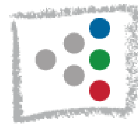


CUTE Plug-in

Greenbar for C++

User Manual

HSR – University of Applied Sciences Rapperswil

Institute for Software

February 17, 2014

1 Overview

CUTE is a header only unit testing framework for C++. For being most efficient using such a framework, it is mandatory to have tight integration into the IDE at hand. The CUTE plug-in provides this integration for the C++ Development Tooling (CDT), a C++ IDE for Eclipse. The plug-in provides the following features:

CUTE Executable Project A C++ test project configured with the CUTE headers and boilerplate source code for a simple test case. It can directly be compiled to a test executable, similar to a hello world test.

CUTE Test Suite Test suites can be used for neat structuring of unit tests. Suites can either be added to an existing CUTE project or separated into their own new project.

CUTE Library Project A C++ test project targeting at testing a library project. Such projects can be automatically connect to other existing library projects, configured with the required dependencies.

Result Visualization After execution of the CUTE test application the results of the unit tests are visualized with a green or red bar. For each failing test a diff view can be displayed to see the expected and actual result.

Rerunning Specific Tests Tests can be executed selectively from the CUTE unit test results view. This avoids executing the complete set of unit tests when only a small subset is of interest.

Test Generation and Registration The CUTE plug-in provides support in writing test code by generating and registering test functions. It also recognizes if a test is not registered and supports the programmer in adding tests to suites.

Test Driven Development Support When writing tests before the actual functionality which is to be tested, much source code can be generated for types, functions, variables, etc. out of the test context. The CUTE plug-in recognizes undeclared identifiers, wrong parameter numbers and missing operators and provides one-click generation of scaffolding code.

In addition to the basic functionality, there are additional plug-ins for the CUTE plug-in, augmenting its capabilities with mock object support, test code coverage visualization, integration of boost headers and useful automated refactorings. The available extensions are described in section 6.

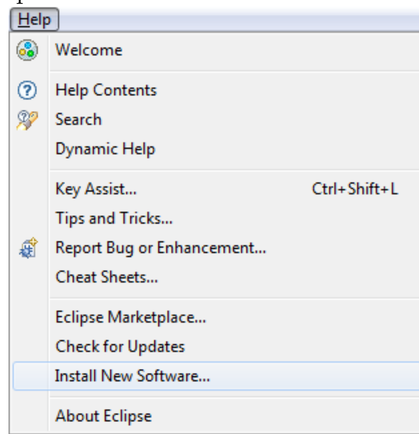
2 Installation

Installing the CUTE plug-in is performed using the CUTE update site [1]. This is NOT a download page!

2.1 Installation Steps

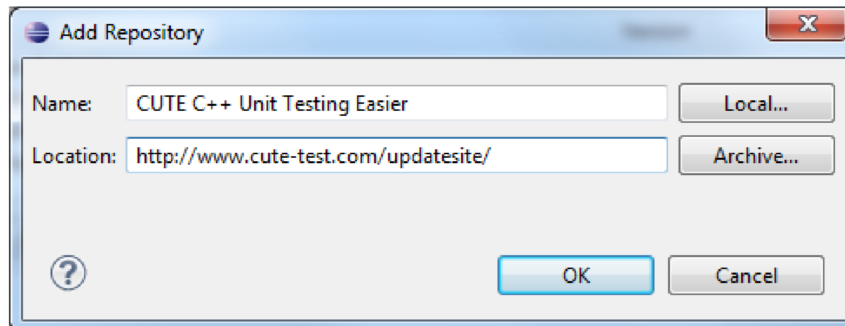
To install the CUTE plug-in perform the following steps:

1. In your Eclipse CDT choose `Help > Install New Software` in the menu to start the Eclipse software installer.



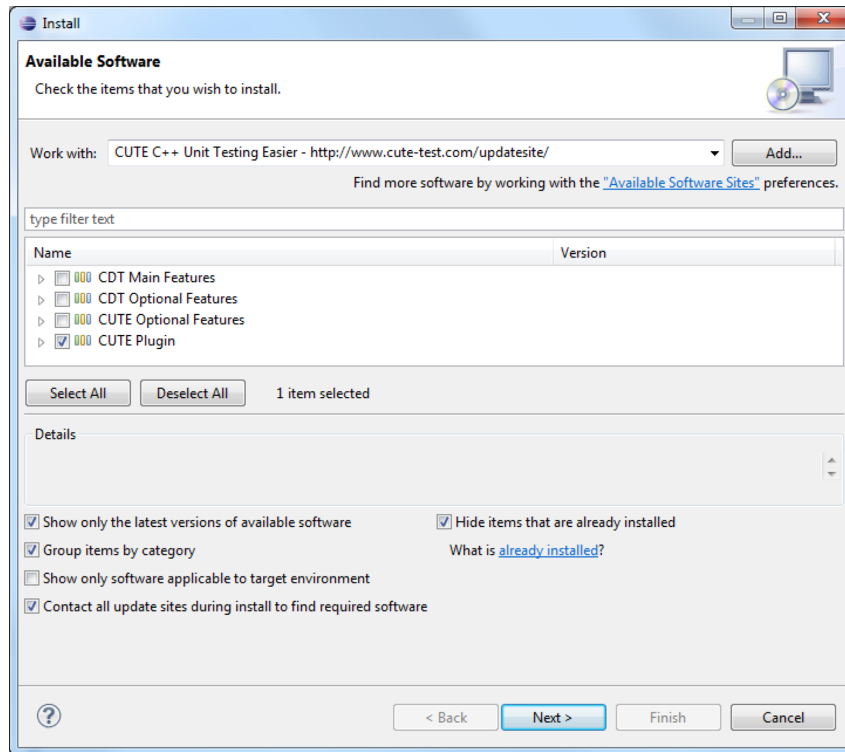
2. Click `Add...` to add the CUTE repository

3. Type (or copy) `http://www.cute-test.com/updatesite` into the `Work with:` text field and confirm with enter.



4. Select the CUTE Plug-in feature:

[1] http://www.cute-test.com/updatesite



5. Click the button `Next`. Eclipse will then resolve dependencies and show the install dialog.

6. Read and accept the licence agreements and click `Finish` if you want to continue installing CUTE and agree with the licences.

