'd be difficult for the user to understand the reason of the interruption. This had to be changed. If the problem was of severe nature, the conversion had to be able to pull through. In case of fatal errors, the cause had to be better understandable for the user.

Log all error messages

The first step to deal with any problems was to log the exception and error messages in the console.
Solution: This was easy to implement, since a LogMsg()-method already existed for the CUTE plug-in. It only had to be implemented in the Activator-class (see Figure 4.1) of the conversion plug-in.

Inform user about fatal errors

In case of fatal errors (for example if a header file would be created, even if another header file with the same name would already exist), the user had to be informed about the cause of the interruption.
Solution: In the handler class of the plug-in, an error dialog that would pop up in case of interruption was implemented (see Figure 4.3). The original error message would appear in the dialog too.

Warn user about small problems

Any small problem that wouldn't corrupt the whole conversion process, but that the user should still be aware of, had to be handled properly. **Solution:** All known problems were registered as warnings that would appear in the wizard if arising (see section 5.3). This way, these problems could easily by handled manually by the user, without standing in the way of converting the whole project.

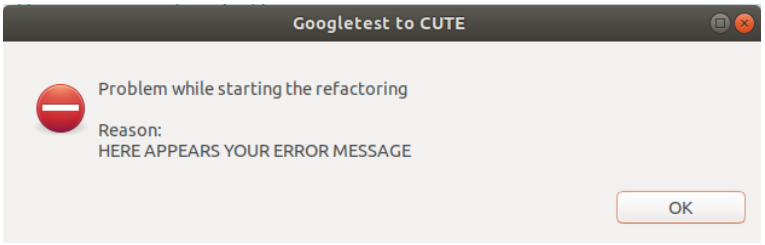


Figure 4.3: *Dialog showing fatal errors*

5 Results

Towards the end of the project, the conversion plug-in was tested on an existing project found in the Internet: Libphonenumber, which is a Java, JavaScript and C++ library developed by Google for formatting phone numbers.[5] The project was chosen because it's large enough without being too large, and obviously, it uses Googletest for testing. The project implements both Googletest macros TEST (without fixture) and TEST_F (with fixtures). Testing the plug-in on Libphonenumber has shown what the plug-in - in its actual state - is able to handle, and what are its limitations.

5.1 Achievements

At project start, two main goals were defined for the scope of the project: make the conversion of a whole project at once possible, and implement a conversion for the TEST_F macro. Both goals could be achieved. Concretely, the plug-in is now able to:

- traverse a project to collect all implementation files that may contain Googletests;
- handle the refactoring of a whole project at once;
- convert the TEST macro (this was already implemented in the original plug-in, see chapter 3);
- convert the TEST_F macro and the fixture class to its CUTE counterpart;
- change the test names by adding the prefix "_cute" to avoid some naming collisions;
- convert all main assertions (this was also already implemented in the original plug-in, see Table 3.1);
- convert files containing both TEST and TEST_F macro together (as long as they don't share the same suite name);
- register the tests within the converted files;
- create a header file per each converted file to comply with the CUTE best practice of having one implementation/header pair per test suite (however, depending on the original Googletest file, more that one test suite may appear in one implementation/header pair);
- create a new "Cute_Main.cpp"-file to run the CUTE tests;

- add the “CUTE Nature” automatically;
- show all changes in a wizard before finalizing the refactoring;
- signal if an assert is not supported by the plug-in, indicating which file is concerned;
- warn the user about other problems to help him solve them (if the problem is not of severe nature, the conversion can be pulled through);

5.2 Limitations

In the end, the converted Libphonenumber test project couldn't be compiled. Some problems remained that couldn't be easily solved by hand, showing the limitations of the plug-in in its actual state. Concretely, a further team could work on the following issues:

