Tutorial - Unit Test Generation and Execution

This lesson contains a set of exercises that cover various aspects of unit testing. Using the information provided in the introduction, along with testing practices learned in the exercises, you can formulate a testing philosophy that meets your specific requirements. For details on performing testing and coverage analysis in C++test, refer to the Test Generation and Execution section of the C++test User's Guide.

In this section:

Prerequisites

The ATM project must be available in yourworkspace.See Tutorial - Creating a C++test Project for details on how to achieve this.

Introduction to C++test Unit Testing

Any test activity requires definition and understanding of:

- Testing goals/requirements
- Test metrics
- Test strategy
- Test budget

Isolated Versus Project Scope Unit Tests

One consideration in choosing an approach is whether to perform unit testing in isolation or perform API tests on the full project scope.

- Unit testing in isolation implies creating tests that generally do not use functions defined outside of the tested file-those are replaced with stubs for the purposes of testing.
- Unit or API testing with project scope creates tests that are fully coupled-tests that use any and all external functions available to the tested unit. The result is that the number of required stubs is significantly less, and each test in the set can exercise one or more functions, therefore resulting in more overall test coverage relative to the number of tests.

There are trade-offs associated with either method, but the project requirements and the determined testing phase of the SDLC typically dictate the unit testing approach. C++test supports both approaches, as well as a creative mix of the two.

Isolation Unit Test Considerations

- Project test requirements mandate unit test in isolation (with the goal of validating low-level functionality before code integration).
- Tests are required to be submitted together with newly-developed source code. Test in isolation can be performed as soon as the file in question compiles.
- The project is being developed from scratch, and in many cases most of the dependency components or classes are not available to the developer writing tests.

For small sets of files, unit tests in isolation generally require less work to configure and run. Processing a much smaller file may also substantially reduce the C++test's overall test preparation and runtime. When testing larger sets of files isolation, care should be exercised when when defining stubs and managing stub files.

Note that tests developed during test-in-isolation can generally be reused for project-level testing in a later test cycle. Coupling of the tested code to the rest of the system can be adjusted by defining specific source files, libraries, and stub files to use or ignore for test purposes. All of that can be accomplished via various settings in C++test Test Configurations.

Project (Coupled) Unit Test Considerations

- The project is an incremental modification of an existing code base.
- Validating overall project functionality / use cases is more important than achieving 100% code coverage.

Testing in project scope means that the scope of what you have selected for testing will be expanded to include all symbols defined in the project, unless they are explicitly stubbed out with "user" stubs.

Project-level unit tests will likely take more time to process and prepare in the C++test flow- but this is only wall clock time. The overall effort in setting up project-based tests is generally less.

If all symbols in your project are resolved, the project can be successfully linked, and your primary concern is unit testing to achieve code coverage requirements, then the best method is to initially unit test in project scope. You can still do class-based testing, but as you drive coverage on the selected class, other code within the project will also be tested and coverage reported.

Basic Procedures

Step	Project scope	File scope (isolation)
Phase 1: Set up test configurations	For host-based testing, the built-in configurations for generating tests, generating stubs, and running tests should work without modifications. However, it is a good practice to duplicate the built-in configurations to the user-defined folder to ensure a consistent environment. Always the duplicate configuration so that you can easily identify it and remember its purpose.	Same for both methods
Phase 2: Select class or classes	The test scope will be expanded to include the entire project for the selection. However, only unit tests associated with the selected code will be run.	The test scope can be set by selecting one or more source files in the project. To prevent undefined symbol errors when running unit tests, you should use the same scope or a subset of the scope that was used to generate stubs.
Phase 3: Generate unit tests	Run the Generate Unit Tests configuration to setup the test infrastructure and create test suites. The test generation is not specific to project scope or file scope, so you can generate tests and use them for either testing scheme. After tests are created, you can open the source code for the test suites from the project tree or use the C++test Test Case Explorer to navigate to the generated unit tests	Select the test scope in the project tree and run the Generate Unit Tests configuration. If tests exist for code outside of the unit test scope, they will be ignored. If tests are missing for code within the scope, that code will not be tested. After tests are created, you can open the source code for the test suites from the project tree or use the C++test Test Case Explorer to navigate to generated unit tests.
Phase 4: Generate stubs	The Stubs view provides visibility into symbol usage. A built-in configuration is provided that will load the Stubs view and display the symbol, definition type, and location. If you plan to integrate this configuration into your process, you should duplicate and rename the configuration. Before stub generation, the Stubs view will list symbols that are not available or that are using original definitions (original code is available in the scope). After stub generation, you should see all the symbols defined as Original or Auto. If user stubs are created, they will show as User Definition. To auto generate stubs, select the test scope in the project tree and run the stubs generation configuration. The stubs are created as source code in the project stubs/autogenerated directory by default. If original definitions are available for all symbols, no stubs will be generated. Review the Stubs view to ensure that all symbols are defined.	Same for both methods
Phase 5: Run the unit tests	Select the scope and execute the configuration for running tests. The Test Configuration will generate a test executable using the generated test infrastructure, test suites, and available stubs. The executable runs, creates result files, and loads them into the C++test GUI for analysis. You can also use the C++test Test Case Explorer to view and select one or more test cases to run. If tests exist for code outside of the files or tests you have selected, they will be ignored. If tests are missing for code within the selection, that code will not be exercised.	Same procedure for both methods. Additional note: When selecting the scope for testing, you must select the same scope or a subset of the scope that was used when generating stubs.