After the installation of the CUTE plug-in you need to restart the Eclipse IDE.

2.2 Requirements

For installing the CUTE plug-in the Eclipse CDT (C++ Development Tooling) Kepler release is required for satisfying all required dependencies. The latest version of Eclipse CDT can be downloaded from the Eclipse download site [2]. The latest version the CUTE plug-in has been tested with is CDT 8.2.1, the service release 1 for Kepler.

2.2.1 Header Dependencies

The latest CUTE headers (version 2.0) require to be compiled with a C++11 compliant compiler, like GCC 4.8 [3]. When using CUTE headers with earlier

[2] http://www.eclipse.org
[3] http://gcc.gnu.org/

versions of C++ there is a dependency to Boost headers [4]. For providing a convenient integration of Boost into CUTE projects, there is a separate plug-in. Please refer to section 6.1 for more information.

2.3 Earlier Releases

For Eclipse CDT versions before Kepler the CUTE plug-in versions are still available through the corresponding update sites:

- Helios - http://www.cute-test.com/updatesite/helios
- Indigo - http://www.cute-test.com/updatesite/indigo
- Juno - http://www.cute-test.com/updatesite/juno

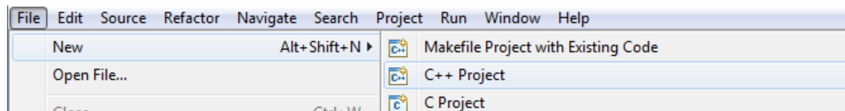
These versions of the CUTE plug-in are not updated anymore. Thus, they might lack some functionality which is available in the latest version of the plug-in.

3 First Steps

This section gives a step-by-step introduction to the projects available in the CUTE plug-in.