- The converted project wasn't able to resolve some dependencies, since Libphonenumber uses test classes as C++ “friends” whose names were changed during the conversion. It may be good to find a way to get around this problem.
- Some advanced constructs (like “::testing::AssertionResult”)[3] couldn't be converted (this problem concerned only one test file though). A next step would be to look closely at these constructs and see if they're widely used in practice, to decide if they should be supported by the plug-in. Otherwise, some warning should be implemented.
- A further remaining problem was a type error comparing pointers with ints (the number 0 representing nullptr) that happened to be used several times in Libphonenumber. It's not clear if the problem can be solved working on the CUTE plug-in.
- In one test file, a problem occurred because a constructor in a fixture class was wrongly overloaded. In Googletest, fixture classes can implement both constructors and the SetUp()-function to initialize the test object. But CUTE only works with constructors. Therefore, some collisions can occur after the conversion of fixture classes. It may be good to have a closer look at the fixture classes implemented in the Libphonenumber project, to see how often constructors are used in such classes, and if the fixture class conversion should be extended to deal with more complicated cases, maybe even with fixture classes using constructors with arguments.
- Another main limitation of the plug-in is its difficulty handling projects imported over CMake[11]: the file conversion works, but can not be finalized, since the converted targets are only virtual (see Figure 5.1). Researches have shown that CMake is widely used to build and import C++ projects into Eclipse. It would be an improvement to find a way to deal with such projects.