Introduction to Unit Test Exercises

The following exercises are presented in the order in which they are mostly liked to be used during project testing. The flow assumes that you must test before the code is complete and all functionality is available. The final exercise covers how to generate reports on your testing efforts.

C++test Test Configurations are used to setup and manage tests. In practice, built-in Test Configurations are used to create templates. You should copy them to the User-defined folder so they can be reviewed and modified to meet your requirements. The exercises below require several test configurations. We will cover the details of each test configuration in the individual exercises. It is assumed from previous lessons that you are familiar with duplicating and modifying test configurations.

Setup for Exercises

To prepare for the following exercises, you need to have a project setup with a fresh copy of the ATM example code, as described in Tutorial - reating a C++test Project.

GNU Host-Based Testing Procedure

1. Create a project directory for unit testing using the C++test ATM example code.
    - For example, `C:\C++test\Tutorial\ATMEclipseGnu\ATM`

2. Copy the following files from `[C++test install directory]\examples\ATM` to the new directory:
    - `Makefile`
    - `Account.cxx`
    - `ATM.cxx`
    - `Bank.cxx`
    - `BaseDisplay.cxx`

3. Create a `C:\C++test\Tutorial\ATMEclipseGnu\ATM\include` directory and copy the following files to the include directory.
    - `Account.hxx`
    - `ATM.hxx`
    - `Bank.hxx`
    - `BaseDisplay.hxx`

4. Make sure that your Windows environment includes paths to `gcc, g++, make`, etc.

5. Start C++test and create a new workspace in the ATM parent directory you created.
    - For example: `C:\C++test\Tutorial\ATMEclipseGnu\workspace`

6. From the C++test Perspective, choose **File> New> Project**.

7. Expand **C++**, select **C++ Project**, and click **Next**.

8. Enter ATM in the **Project name** field.

9 Clear **Use default location**.

10. Browse to the location of the ATM Makefile.
    - For example, `C:\C++test\Tutorial\ATMEclipseGnu\ATM`

11. Under **Project Types**, select **Makefile project> Empty Project**.

12. Under **Toolchain**, select **Cygwin GCC**.

13. Click **Finish**.

14. When you see a dialog asking whether you want to open the associated perspective, respond according to your preferences. You can choose to work in the C/C++ perspective or the C++test perspective.

15. Run static analysis on ATM.cxx as a sanity check for project setup. See Tutorial - Analyzing Code Against Coding Standards for more information or running static analysis.

Exercises

- Exercise 1 - Generating Unit Tests
- Exercise 2 - Using the Test Case Explorer
- Exercise 3 - Reviewing Stub Information
- Exercise 4 - Generate Stubs for Missing or Undefined Functions
- Exercise 5 - Executing Unit Tests on Files in Project Scope
- Exercise 6 - Executing Unit Tests on an Isolated File (File Scope)
- Exercise 7 - Review Unit Test Results and Fix Unit Test Problems
- Exercise 8 - Reviewing Code Coverage Results and Expanding Code Coverage
- Exercise 9 - Running Unit Tests on all Project Files
- Exercise 10 - Reviewing and Verifying Test Outcomes and Creating Regression Tests
- Exercise 11 - Generating Unit Test Reports
- Exercise 12 - Using Data Sources in Unit Tests
- Exercise 13 - Generating Unit Tests Using Factory Functions
- Exercise 14 - Executing Existing CppUnit Tests Under C++test
- Exercise 15 - Debugging Unit Tests With GDB