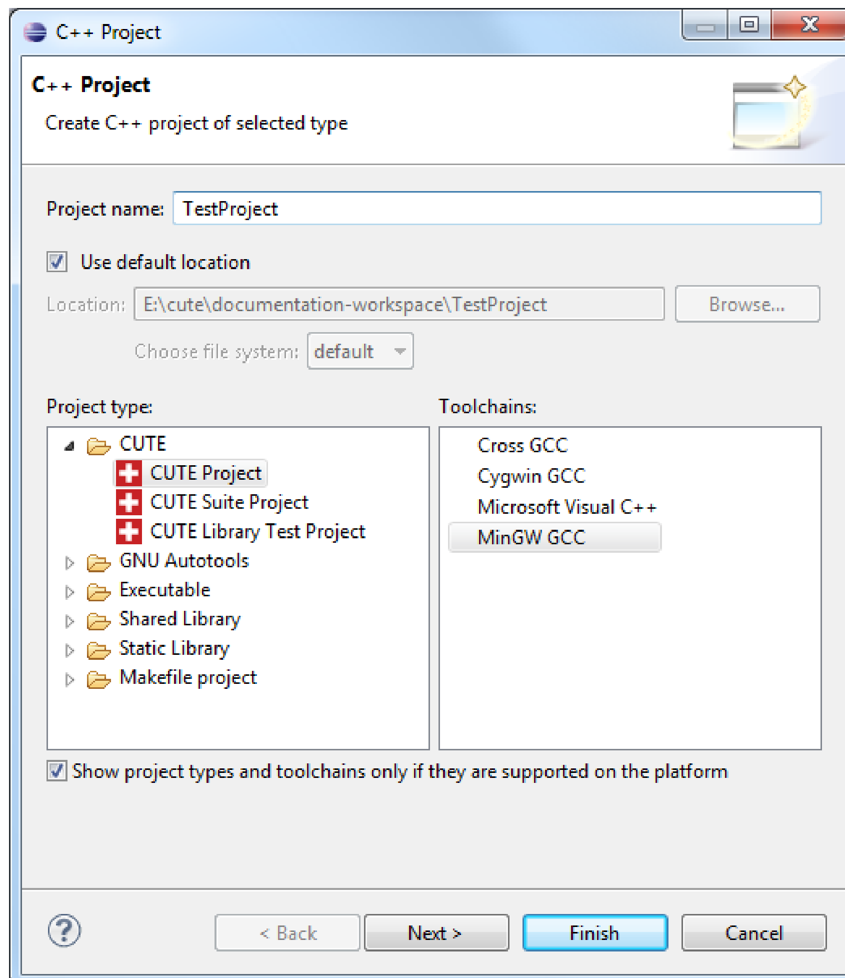
3.1 Executable Project

1. Create a new CUTE project:

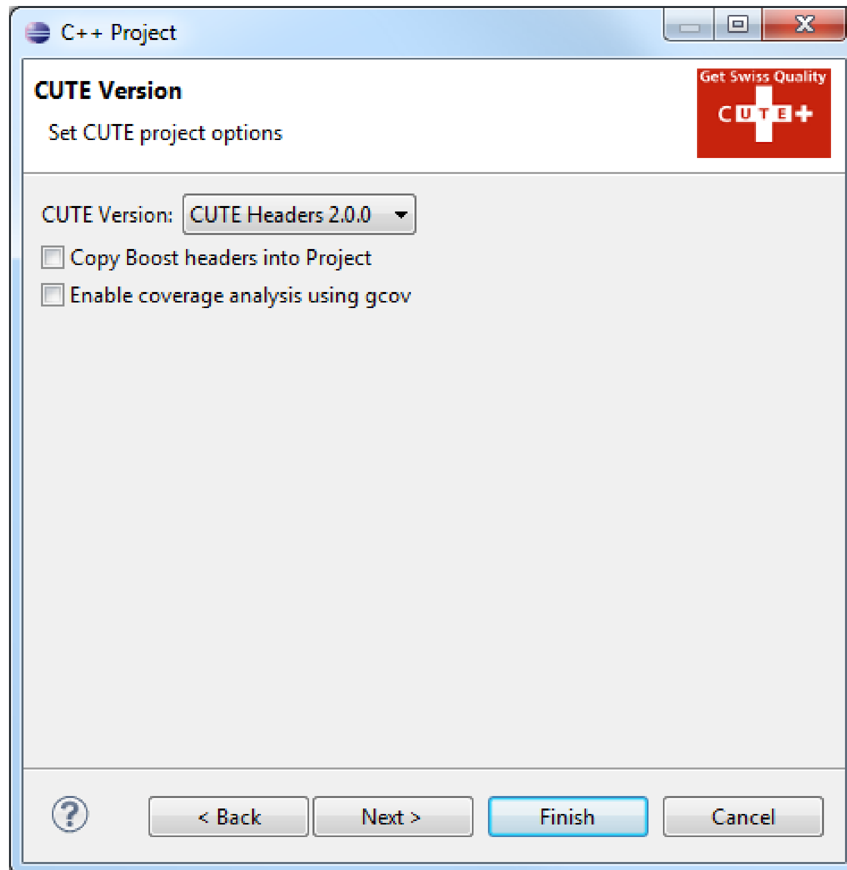


2. - In the `C++ Project` dialog select `Test Project` in the `CUTE` category.
   - Choose the toolchain you would like to use for compiling your test.
   - Enter the name of your test project.
   - Click `Next >`.

[4] http://www.boost.org/



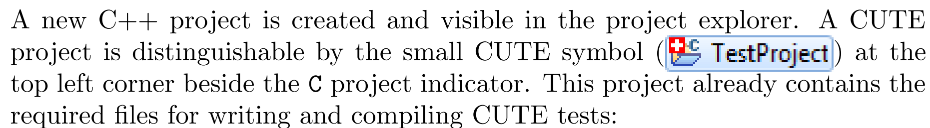
3. In the CUTE version dialog you can select the CUTE header version you intend to use. Using the latest version is recommended.



Optional: There might be additional configuration options, depending on additional plug-ins you might have installed. You can enable the integration of Boost headers or coverage visualization generated from Gcov data. Refer to the corresponding plug-in descriptions in section 6 for detailed information.

4. Click `Finish` for creating the new project.

3.1.1 Project Contents

A new C++ project is created and visible in the project explorer. A CUTE project is distinguishable by the small CUTE symbol () at the top left corner beside the `C` project indicator. This project already contains the required files for writing and compiling CUTE tests:

- `src` directory containing a `Test.cpp` file
- `cute` directory containing the CUTE headers selected in the CUTE version dialog.

- The project is configured to use the `cute` directory as additional include directory. Thus it is not required to write additional parts of the CUTE header paths into include directives.

3.1.2 Generated Test Source

The contents of the `Test.cpp` file consists of the required `main` function which creates a test suite. There is also a minimal failing test case, which is registered in the test suite.

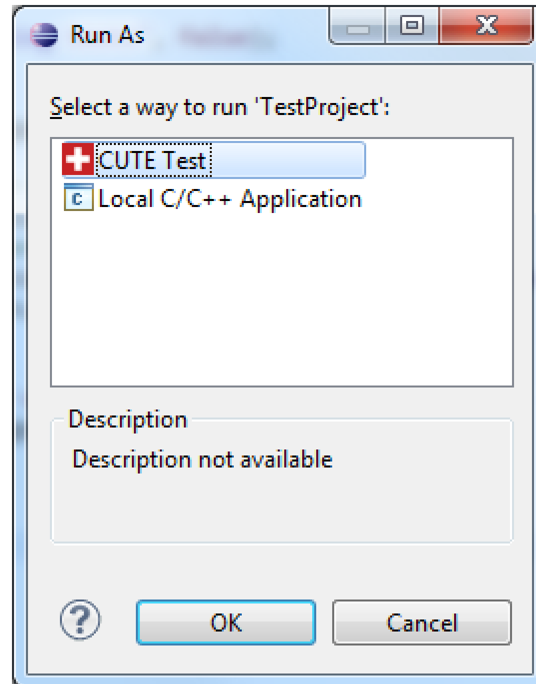
```
#include "cute.h"
#include "ide_listener.h"
#include "xml_listener.h"
#include "cute_runner.h"

void thisIsATest() {
  ASSERTM("start writing tests", false);
}

void runAllTests(int argc, char const *argv[]){
  cute::suite s;
  //TODO add your test here
  s.push_back(CUTE(thisIsATest));
  cute::xml_file_opener xmlfile(argc,argv);
  cute::xml_listener<cute::ide_listener<> > lis(xmlfile.out);
  cute::makeRunner(lis,argc,argv)(s, "AllTests");
}

int main(int argc, char const *argv[]){
  runAllTests(argc,argv);
  return 0;
}
```

This code should be compilable and executable as is, depending on your environment. Your compiler should be capable of compiling C++11 or you need to have the headers of Boost available in your include path. See section 2.2 for more details.

3.1.3 Compiling & Running Tests

Compiling the test project is straightforward like compiling any other project with CDT. Just press the build button ( ) for building your project. The project should be compiled with no errors. This creates an executable in the `Debug` or `Release` directory, depending on the selected build configuration.

Running the compiled tests is invoked by pressing the `Run` button (). A dialog will appear, asking for a run configuration to be created for you executable project. Select `CUTE Test` and press `OK`. The tests will be executed.