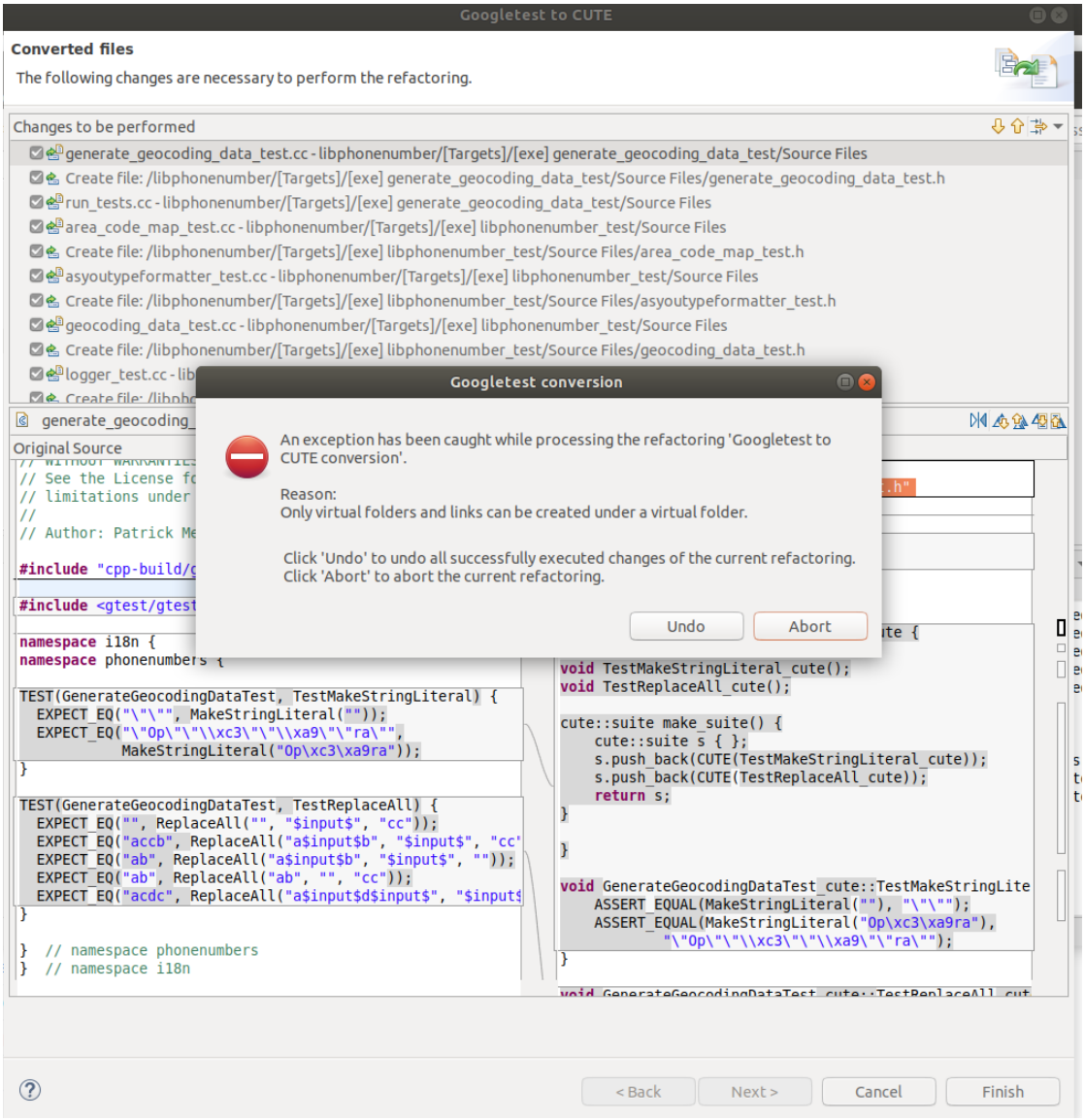


Figure 5.1: *Error dialog popping up after pressing the "Finish"-button to finalize the conversion of CMake "Targets"*

- For the Libphonenumber conversion, all assertions supported by the plug-in were sufficient. Nevertheless, it would be interesting to look at some other real-life projects to see if further assertions should be supported.
- Other limitations are listed at the end of the user manual (see section 5.3). It may be interesting to look at them to decide where something should/could be done.

5.3 User Manual

Before starting, make sure that the *prerequisites* for the conversion are satisfied:

- Your C++ (test) project is imported in Eclipse CDT[1]
- It was imported as normal C++ project (and not, for instance, using CMake[11])
- The CUTE plug-in[7] is installed
- Your C++ project is built

To *start the conversion*, right click on the project, folder or file you want to convert and choose "CUTE → Convert GTest to CUTE" in the menu (see Figure 5.2).

