Generating Test Cases for Regression Testing and Exception Finding

You can capture functional snapshots in test cases for regression testing. Test cases can also identify conditions that could result in exceptions, which may result in system and application instability, security vulnerabilities (such as denial of service attacks), poor performance and application response time, and frequent down time.

Sections include:

About Automated Test Case Generation

C++test automatically generates test cases according to the parameters defined in the Test Configuration's Generation tab. These test cases use a format similar to the popular CppUnit format.

Generating Tests to Verify New Functionality

If you want to verify the functionality of new code, we recommend that you automatically generate 1- 2 tests per function to start.

After you generate and execute these tests, you can then extend the test suite with user-defined test cases as described in Extending and Modifying the Test Suite.

Generating Tests for Regression Testing

If you want to create a snapshot of the code's current behavior to establish a regression testing baseline (e.g., if you are confident that the code is behaving as expected), you can run a test using the "Unit Testing> Generate Regression Base" built-in Test Configuration. When this Test Configuration is run, C++test will automatically verify all outcomes.

These tests can then be run automatically, on a regular basis (e.g., every 24 hours) to verify whether code modifications change or break the functionality captured in the regression tests. If any changes are introduced, these test cases will fail in order to alert the team to the problem.

During subsequent tests C++test will report tasks if it detects changes to the behavior captured in the initial test. Verification is not required.

With the default settings, C++test generates one test suite per source/header file. It can also be configured to generate one test suite per function or one test suite per source file (see Customizing Generation Options for details).

Safe stub definitions are automatically-generated to replace "dangerous" functions, which includes system I/O routines such as `rmdir()`, `remove()`, `rename()`, etc. In addition, stubs can be automatically generated for missing function and variable definitions (see Understanding and Customizing Automated Stub Generation for details). User-defined stubs can be added as needed (see Adding and Modifying Stubs for details).

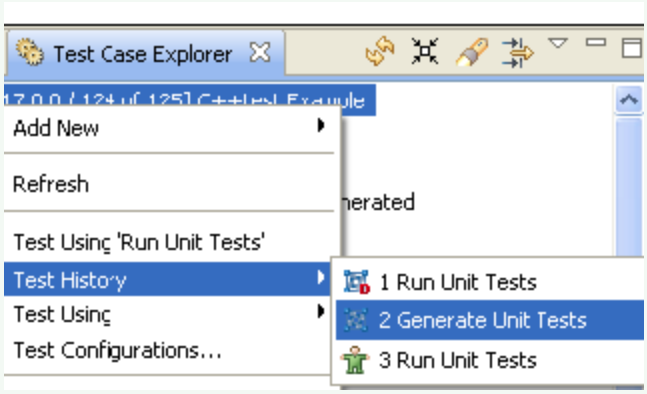
Generating Test Cases

The general procedure for test case generation is:

1. Identify or create a Test Configuration with your preferred test generation settings.
   - For a description of preconfigured Test Configurations, see Built-in Test Configurations.
   - For details on how to create a custom Test Configuration, see the Configuring Test Configurations and Rules for Policies. Details on C++test-specific options are available at Configuring Test Configurations.
2. Run the Test Configuration
   - For details on testing from the GUI, see Testing from the GUI.
     For details on testing from the command line, see Testing from the Command Line Interface.

Tip - Generating Tests from the Test Case Explorer

You can generate tests for a project directly from the Test Case Explorer (which can be opened by choosing **Parasoft> Show View> Test Case Explorer**). Just right-click the project node in the Test Case Explorer, then choose the desired test generation Test Configuration from the **Test History** or **Test Using** shortcut menu.



For details about the Test Case Explorer, see Exploring the C++test UI.

3. Review the generated test cases.
    - For details, see Reviewing Automatically-Generated Test Cases.
4. (Optional) Fine-tune test generation settings as needed.
    - For details, see Generation Tab Settings - Defining How Test Cases are Generated.