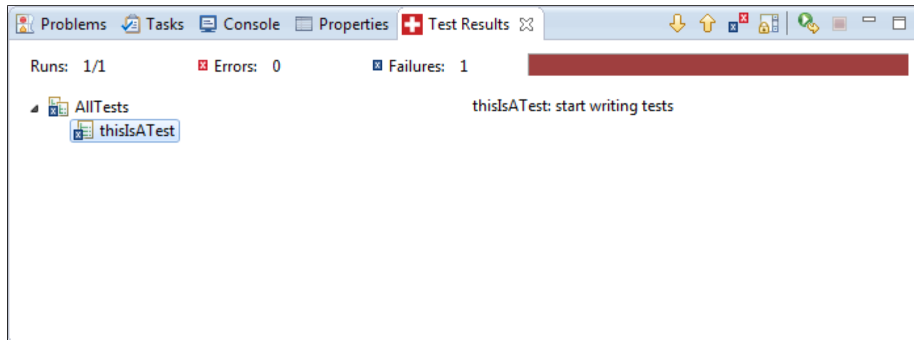
3.1.4 Test Result

Console Output After executing the tests the result will appear in the Eclipse console. Running the generated test will result in the following console output:

```
#beginning AllTests 1

#starting thisIsATest

#failure thisIsATest ..\src\Test.cpp:7 thisIsATest: start writing tests

#ending AllTests
```

The indicated failure in the output contains a link to the location of the assert that has failed. It is clickable and Eclipse will jump to the corresponding location in the source code for easy access to the failed test.

Test Result View When having large test output the console is not the convenient way for checking test results. The CUTE plug-in provides an additional test result view to visualize the results properly.



The Test Result view is a dedicated window provided by the CUTE plug-in for easy access to the test results. It summarizes the output of the unit tests as follows:

- Green or red bar to indicate whether an error or failure occured.
- Number of executed tests
- Number of errors - Tests failed due to an unexpected problem, like an uncaught exception.
- Number of failures - Test failed at an assert as the result to be checked was not as expected.

The test suites and test cases are displayed hierarchically in a tree. Each node has a marker indicating the test result: `Success`, `Failure` or `Error`. Unsuccessful tests can be selected. If available the message provided by the assert is displayed on the right hand side. If no message is given the expected value and the effective result are both displayed for comparison. This provides a useful insight on why the test failed.

XML Output As it is not feasible to use the test result view on a build server and parsing the console output can be tedious, there is an additional possibility to access the result of CUTE tests. With the current `Test.cpp` template an XML test result file is automatically generated. This file is located in the project root by default and named like the test executable plus `.xml`, e.g. `TestResult.exe.xml`. This test result XML file can be used by a build server to evaluate test results.

Test Result : AllTests

1 failures (±0)

2 tests (+1)
Took 0 ms.
add description

All Tests

Test name	Duration	Status
thisIsARunningTest	0 ms	Passed
thisIsFailingTest	0 ms	Failed

3.1.5 Executing Specific Tests

It is not required to always run all CUTE tests. If only a specific test or a specific subset of tests shall be executed, it is possible to select the tests which need to be executed in the `Test Results` view, right-click them and select `Rerun Selected` from the context menu.

Problems Tasks Console Properties Search Test Results

Runs: 3/3 Errors: 0 Failures: 1

SuiteName
testWeatherIsNice
testProofNPisInP
testMosquitosAre
Rerun Selected
Rerun Test

3.2 Library Project

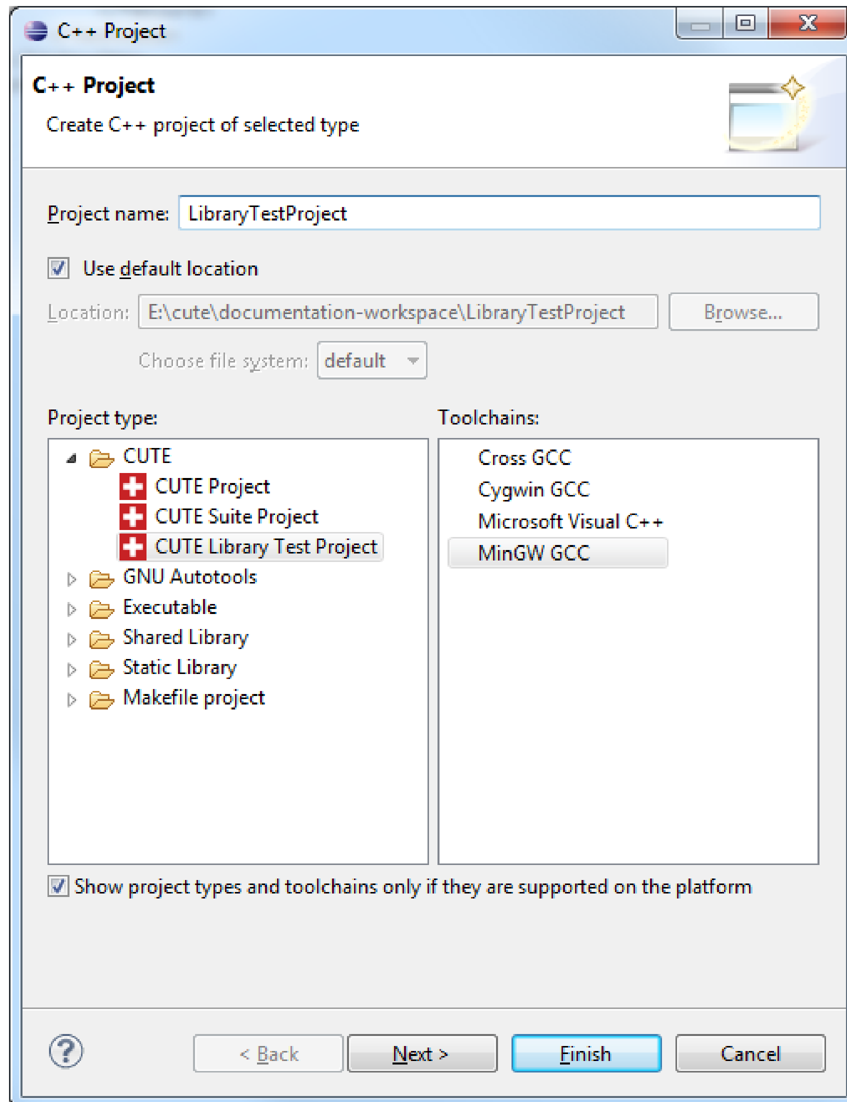
If you want to add a test suite for an existing library project, CUTE provides assistance in setting up the test project. Similar to creating a normal CUTE project you can create a library test project as follows:

1. Create a new CUTE project:

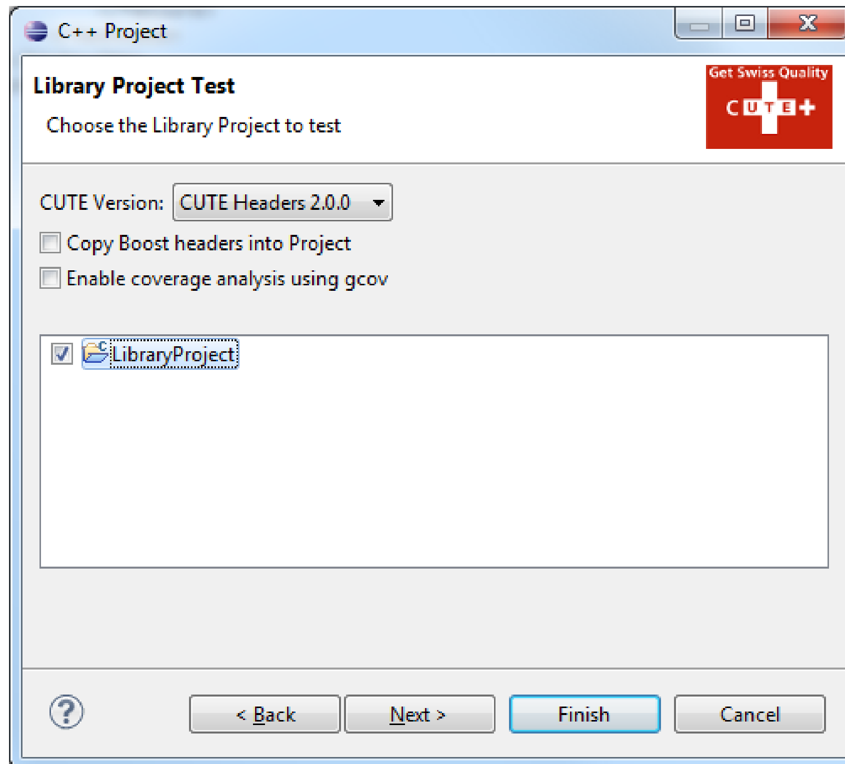
   File Edit Source Refactor Navigate Search Project Run Window Help
   New Alt+Shift+N ▸ Makefile Project with Existing Code
   Open File... C++ Project
   C Project

2. • In the `C++ Project` dialog select `CUTE Library Test Project` in the `CUTE` category.

- Choose the toolchain you would like to use for compiling your test.
- Enter the name of your test project.
- Click `Next >`.



3. In the CUTE version dialog you can select the CUTE header version you intend to use. Using the latest version is recommended. Here you can also select the library project to be tested. This selection is mandatory.



Optional: There might be additional configuration options, depending on additional plug-ins you might have installed. You can enable the integration of Boost headers or cover visualization generated from Gcov data. Refer to the corresponding plug-in descriptions in section 6 for detailed information.

4. Click `Finish` for creating the new library test project.

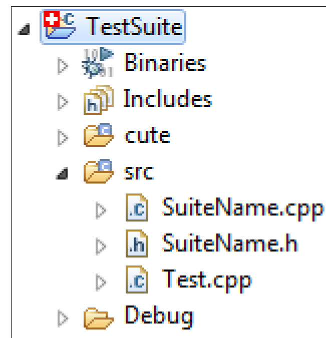
The created library test project is automatically configured to have a dependency to the selected library project. The following configuration options are affected:

- Include path is extended by the library's project path.
- The library project is added as a library for the test project, including extended library path and the corresponding library name.
- A project dependency is added from the test project to the library project.

These settings could be added to a normal CUTE project manually as well. They are located in the project properties of the test project: `Properties > C++ General > Paths and Symbols`

3.3 Test Suite

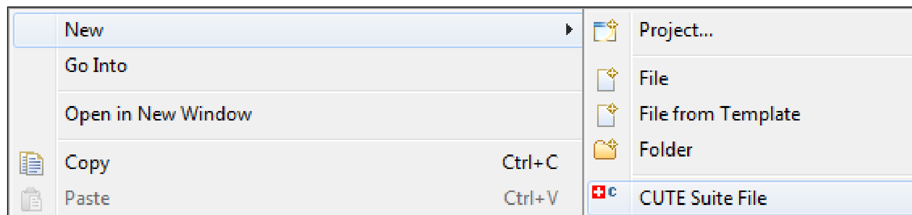
The third project kind, `CUTE Test Suite`, creates a CUTE project including a separated named test suite. It is created like the a common CUTE test project, but it allows to explicitly name the suite during creation.