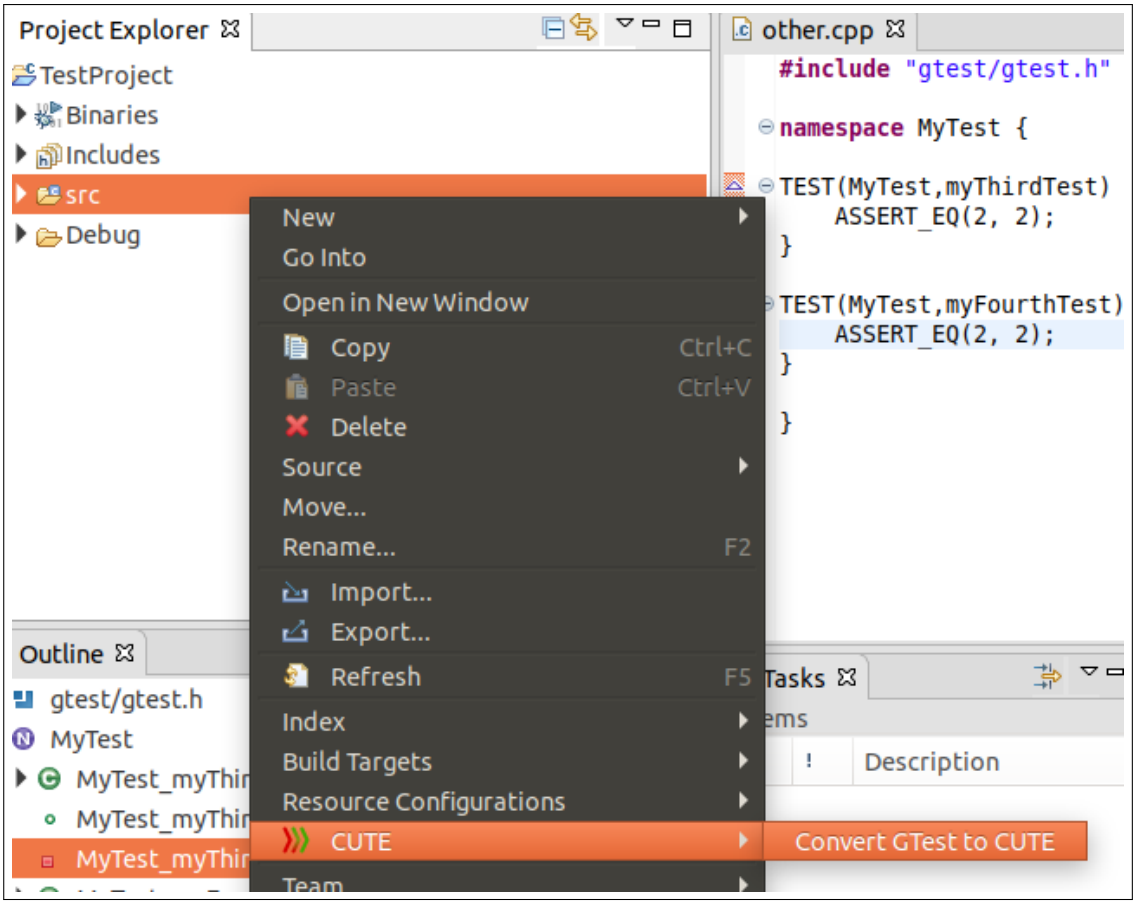


Figure 5.2: *Starting the conversion process in Eclipse*

After all files have been processed (depending on the size of the project, this step may take some time), the *changes are visualized* in a preview wizard (see Figure 5.3). In the

upper part of the wizard, you'll find a list of all suggested changes. Selecting one of them will show you more details concerning the change:

- For any implementation file that contained a Googletest macro (TEST or TEST_F), you should be able to see the differences between the old and the new (CUTE) version of the file
- For any converted file, a header file should have been newly created and selecting it should show you the content of the new file
- Toward the end of the list, you should see the newly created "Cute_Main.cpp"-file, which contains your new main function to run all the converted tests
- The last element of the list should be the change "Add CUTE Nature", which allows you to automatically add the nature needed to run the CUTE tests (selecting this change will show nothing)

You can select and unselect any change listed in the preview wizard. When you eventually click the "Finish" button, the conversion is *concluded* and all selected change are definitely operated. It is still possible to go back to your original files using the common "undo" Eclipse functionality. *Build* the converted tests and try to run them. Depending on the size and the complication level of the converted project, some adjustments still may be needed before the tests can run properly.

Known limitations

Some problems are *handled directly* by the plug-in:

- If you convert a file containing an assertion that is still not supported by the plug-in, a warning notice should appear in the wizard, indicating you the file concerned by the warning (see Figure 5.4). Going to the actual file (in the project, not in the wizard) should allow you to find the actual assertion, since a marker should show on the left side of it.
- If not any Googletest case is found, but a Googletest main function is, the following warning should come up: "Not any Googletest have been found to be converted".
- Another warning should appear if the same test suite name is used over two different files (see Figure 5.5).
- If a header file is created, but another headed file with the same name existed before the conversion, an error message will pop up after pressing the "Finish" button, explaining where the error comes from.

The plug-in also have some *other known limitations*:

- Since a new namespace is created for each test suite, some internal dependencies may not work anymore after the conversion without being adjusted manually.