Customizing Generation Options

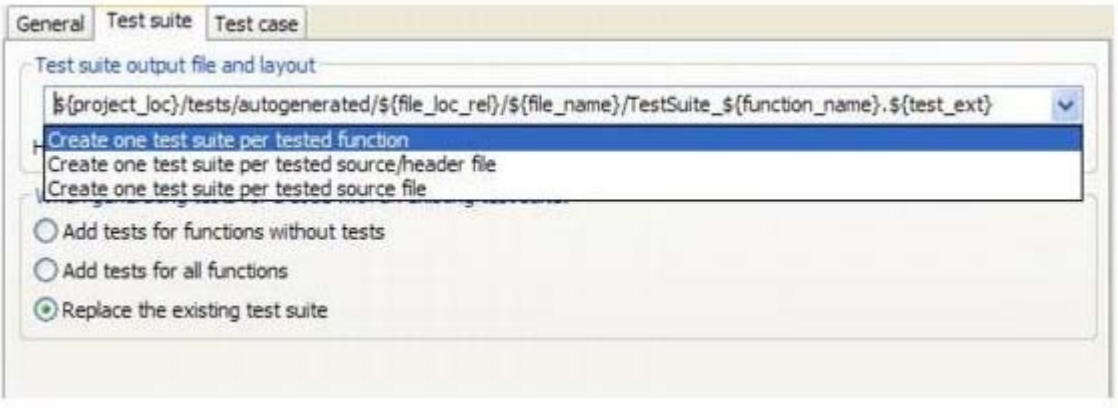
You can control a number of generation options by customizing the options in the Test Configuration's Generation tab.

Controlling the Test Suite's File Name, Location, and Layout

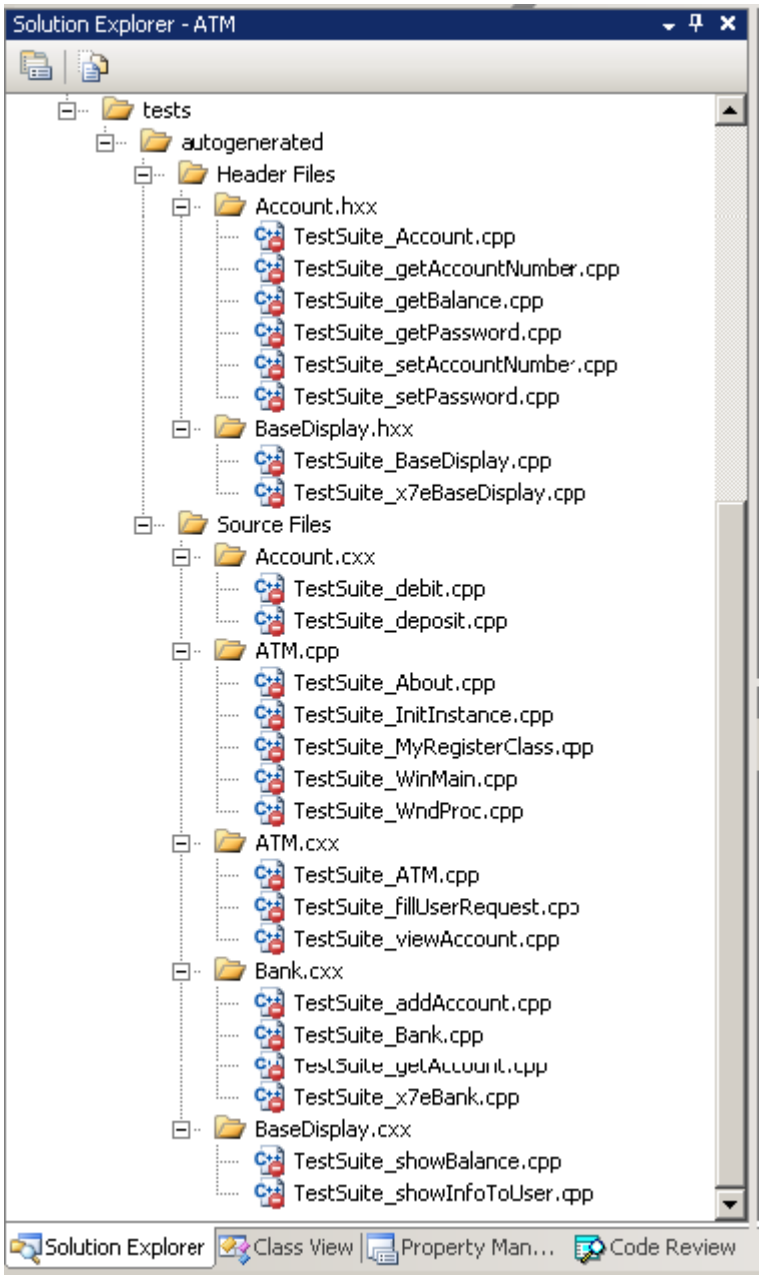
The generated test suite's file name, location, and granularity/layout can be controlled by customizing options in the Test Configuration's **Generation> Test suite** tab.