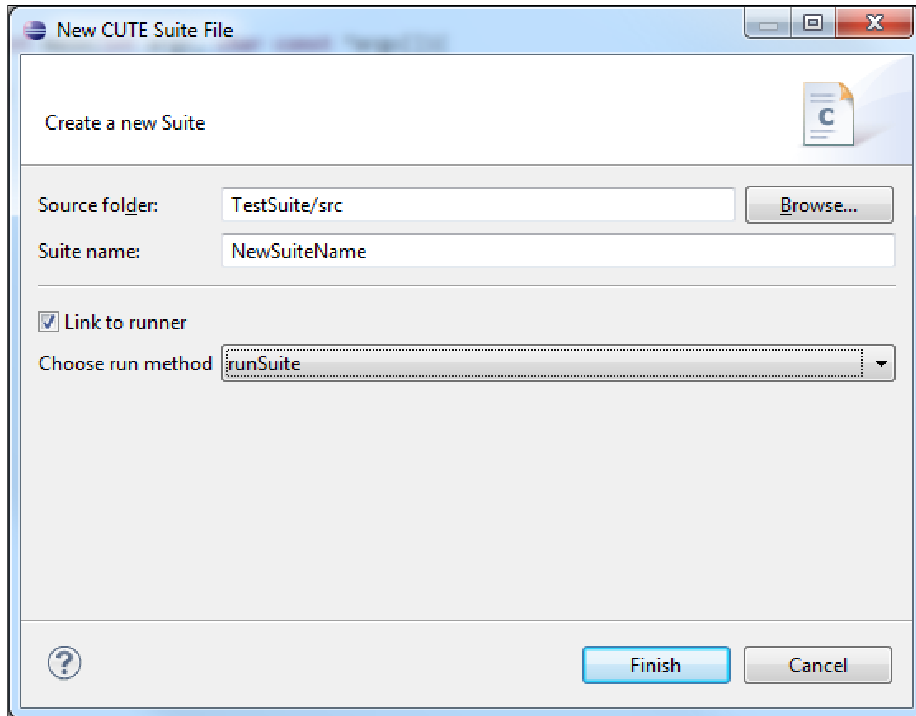
The suite is properly separated into header and source files, named according to the suite name.

3.4 New Test Suite

It is possible to add a new suite to an existing CUTE project.

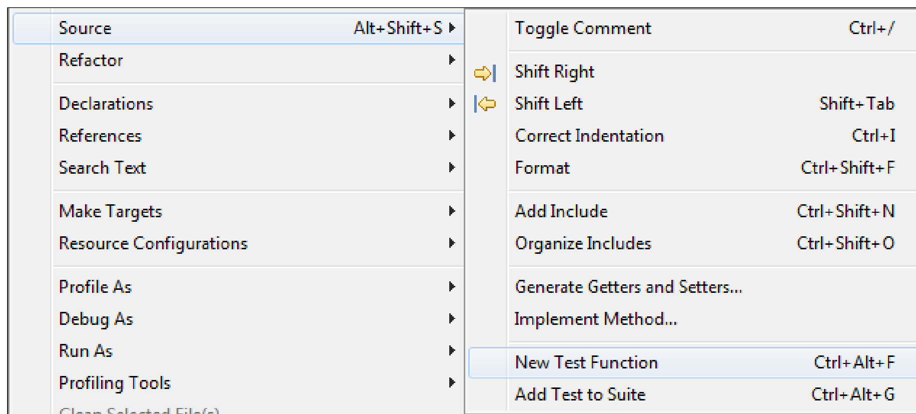


The newly created suite can be added to an existing test runner at creation.



3.5 New Test Function

The CUTE plug-in supports the user by creating boilerplate code for a new test function, including the registration of the newly created test. This function is accessed through the context menu or by pressing `Ctrl + Alt + F`.



A new test function is created:

```
void newTestFunction() {
  ASSERTM("start writing tests", false);
}
```

The created test function is also registered in the test suite. Through linked edit mode the test name can be changed directly in all occurring places.

```
void runAllTests(int argc, char const *argv[]){
   cute::suite s;
   s.push_back(CUTE(thisIsATest));
   s.push_back(CUTE(newTestFunction)); //Newly created test
   cute::xml_file_opener xmlfile(argc,argv);
   cute::xml_listener<cute::ide_listener<> > lis(xmlfile.out);
   cute::makeRunner(lis,argc,argv)(s, "AllTests");
}
```

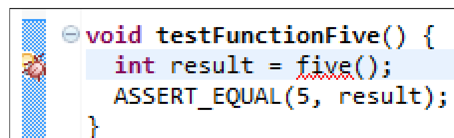
4 TDD Features

When developing with a test driven approach, the test case is written before the productive code. From the test code some information can be extracted about the program elements to be tested. As those elements might not already exist, it is possible to generate them partially from the locations in the test where they occur.

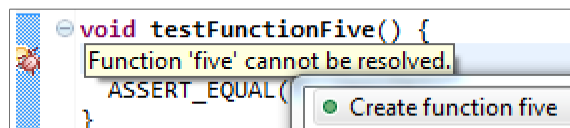
As an example let us consider the following test case:

```
void testFunctionFive() {
   int result = five();
  ASSERT_EQUAL(5, result);
}
```

Obviously, there should be a function with the name `five`, which has the return type `int`. The CUTE plug-in recognizes this and marks the code of the missing function with a CodAn maker, a small red bug icon.



The plug-in suggests a resolution for this bug. This allows to generate the scaffolding code required to make the test at least compile. Of course, program logic still has to be added manually.



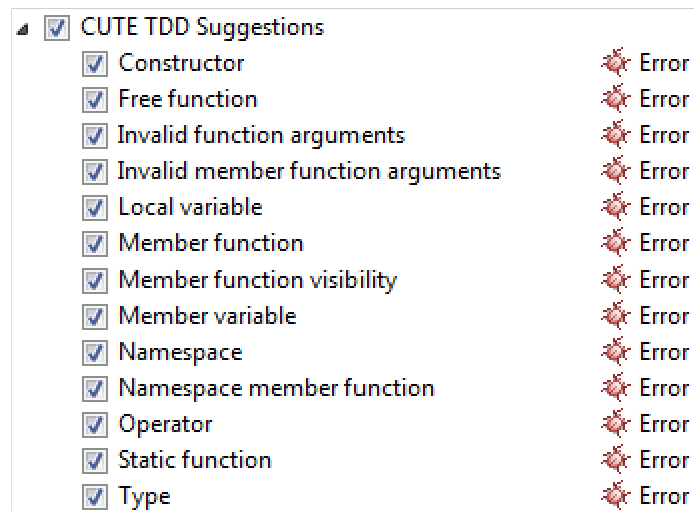
This so called quickfix is able to deduce the return type of the function and creates a corresponding function definition in the same file. This definition can then easily be moved to the desired location.

```
int five() {
    return int();
}
```