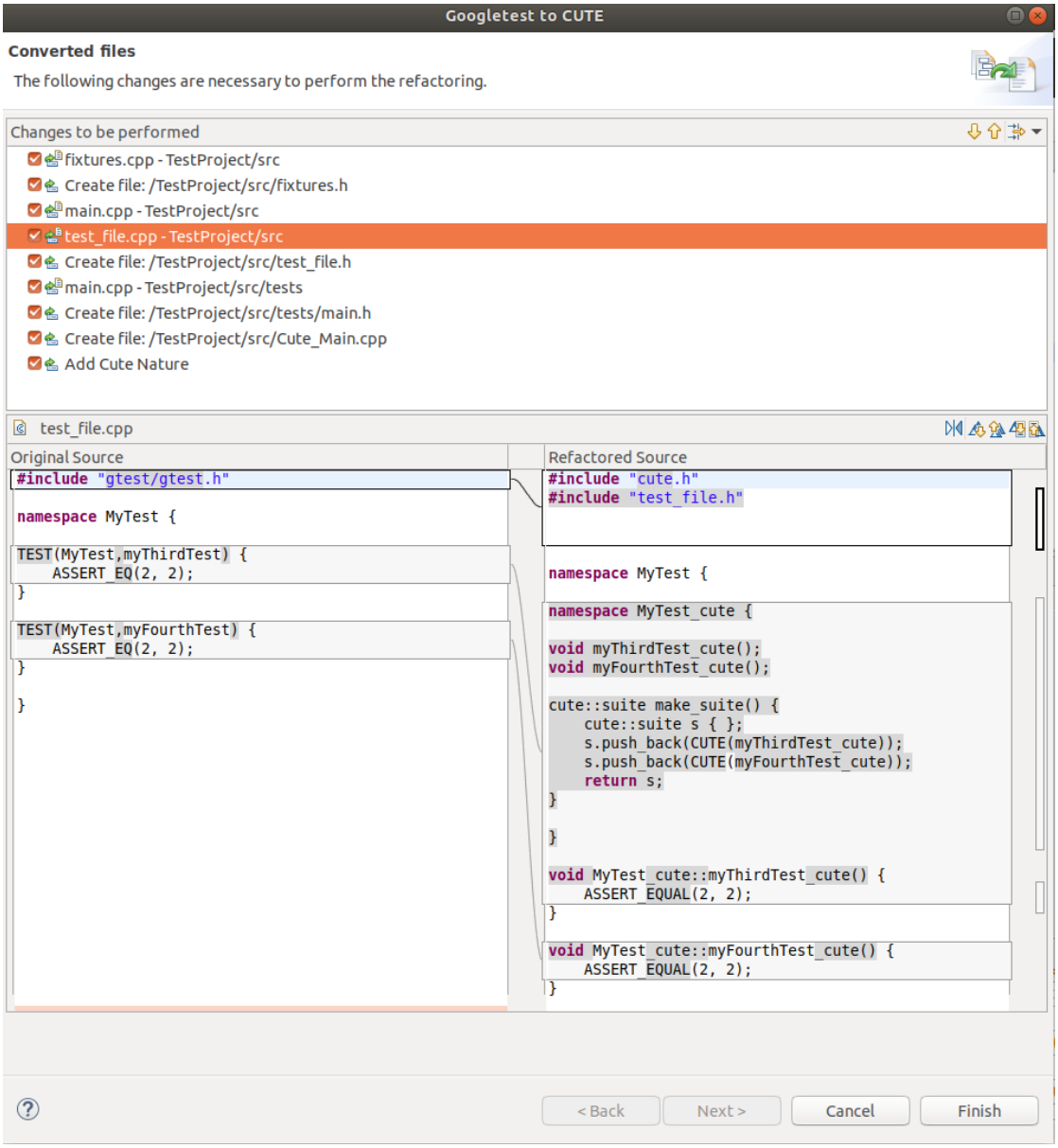


Figure 5.3: *Preview from all suggested changes*

- Any Googletest macro or assertion put in a header file will not be converted, since the plug-in only operate changes on implementation files.
- Any file containing a Googletest assertion but not any TEST or TEST_F macro will not be converted.
- If a TEST and a TEST_F macro share the same name for a test case, the conversion