To change the default test suite output settings, first select one of the following three pre-defined output and layout options from the **Test suite output file and layout** box:

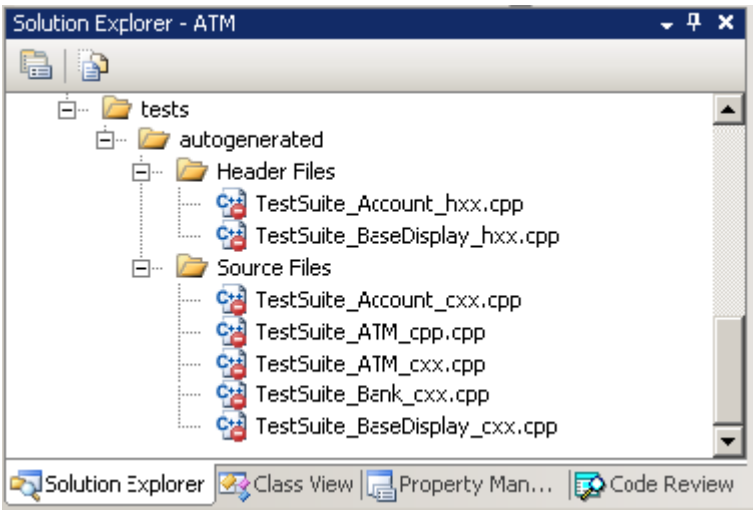
- Create one test suite per function
- Create one test suite per source/header file
- Create one test suite per source file



If the **Create one test suite per function** option is selected, a test suite generated for the sample ATM project (included in the examples directory) would look like this:



If the **Create one test suite per tested source/header** option is selected, a test suite generated for the sample ATM project (included in the examples directory) would look like this:



After selecting one of these options, you can customize the pattern as needed (for instance, to generate tests into the source location). You can use the following variables when you are customizing the pattern:

- ${test_ext} - C++test-specific extension of a test suite file (.cpp).
- ${file_name-} File name.
- ${file_base_name-} File name without extension.
- ${file_ext-} File extension.
- ${file_loc-} File location.
- ${file_loc_rel-} File location relative to the project root.
- ${file_uid-} File unique identifier.
- ${function_name-} Tested function name.
- ${function_uid-} Tested function unique identifier (hash-code computed from the function signature/mangled name).
- ${src_file_name-} Name of context (source) file. (A "context file" is a source file that describes the compilation unit in which the given tested function is defined).
- ${src_file_base_name-} Name of context (source) file without extension.
- ${src_file_ext-} Extension of context (source) file.
- ${src_file_loc-} Context (source) file location.
- ${src_file_loc_rel-} Context (source) file location relative to the project root.
- ${src_file_uid-} Source (context) Context file unique identifier (hash-code computed from the source file location).

Key

- file = The source/header file where the tested function is defined.
- source file = The source file that defines a compilation unit where the tested function is defined.

Warning

Removing some variables can lead to overlapping test suites. C++test will alert you to this by displaying the error message "Test suite output file pattern is ambiguous."

C++test uses the following internal checks and restrictions related to changing the basic patterns for location of generated tests:

- All automatically generated tests are of "included" type (the test suite file is "glued "together with the given source file/compilation unit).
- It tries to prevent cases where test cases for functions from different compilation units would be placed in the same test suite file (because it is not possible to correctly glue such a test suite file with original source file).
- C++test has different variables (which are resolved based on the file under test, its name, its location etc.) that can be used to make the test suite file pattern unambiguous (in terms of the item above).
- One commonly-used strategy is to generate a test suite file into a file/location that has the original file name/location in it. This the default pattern:

  ${project_loc}/tests/autogenerated/${file_loc_rel}/
  TestSuite_${file_base_name}_${file_ext}.${test_ext}

  This is not ambiguous because ${file_loc_rel} and ${file_base_name} variables are used (even though there are a number of files with the same name in the project, their location will be different—and that location will be a part of the test suite file name/location).
- There are also other available variables—for example, ${file_uid}, ${src_file_uid}—that can be used instead of the ${file_loc_rel} / ${file_base_name} pair while keeping the pattern unambiguous. These variables are resolved into a hash code of the original file location. For example, a pattern like

  ${project_loc}/tests/autogenerated/
  TestSuite_${file_base_name}_${file_uid}_${file_ext}.${test_ext}

  will result in the following test suite:

  ATM/tests/autogenerated/TestSuite_Account_d7a5efc6_hxx.cpp

Appending or Replacing Existing Tests

You can also control whether C++test will append or replace existing tests if the generated test file has the same name and location as an existing test suite file. This behavior is determined by the **When generating tests for code with an existing test suite** setting, which provides the following options:

- **Add tests for functions without tests:** C++test will generate test cases for functions without tests. The existing tests will not be affected or modified.
- **Add tests for all functions:** C++test will generate test cases for all functions. The existing tests will not be affected or modified.
- **Replace the existing test suite:** C++test will generate test cases for all functions. The existing test suite will be removed and then replaced with the new one.