Such issues can be recognized outside test code as well and therefore, can be very useful in efficient development. The CUTE plug-in provides markers and quickfixes for the following cases:

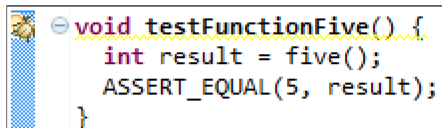
- Constructors
- Member and non-member functions
- Invalid (member) function arguments
- Local and member variables
- Function visibilities
- Namespaces
- Operators
- Static functions
- Types

Semantic checks for creating the markers can be configured in the preferences of CDT: `Window > Preferences > C/C++ > Code Analysis`

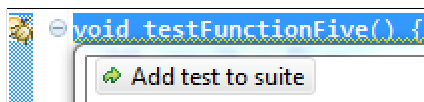


4.1 Unregistered Tests

The CUTE plug-in recognizes if a test function or a test functor is not registered in any test suite. It marks the affected test function with a CodAn warning marker. A test function is recognized as such if it has an empty parameter list and an `ASSERT` macro call inside.



For this kind of CodAn marker a resolution is provided, which registers the test in the test suite.




```
void runAllTests(int argc, char const *argv[]){
  cute::suite s;
  s.push_back(CUTE(thisIsATest));
  s.push_back(CUTE(testFunctionFive)); //Newly registered test
  cute::xml_file_opener xmlfile(argc,argv);
  cute::xml_listener<cute::ide_listener<> > lis(xmlfile.out);
  cute::makeRunner(lis,argc,argv)(s, "AllTests");
}
```

5 CUTE Headers

The CUTE - C++ Unit Testing Easier test framework is a header only unit test framework for C++. It can be used independently of the plug-in for Eclipse CDT. All source code of the framework is available and delivered along with the plug-in. Each project created with the plug-in consists of its own set of header files. If the headers should be stored in one common place on the file system, the project setups have to be adapted accordingly. This does not affect the functionality of the plug-in though.

5.1 Functionality

We will not delve into the internals of the CUTE framework deeply here. Details can be found in the source code or described in the corresponding online resources [5].

CUTE provides the following assert macros:

ASSERT Takes a condition which must evaluate to true.

[5] http://www.cute-test.com/projects/cute/wiki/Theory_of_Operation_and_Goals

ASSERT_EQUAL Takes two arguments which must be equality comparable and expects this comparison to be true.

ASSERT_EQUAL_DELTA Takes three arguments: Expected, actual and delta values. This assert is used for assertion of floating point values.

ASSERT_GREATER Takes two arguments which must be comparable and expects the first to be greater than the second. left > right shall be true.

ASSERT_LESS Takes two arguments which must be comparable and expects the first to be less than the second. left < right shall be true.

ASSERT_LESS_EQUAL Takes two arguments which must be comparable and expects the first to be less or equal than the second. left <= right shall be true.

ASSERT_NOT_EQUAL Takes two arguments which must be equality comparable and expects this comparison to be false.

ASSERT_THROWS Takes two arguments: A piece of code (usually a function call) and the type the code is expected to throw.

FAIL Takes a condition which must evaluate to false.

Every assert macro is also available with an `M` suffix, which stands for message. The corresponding assert takes an additional first argument, which is the message to be emitted if the assert fails. Since version `2.0` the CUTE headers also support data driven tests.

5.2 Dependencies

The templates of the CUTE headers rely on specific headers of the boost library:

- bind.hpp
- function.hpp
- type_traits

Those headers have to be available for compiling CUTE unit tests. If they do not exist in the build system there is an additional plug-in available, which provides the required headers and adds them to a newly created CUTE project. See section 6.1 for more information.

The dependency to Boost is obsolete when compiling C++11 code, as CUTE, since header version 2.0, provides full functionality using C++11 features. Recent compilers should be able to compile C++11 code, which we strongly recommend to use.

Furthermore, it is possible to disable `iostream` dependency for embedded devices. In order to do this you need to set the `DONT_USE_IOSTREAM` symbol to `1`.

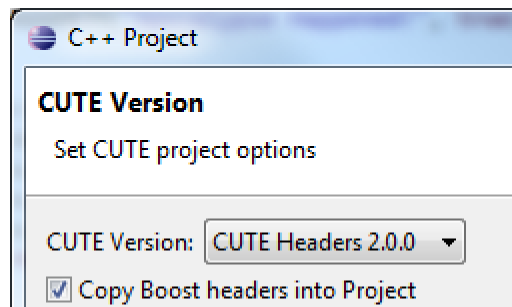
6 Additional Plug-ins

Beside the CUTE plug-in, several other plug-ins are available for CUTE. Some depend on the core functionality of the CUTE plug-in and others can be used independently.

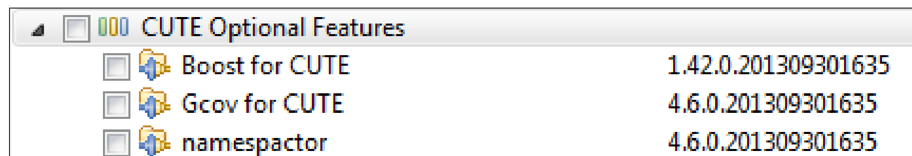
6.1 Boost

CUTE headers, when compiled with former C++ standards (before C++11), rely on specific Boost headers. If in the target environment Boost is not available CUTE cannot be used. To have a convenient integration of Boost, an additional Boost plug-in for the CUTE plug-in is available. This plug-in contains Boost headers which are copied on demand into new CUTE projects. The plug-in also configures the project to use the directory of the copied Boost headers as include path.