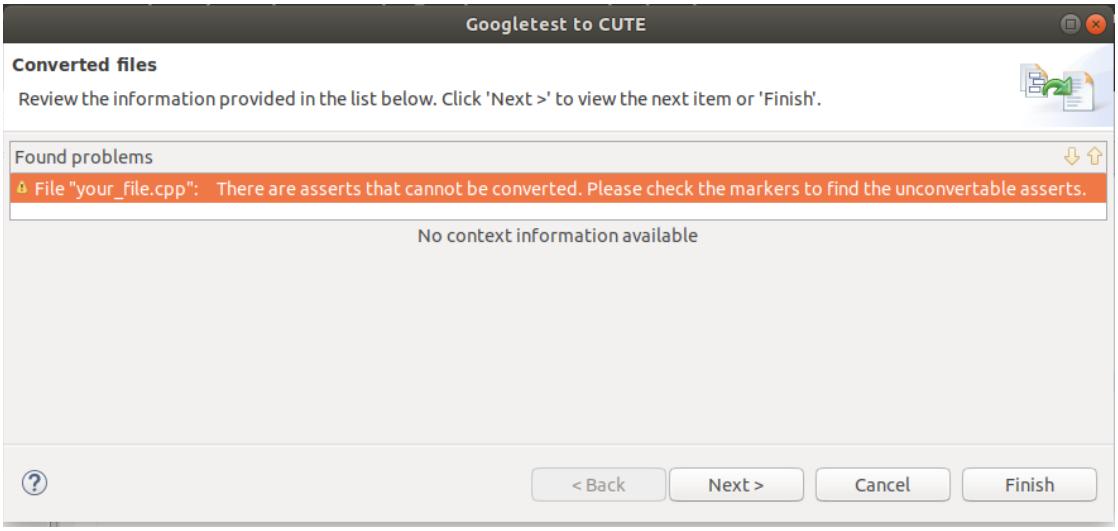


Figure 5.4: *Warning for not supported asserts*

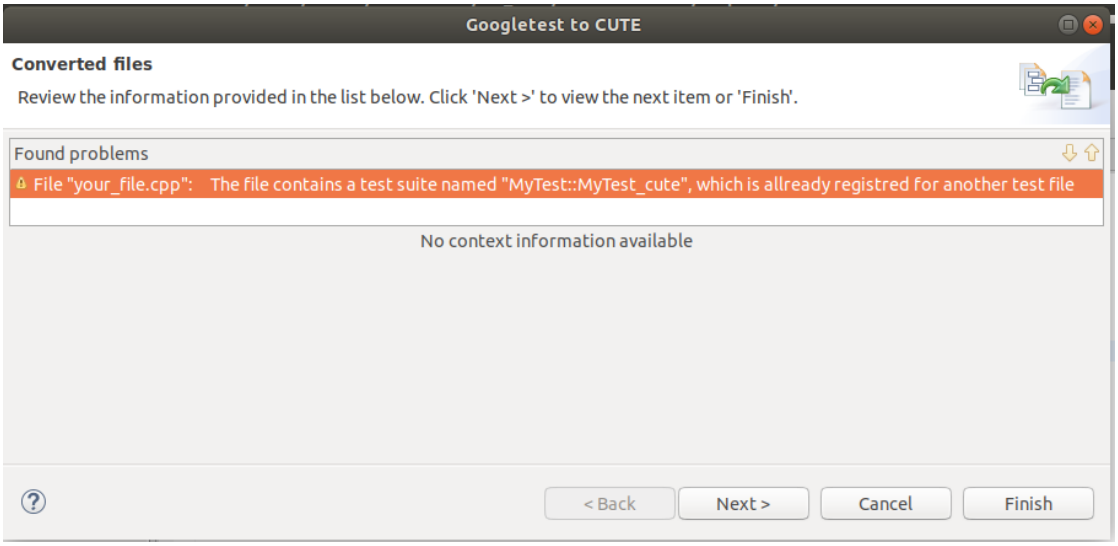


Figure 5.5: *Warning when two different files use the same test suite name*

will result in a name collision, since two different functions sharing the same name will be taking care of the registration of different tests (see Listing 5.1).