How does C++test determine if there are existing test cases for a given function?

It looks for the CPPTEST_CONTEXT and CPPTEST_TEST_SUITE_INCLUDED_TO markers inside the test suite file.

```
1 #include "cpptest.h"
2
3 /* CPPTEST_CONTEXT("/ATM/ATM.cxx"); */
4 /* CPPTEST_TEST_SUITE_INCLUDED_TO("/ATM/ATM.cxx"); */
5
6 class TestSuite_ATM_cxx_85315171 : public CppTest_TestSuite
7 {
8     public:
```

Common Test Generation Goals

The following table explains how to configure the Test Configuration's generation options to accomplish common test generation goals. Options covered include the **Generation> General** tab's **Generate tests for code** option and the **Generation> Test suite** tab's **When generating tests for a code with an existing test suite** option.

Goal	Settings
To generate an initial set of tests	For **Generate tests for code**, enable **Without test suites**. Specify additional parameters (function access level, output file location/name etc.).

To update an existing automatically-generated test suites with tests for new functions (do not generate new test suites)	For **Generate tests for code**, enable **With up-to-date test suites With out-of-date test suites**. For **When generating tests for a code with an existing test suite**, enable **Add tests for functions without tests**. Specify additional parameters (function access level, output file location/name etc.).
To synchronize automatically-generated tests with the current code - append missing tests, create missing test suites	For **Generate tests for code**, enable **Without test suites**, **With up-to-date test suites**, and **With out-of-date test suites**. For **When generating tests for a code with an existing test suite**, enable **Add tests for functions without tests**. Specify additional parameters (function access level, output file location/name etc.).
To fully reset existing automatically-generated tests	For **Generate tests for code**, enable **Without test suites**, **With up-to-date test suites**, and **With out-of-date test suites**. For **When generating tests for a code with an existing test suite**, enable **Replace the existing test suite**. Specify additional parameters (function access level, output file location/name etc.).

Note

C++test keeps all auto-generated tests in the original project location. The appropriate directory structure (similar to the Solution Explorer view) will be created automatically, using the project directory as a root location.

Choosing the Layout Option That Suit Your Goals

This section explains how to configure the **Test suite output file and layout** option (in the Test Configuration's **Generation> Test suite** tab) to suit various layout needs. To help you understand how each option discussed translates to actual projects, we show how it would affect the following sample project:

```
MyProject
 Header Files
   MyClass.h // contains foo() definition
 Source Files
   MyClass.cpp // contains bar() and goo() definitions
```

To generate a single test suite file for each function, keep tests in a separate directory

Use `${project_loc}/tests/${file_loc_rel}/${file_name}/TestSuite_${function_name}.${test_ext}`

Sample layout:

```
MyProject
 Header Files
   MyClass.h
 Source Files
   MyClass.cpp
 tests
   Header Files
     MyClass.h
        TestSuite_foo.cpp // contains tests for foo()
   Source Files
     MyClass.cpp
        TestSuite_bar.cpp // contains tests for bar()
        TestSuite_goo.cpp // contains tests for goo()
```

To generate a single test suite file for each source/header file, keep tests in a separate directory

Use `${project_loc}/tests/${file_loc_rel}/TestSuite_${file_base_name}_${file_ext}.${test_ext}`

Sample layout:

```
MyProject
 Header Files
   MyClass.h
 sSource Files
   MyClass.cpp
 tests
   Header Files
     TestSuite_MyClass_h.cpp // contains tests for foo()
   Source Files
     TestSuite_MyClass_cpp.cpp // contains tests for bar() and goo()
```

To generate a single test suite file for each source/header file, keep tests with the tested files

Use `${project_loc}/${file_loc_rel}/tests/TestSuite_${file_base_name}_${file_ext}.${test_ext}`

Sample layout:

```
 Header Files
   MyClass.h
   tests
     TestSuite_MyClass_h.cpp // contains tests for foo()
 Source Files
   MyClass.cpp
   tests
     TestSuite_MyClass_cpp.cpp // contains tests for bar() and goo()
```