Include Boost Headers Option When the Boost plug-in is available the second page of the `New CUTE Project` wizard is augmented with the option to include the headers in the created project.



Installation The Boost plug-in is available at the update site of the CUTE plug-in itself [6]. It is selectable in the `CUTE Optional Features` group.



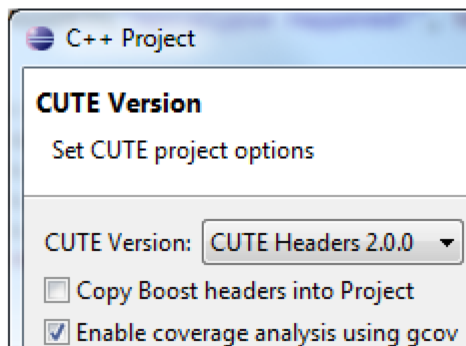
This plug-in contains additional licencing information, because of the distribution of the Boost headers. Before the plug-in can be installed, the corresponding license agreement has to be accepted.

[6] http://www.cute-test.com/updatesite

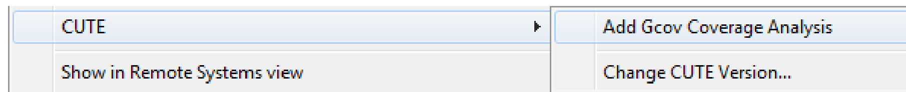
6.2 Gcov

C++ compilers of the GNU Compiler Collection have the capability to compile the program to emit coverage information when executed. With the CUTE Gcov plug-in this information can be visualized in Eclipse. For making this information available, the tests have to be compiled with specific compiler options set.

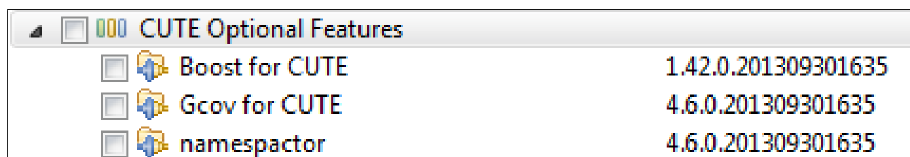
Enable Gcov When the Gcov plug-in is available the second page of the `New CUTE Project` wizard is augmented with the option to display coverage information after test execution. The Gcov plug-in automatically adds the required flags for compilation to the miscellaneous compiler options in CDT: `-fprofile-arcs -ftest-coverage`



It is possible to add and remove display of coverage information after the project has been created through the context menu of the project. The menu entry `CUTE` in the context menu is only available for CUTE projects.



Installation The Gcov plug-in is available at the update site of the CUTE plug-in itself [7]. It is selectable in the `CUTE Optional Features` group.



The plug-in is an extension to the CUTE plug-in and requires it to be installed, subsequently, it cannot be used standalone.

[7] http://www.cute-test.com/updatesite

Visualization If enabled, after running the CUTE tests, the Gcov information is retrieved from the created Gcov files.

```
int executed() {
    return 1;
}

int partially(bool condition) {
    if (condition) {
        return 2;
    }
    else {
        return 3;
    }
}

int omitted() {
    return 4;
}
```

Executed Code that has been executed completely is highlighted in green.

Partially Code that has been executed partially is highlighted in yellow.

Omitted Code that as not been executed at all is hightlighted in red.

6.3 Mockator

Mock objects and seams are very important when creating unit tests for program code. The Mockator [8] plug-in provides exceptional support for refactoring towards seams and creating test doubles. The following seams are supported:

Object Seam Based on inheritance to inject a subclass with an alternative implementation. Mockator helps in extracting an interface and in creating the missing test double including all used member functions.

Compile Seam Inject dependencies at compile-time through template parameters. Extract a template parameter and Mockator creates the missing test double including all used member functions.

Preprocessor Seam With the help of the preprocessor, Mockator redefines function names to use an alternative implementation.

Link Seam : Mockator supports three kinds of link seams:

- Shadowing functions through linking order (override functions in libraries with new definitions in object files)

[8] http://www.mockator.com

- Wrapping functions with GNU's linker option -wrap (GNU Linux only)
- Run-time function interception with the preload functionality of the dynamic linker for shared libraries (works only with GNU Linux and MacOS X)

Installation The Mockator plug-in is independent of the CUTE plug-in and can be obtained separately. It is available at an external update site [9]. For detailed information please refer to its separate documentation.

6.4 Namespactor

Namespactor is an additional plug-in available. It provides namespace-related automated refactorings for C++. If the plug-in is installed the following additional automated refactorings are available in the `Refactor` menu in CDT.

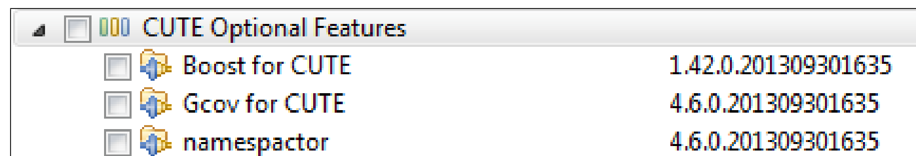
Extract Using Namespace Directive... Introduces a using directive and removes the then obsolete qualifiers from declarators. This refactoring should not be used in header files.

Extract Namespace Using Declaration... Instead of extracting all qualifiers from one namespace, this functionality allows to extract a single qualified name into a using declaration. Which leaves a shorter name at the place of usage.

Inline Using... Changes the identifiers in the code to include their namespace or class prefix, by eliminating using namespace directives or using declaration.

Quality Unqualified Name... Qualifies an existing unqualified identifier from a namespace or a class with its name qualified.

Installation The Namespactor plug-in is independent of the CUTE plug-in and can be obtained separately. It is available at the CUTE update site [10] in the `CUTE Optional Features` group.



[9] http://www.mockator.com/update
[10] http://www.cute-test.com/updatesite