- If a fixture class implements one SetUp()-function as well as a constructor, the conversion results in a fixture struct having two constructors.
- The plug-in doesn't support the conversion of fixture classes using constructors with arguments.

```
#include "cute.h"
#include "test_file.h"

namespace MyTest_cute {

void myNormalTest_cute();

cute::suite make_suite() {
  cute::suite s { };
  s.push_back(CUTE(myNormalTest_cute));
  return s;
}
}

namespace MyTest_cute {

struct MyTest {
  MyTest() {
  }

  void myFixtureTest_cute();
};

cute::suite make_suite() {
  cute::suite s { };
  s.push_back(CUTE_SMEMFUN(MyTest, myFixtureTest_cute));
  return s;
}
}

void MyTest_cute::myNormalTest_cute() {
  ASSERT_EQUAL(2, 2);
}

void MyTest_cute::MyTest::myFixtureTest_cute() {
  ASSERT_EQUAL(2, 2);
}
```

Listing 5.1: *Name collision after the conversion of two TEST and TEST_F macros sharing the same suite name*

- The plug-in only changes a file when it finds TEST or TEST_F macros, so that any Googletest ASSERT (or EXPECT) standing alone in a separate file and called by a function from another file will not be converted by the plug-in.
- Any fixture class standing alone in a file that doesn't contain any TEST_F macro will also not be converted.

6 Conclusion

At the end of the semester, the "Googletest to Cute"-plugin was a step further. The first goal of implementing a solution suitable for tests spreading over several translation units was reached and a whole project can now be converted at once. The second goal of being able to convert the widely used Googletest's TEST_F macro is reached too, but with some limitations: Any fixture class initializing the test objects differently than by overriding the SetUp()-function or implementing a simple constructor with no argument will not be converted correctly. Without further analysis, it is difficult to say if this limitation concerns a lot of real-life projects, and what exactly the conversion plug-in should support or not regarding to fixture classes.
Further on, some bugs remaining of the original plug-in were solved, and the user can now better understand what's happening when a problem occurs, since an accurate error handling that distinguishes between simple warnings and information about serious problems has been implemented.
If would have been a great success if the plug-in would have worked on the Libphonenumber project[5] in the end of the project. But regarding to the given time and the complication level of the Libphonenumber project, it seems normal that problems remain (see section 5.2). The next step would be to look for other real-life projects using Googletest to better define the functionalities the plug-in should implement to be considered as a robust conversion tool.

Bibliography

[1] Eclipse (2019). Eclipse CDT (C/C++ Development Tooling). https://www.eclipse.org/cdt. Accessed: 22.05.2019.

[2] Google (2019a). Google search result for 'best testing framework c++'. https://www.google.ch/search?q=best+testing+framework+c%2B%2B&oq=best+testing+framework+c%2B%2B&aqs=chrome..69i57.873j0j8&sourceid=chrome&ie=UTF-8. Accessed: 25.02.2019.

[3] Google (2019b). Googletest Advanced. https://github.com/google/googletest/blob/master/googletest/docs/advanced.md. Accessed: 26.05.2019.

[4] Google (2019b). Googletest Primer. https://github.com/google/googletest/blob/master/googletest/docs/primer.md. Accessed: 25.02.2019.

[5] Google (2019c). Libphonenumber. https://github.com/googlei18n/libphonenumber. Accessed: 27.02.2019.

[6] IFS (2019a). Cute documentation. https://cute-test.com/guides. Accessed: 25.02.2019.

[7] IFS (2019b). Cute main page. https://cute-test.com. Accessed: 25.02.2019.

[8] IFS (2019c). IFS main page. https://ifs.hsr.ch/. Accessed: 25.02.2019.

[9] IFS (2019d). IFS private GitLab. https://gitlab.dev.ifs.hsr.ch/. Accessed: 22.05.2019.

[10] IFS (2019e). Cevelop main page. https://www.cevelop.com. Accessed: 29.05.2019.

[11] Kitware (2019). CMake main page. https://cmake.org/. Accessed: 28.05.2019.

[12] LaTex (2019). LaTex main page. https://www.latex-project.org/. Accessed: 30.05.2019.

[13] TeXstudio (2019). TeXstudio main page. https://www.texstudio.org/. Accessed: 30.05.2019.

[14] R. J. J. V. Erich Gamma, Richard Helm. *Design Patterns : Elements of Reusable Object-Oriented Software*. Pearson Education (US), 1997.

[15] S. Gschwind and R. Venzin. *Googletest to CUTE Converter*. Hochschule für Technik Rapperswil, 2018.

[16] S. Tobias. *Cevelop Plug-in Development.* IFS, 2016.